Hadrien Barral,
Rémi Géraud-Stewart,
Georges-Axel Jaloyan,
David Naccache

École normale supérieure, PSL Research University



The ABC of Next-Gen Shellcoding

What we're interested in today

- Embedded devices, especially smartphones.
- Many people have phones. Some are security-conscious.
- Phones contain a lot of sensitive data: attractive targets.
- The usual hacking toolset is very limited as of now.



Who we are

- Hadrien Barral: Hacker. I like hacking stuff.
- Georges-Axel Jaloyan: PhD student. I like reversing open source binaries.
- Rémi Géraud-Stewart: Hacker. I like whisky and maths.



Why this matters and what is this about

- Creative methods
- . . . to write constrained shellcodes
- . . . on new architectures

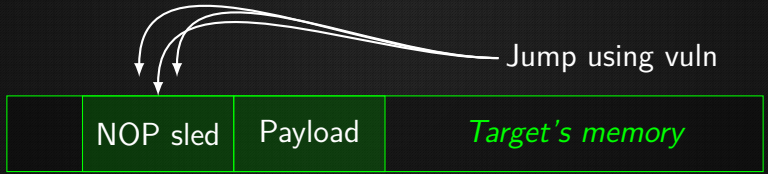
1. This is not the shellcode you're looking for

Shellcode 101

For those hiding in the back

- Code that you wrote (or found) in the target's memory
- That gives you some power (e.g., pop a shell)
- That you can jump to using some vuln (e.g. buffer overflow, UAF...)

Typical scenario: send a carefully crafted string to the target and profit.



Shellcode 102

It's sometimes not as easy

- If treated as a string, the shellcode can't contain \x00
- Stack protections (NX, ...)
- Limited memory available for shellcode
- Jumping to shellcode may be tricky (ASLR...)

Many clever techniques exist to work around these obstacles (ROP, heap sprayin', egglayin', ...)

But...

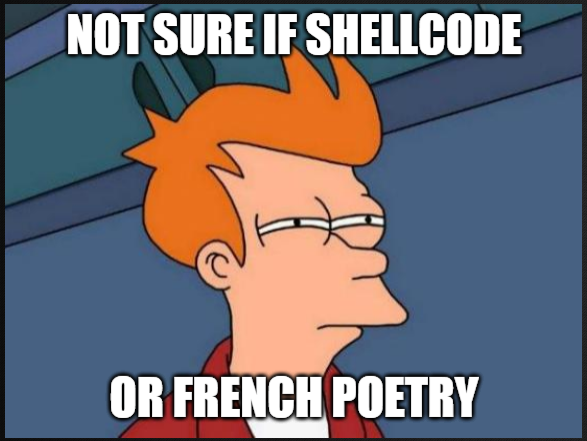
Shellcode 102

Even ignoring the obstacles, take your typical kindergarten shellcode:

```
00000000: 6821 0a00 0068 6f72 6c64 686f 2c20 5768  h!...horldho, Wh
00000010: 4865 6c6c ba0e 0000 0089 e1bb 0100 0000  Hell............
00000020: b804 0000 00cd 80                        .......
```

Some people try to detect and stop shellcodes

Can you believe that?



The textbook shellcode

- Pretends to be user input but...
- ...Has many NOP instructions
- ...And non-printable characters
- ...With `/bin/sh` somewhere
- ...May look like another code

All this is suspicious and easily detectable

Shellcode 103: Fly under the radar

How to pass as human input?

Idea:

- Use only ASCII-printable characters?
- ... only alphanumeric characters?
- ... only English words?
- ... only Shakespeare quotes?

Advantages:

- Less likely to trigger alarm, plausible deniability
- Less likely to be escaped / broken
- Can serve as a pick-up line

Disadvantages:

- Is this even possible?

Shellcode 103: Fly under the radar

How to pass as human input?

Idea:

- Use only ASCII-printable characters?
- ... only alphanumeric characters?
- ... only English words?
- ... only Shakespeare quotes?

Advantages:

- Less likely to trigger alarm, plausible deniability
- Less likely to be escaped / broken
- Can serve as a pick-up line

Disadvantages:

- Is this even possible? YES

ASCII	IA32
'A'	inc %eax
'B'	inc %edx
'C'	inc %ebx
'D'	inc %esp
'E'	inc %ebp
'F'	inc %esi
'G'	inc %edi
'H'	dec %eax
'I'	dec %ecx
'J'	dec %edx
'K'	dec %ebx
'L'	dec %esp
'M'	dec %ebp
'N'	dec %esi
'O'	dec %edi

ASCII	IA32
'P'	push %eax
'Q'	push %ecx
'R'	push %edx
'S'	push %ebx
'T'	push %esp
'U'	push %ebp
'V'	push %esi
'W'	push %edi
'X'	pop %eax
'Y'	pop %ecx
'Z'	pop %edx
'a'	popa
'h'	push imm32
'j'	push imm8

ASCII	IA32
'p'	jo short disp8
'q'	jno short disp8
'r'	jb short disp8
's'	jnb short disp8
't'	je short disp8
'u'	jnz short disp8
'v'	jbe short disp8
'w'	ja short disp8
'x'	js short disp8
'y'	jns short disp8
'z'	jpe short disp8
'8'	cmp r/m8 r8
'9'	cmp r/m32 r32
'0'	xor r/m8 r8

ASCII	IA32
'A'	inc %eax
'B'	inc %edx
'C'	inc %ebx
'D'	inc %esp
'E'	inc %ebp
'F'	inc %esi
'G'	inc %edi
'H'	dec %eax
'I'	dec %ecx
'J'	dec %edx
'K'	dec %ebx
'L'	dec %esp
'M'	dec %ebp
'N'	dec %esi
'O'	dec %edi

ASCII	IA32
'P'	push %eax
'Q'	push %ecx
'R'	push %edx
'S'	push %ebx
'T'	push %esp
'U'	push %ebp
'V'	push %esi
'W'	push %edi
'X'	pop %eax
'Y'	pop %ecx
'Z'	pop %edx
'a'	popa
'h'	push imm32
'j'	push imm8

ASCII	IA32
'p'	jo short disp8
'q'	jno short disp8
'r'	jb short disp8
's'	jnb short disp8
't'	je short disp8
'u'	jnz short disp8
'v'	jbe short disp8
'w'	ja short disp8
'x'	js short disp8
'y'	jns short disp8
'z'	jpe short disp8
'8'	cmp r/m8 r8
'9'	cmp r/m32 r32
'0'	xor r/m8 r8

ASCII	IA32
'A'	inc %eax
'B'	inc %edx
'C'	inc %ebx
'D'	inc %esp
'E'	inc %ebp
'F'	inc %esi
'G'	inc %edi
'H'	dec %eax
'I'	dec %ecx
'J'	dec %edx
'K'	dec %ebx
'L'	dec %esp
'M'	dec %ebp
'N'	dec %esi
'O'	dec %edi

ASCII	IA32
'P'	push %eax
'Q'	push %ecx
'R'	push %edx
'S'	push %ebx
'T'	push %esp
'U'	push %ebp
'V'	push %esi
'W'	push %edi
'X'	pop %eax
'Y'	pop %ecx
'Z'	pop %edx
'a'	popa
'h'	push imm32
'j'	push imm8

ASCII	IA32
'p'	jo short disp8
'q'	jno short disp8
'r'	jb short disp8
's'	jnb short disp8
't'	je short disp8
'u'	jnz short disp8
'v'	jbe short disp8
'w'	ja short disp8
'x'	js short disp8
'y'	jns short disp8
'z'	jpe short disp8
'8'	cmp r/m8 r8
'9'	cmp r/m32 r32
'0'	xor r/m8 r8

ASCII	IA32
'A'	inc %eax
'B'	inc %edx
'C'	inc %ebx
'D'	inc %esp
'E'	inc %ebp
'F'	inc %esi
'G'	inc %edi
'H'	dec %eax
'I'	dec %ecx
'J'	dec %edx
'K'	dec %ebx
'L'	dec %esp
'M'	dec %ebp
'N'	dec %esi
'O'	dec %edi

ASCII	IA32
'P'	push %eax
'Q'	push %ecx
'R'	push %edx
'S'	push %ebx
'T'	push %esp
'U'	push %ebp
'V'	push %esi
'W'	push %edi
'X'	pop %eax
'Y'	pop %ecx
'Z'	pop %edx
'a'	popa
'h'	push imm32
'j'	push imm8

ASCII	IA32
'p'	jo short disp8
'q'	jno short disp8
'r'	jb short disp8
's'	jnb short disp8
't'	je short disp8
'u'	jnz short disp8
'v'	jbe short disp8
'w'	ja short disp8
'x'	js short disp8
'y'	jns short disp8
'z'	jpe short disp8
'8'	cmp r/m8 r8
'9'	cmp r/m32 r32
'0'	xor r/m8 r8

Alphanumeric shellcodes

rix, Phrack 57, 2001 + Basu et al., ICISS, 2014

- The x86 ISA is really nice and smooth when going into alphanumeric:
  - `push`/`pop`/`inc`/`dec` have single letter instructions!
  - `jmp` and `cmp` are available
  - `xor` with many operands are available
- For x86-64: just prefix `0x48` (= `H`) to every instruction (breaks `inc`/`dec` though)
- Follow the tutorial: `https://nets.ec/Alphanumeric_shellcode`

```
jZTYX4UPXk9AHc49149hJG00X5EB00PXHc1149Hcq01q0Hcq41q4Hcy0
Hcq0WZhZUXZX5u7141A0hZGQjX5u49j1A4H3y0XWjXHc9H39XTH394c
```

English shellcode

Mason et al., *English Shellcode*, ACM CCS, 2009

- Generate English compatible subset of IA32...

English shellcode

Mason et al., *English Shellcode*, ACM CCS, 2009

- Generate English compatible subset of IA32...
- ...We have more than alphanumeric: (punctuation, spaces, ...)!

ASCII	IA32
'p.'	jo short $30
'q.'	jno short $30
'r.'	jb short $30
's.'	jnb short $30
't.'	je short $30
'u.'	jnz short $30
'v.'	jbe short $30
'w.'	ja short $30
'x.'	js short $30
'y.'	jns short $30
'z.'	jpe short $30
' ca'	and 61(%ebx), %ah
' An'	and 6E(%ecx), %al
' jo'	and 6F(%edx), %ch

English shellcode

Mason et al., *English Shellcode*, ACM CCS, 2009

- Generate English compatible subset of IA32...
- ...We have more than alphanumeric: (punctuation, spaces, ...)!

ASCII	IA32
'p.'	jo short $30
'q.'	jno short $30
'r.'	jb short $30
's.'	jnb short $30
't.'	je short $30
'u.'	jnz short $30
'v.'	jbe short $30
'w.'	ja short $30
'x.'	js short $30
'y.'	jns short $30
'z.'	jpe short $30
' ca'	and 61(%ebx), %ah
' An'	and 6E(%ecx), %al
' jo'	and 6F(%edx), %ch

English shellcode

Mason et al., *English Shellcode*, ACM CCS, 2009

- Generate English compatible subset of IA32...
- ...We have more than alphanumeric: (punctuation, spaces, ...)!

ASCII	IA32
‘p.’	jo short $30
‘q.’	jno short $30
‘r.’	jb short $30
‘s.’	jnb short $30
‘t.’	je short $30
‘u.’	jnz short $30
‘v.’	jbe short $30
‘w.’	ja short $30
‘x.’	js short $30
‘y.’	jns short $30
‘z.’	jpe short $30
‘ ca’	and 61(%ebx), %ah
‘ An’	and 6E(%ecx), %al
‘ jo’	and 6F(%edx), %ch

English shellcode

Mason et al., *English Shellcode*, ACM CCS, 2009

- Generate English compatible subset of IA32...
- ...We have more than alphanumeric: (punctuation, spaces, ...)!
- Write a decoder with those instructions

ASCII	IA32
‘p.’	jo short $30
‘q.’	jno short $30
‘r.’	jb short $30
‘s.’	jnb short $30
‘t.’	je short $30
‘u.’	jnz short $30
‘v.’	jbe short $30
‘w.’	ja short $30
‘x.’	js short $30
‘y.’	jns short $30
‘z.’	jpe short $30
‘ ca’	and 61(%ebx), %ah
‘ An’	and 6E(%ecx), %al
‘ jo’	and 6F(%edx), %ch

English shellcode

Mason et al., *English Shellcode*, ACM CCS, 2009

- Generate English compatible subset of IA32...
- ...We have more than alphanumeric: (punctuation, spaces, ...)!
- Write a decoder with those instructions
- Fill the gaps with English-looking words

ASCII	IA32
‘p.’	jo short $30
‘q.’	jno short $30
‘r.’	jb short $30
‘s.’	jnb short $30
‘t.’	je short $30
‘u.’	jnz short $30
‘v.’	jbe short $30
‘w.’	ja short $30
‘x.’	js short $30
‘y.’	jns short $30
‘z.’	jpe short $30
‘ ca’	and 61(%ebx), %ah
‘ An’	and 6E(%ecx), %al
‘ jo’	and 6F(%edx), %ch

English shellcode

Mason et al., *English Shellcode*, ACM CCS, 2009

- Generate English compatible subset of IA32...
- ...We have more than alphanumeric: (punctuation, spaces, ...)!
- Write a decoder with those instructions
- Fill the gaps with English-looking words



English shellcode

Mason et al., *English Shellcode*, ACM CCS, 2009

- Generate English compatible subset of IA32...
- ...We have more than alphanumeric: (punctuation, spaces, ...)!
- Write a decoder with those instructions
- Fill the gaps with English-looking words
- Enjoy!



English shellcode: demo

demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo
demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo demo

Alphanumeric on x86 is “fully solved”

- `msfvenom -a x86 -e x86/alpha_mixed`
- In principle we could write a fully functional shellcode from Shakespeare’s works
- However, none of my phones are in x86...

2. Taking RISCs: from laptops to earsides

Transitioning from x86 to ARM

RISC – *reduced* instruction set computer...

Blockers

- No single character instructions anymore
- Few addressing modes – in particular no memory to memory
- Heavy constraints on operands

The above techniques do not work on ARM architectures anymore!



Transitioning from x86 to ARM

RISC – *reduced* instruction set computer...

Blockers

- No single character instructions anymore
- Few addressing modes – in particular no memory to memory
- Heavy constraints on operands

The above techniques do not work on ARM architectures anymore!



Compilation / Emulation / Unpacking

The “compilation” way

Idea:

- Compile assembly code to a constrained instruction set

Pros:

- Easy to compile to one-instruction set computers (e.g Movfuscator)

Cons:

The “compilation” way

Idea:

- Compile assembly code to a constrained instruction set

Pros:

- Easy to compile to one-instruction set computers (e.g Movfuscator)

Cons:

- Does not work when the constraints are mainly on the operands and not on the opcodes
- ...nobody wants to devote their life to writing such a compiler

The “compilation” way

Idea:

- Compile assembly code to a constrained instruction set

Pros:

- Easy to compile to one-instruction set computers (e.g Movfuscator)

Cons:

- Does not work when the constraints are mainly on the operands and not on the opcodes
- ...nobody wants to devote their life to writing such a compiler

(or do they?)

The “emulation” way

Younan et al., Phrack 66, 2009

Idea:

- Write an interpreter for some language

Pros:

- Reusable for different payloads
- It’s feasible
- e.g. Younan’s ARMv7 alphanumeric Brainfuck interpreter

Cons:

The “emulation” way

Younan et al., Phrack 66, 2009

Idea:

- Write an interpreter for some language

Pros:

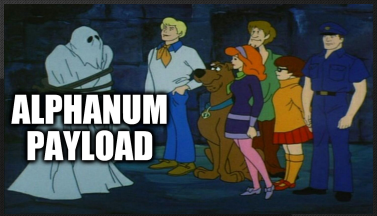
- Reusable for different payloads
- It’s feasible
- e.g. Younan’s ARMv7 alphanumeric Brainfuck interpreter

Cons:

- Interpreted code is toothless...

The “unpacking” way

Barral et al., ISPEC 2016



Idea:

- Encode payload in a constraint-compliant way (e.g. alphanumerically)

The “unpacking” way

Barral et al., ISPEC 2016

Idea:

- Encode payload in a constraint-compliant way (e.g. alphanumerically)
- Identify high-level constraint-compliant constructs (zeroing/increasing registers, ...)



The “unpacking” way

Barral et al., ISPEC 2016

Idea:

- Encode payload in a constraint-compliant way (e.g. alphanumerically)
- Identify high-level constraint-compliant constructs (zeroing/increasing registers, ...)
- Use this to write a minimal unpacker that decodes and execute payload




EVERYONE
X86
RISC-V

Taking it to the next level: RISC-V



Taking it to the next level: RISC-V



Pros

- Simple, clean RISC MIPS-like ISA
- Open Source ISA & Open Hardware*
- Still very WIP

Taking it to the next level: RISC-V



Pros

- Simple, clean RISC MIPS-like ISA
- Open Source ISA & Open Hardware*
- Still very WIP

Cons

- **Makes our job much harder...**

What is alphanumeric 64-bit RISC-V like?

- Some constants `li`, `lui`
- Small increments `addi`
- Forward conditional branches `b*`
- Forward unconditional branches `j`, `jal`
- Arithmetic right shift `sra`
- System registers writes `csr*`
- Misc. floating point operations `fld`, `fmadd`, `fcvt`, ...

What is alphanumeric 64-bit RISC-V like?

- Some constants `li`, `lui`
- Small increments `addi`
- Forward conditional branches `b*`
- Forward unconditional branches `j`, `jal`
- Arithmetic right shift `sra`
- System registers writes `csr*`
- Misc. floating point operations `fld`, `fmadd`, `fcvt`, ...

No loop!
No store / syscall!



Purely alphanumeric RISC-V shellcodes are impossible*

No loop!
No store / syscall!



Not Turing complete; cannot do much...

Purely alphanumeric RISC-V shellcodes are impossible*

No loop!
No store / syscall!



Not Turing complete; cannot do much...



Purely alphanumeric RISC-V shellcodes are impossible*

No loop!
No store / syscall!



Allow ourselves a single other **printable** character



Purely alphanumeric RISC-V shellcodes are impossible*

No loop!
No store / syscall!

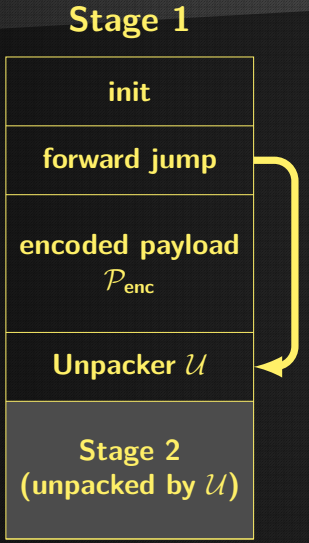


Allow ourselves a single other **printable** character

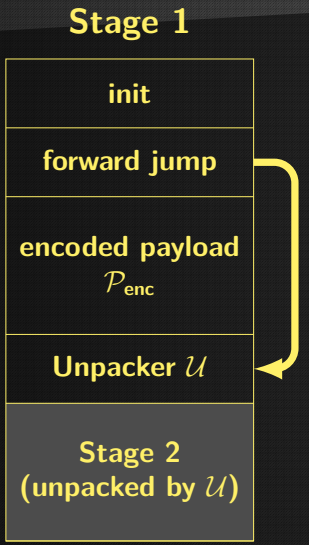


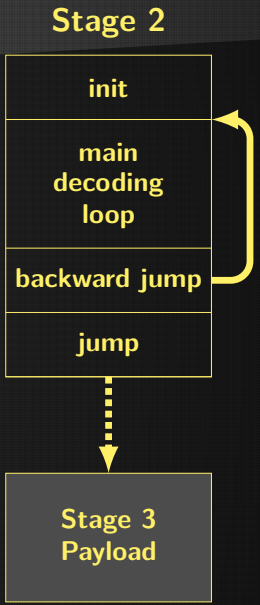
Spoiler Alert: 3 candidates: **#** (hash), **/** (slash) and **´** (tick).

#SudoMakeMeASandwich



#SudoMakeMeASandwich





#SudoMakeMeASandwich: #Demo



3. Think you have seen it all? Getting dirty

http://givemerootaccess/

Pwning you with a single URL

- Changing the available subset to Alphanumeric + `/`.
- No more stores using `sd` for the stage 1.
- Need to find a new memory writing primitive.

http://givemerootaccess/

Pwning you with a single URL

- Changing the available subset to Alphanumeric + `/`.
- No more stores using `sd` for the stage 1.
- Need to find a new memory writing primitive.

Atomic operations

AQ3/ amoor.d t1,s5,(sp)

```
atomic{
t1 := @64[sp]
@64[sp] := @64[sp]|s5
}
```

aA8/ amoand.d a6,s4,(sp)

```
atomic{
a6 := @64[sp]
@64[sp] := @64[sp]&s4
}
```

Alignment is a pain

The **address held** in *rs1* must be **naturally aligned** to the size of the operand [...]. If the address is not naturally aligned, a **misaligned address exception** will be generated.

Alignment is a pain

The **address held** in *rs1* must be **naturally aligned** to the size of the operand [...]. If the address is not naturally aligned, a **misaligned address exception** will be generated.

Good, let's use `addi`.

Alignment is a pain

The **address held** in *rs1* must be **naturally aligned** to the size of the operand [...]. If the address is not naturally aligned, a **misaligned address exception** will be generated.

Good, let's use addi.

```
cat avail | grep addi | sort
```

Alignment is a pain

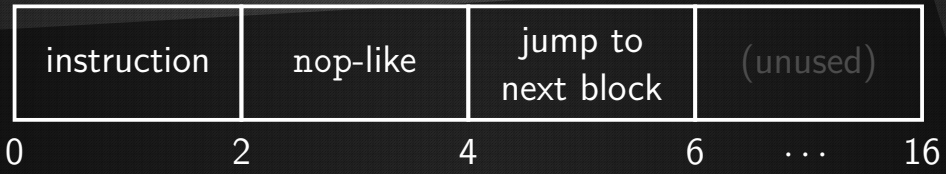
The **address held** in *rs1* must be **naturally aligned** to the size of the operand [...]. If the address is not naturally aligned, a **misaligned address exception** will be generated.

Good, let's use addi.

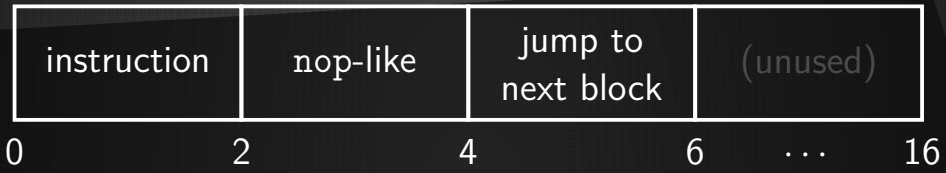
```
cat avail | grep addi | sort
```

```
addi sp, sp, 16
```

Solution: use 16-byte chunks

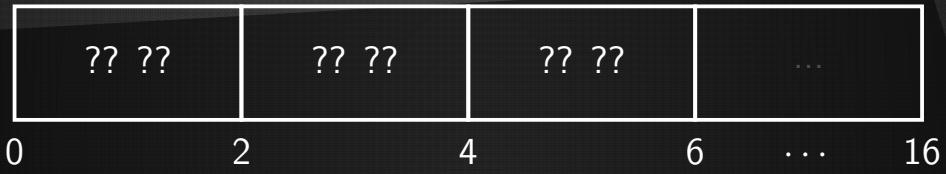


Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

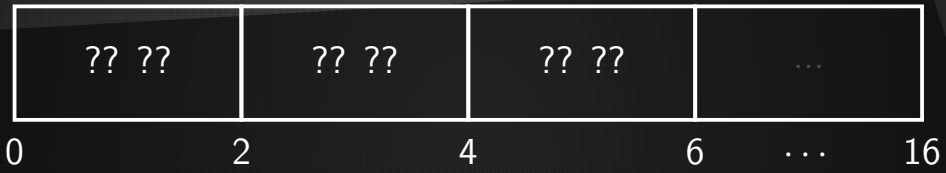
Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

```
tp | 0000 0000 A031 0004
```

Solution: use 16-byte chunks




```
sra             s4,zero,s3
amoand.d.rl     zero,s4,(sp)
amoor.d.aq      a6,tp,(sp)
lui             a0,0x59714
li              s4,12
sra             t1,a0,s4
addiw           t1,t1,-10
amoor.w.aq      t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	0000 0000 0000 0000

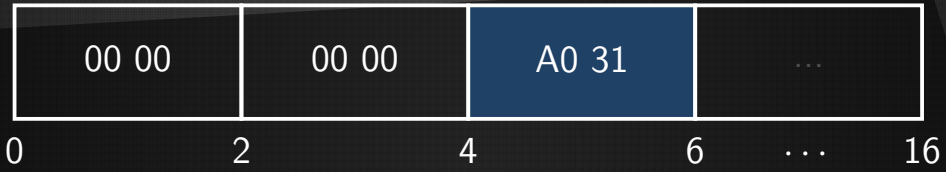
Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	0000 0000 0000 0000

Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	0000 0000 0000 0000

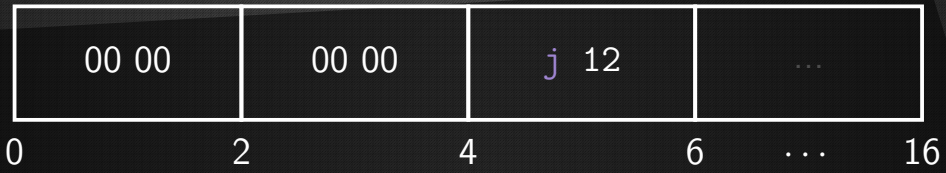
Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	0000 0000 0000 0000

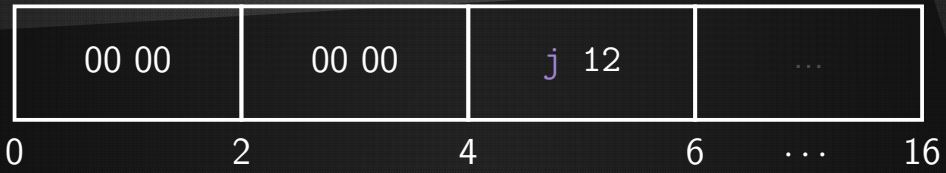
Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	0000 0000 0000 0000
a0	4000 5971 0000 0000

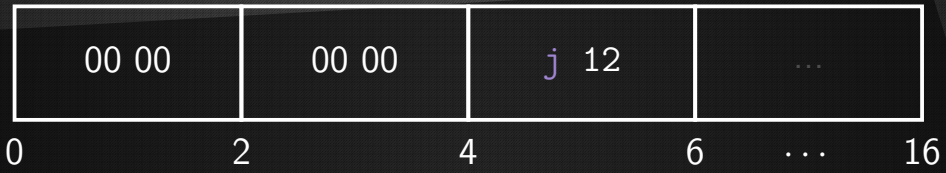
Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	000c 0000 0000 0000
a0	4000 5971 0000 0000

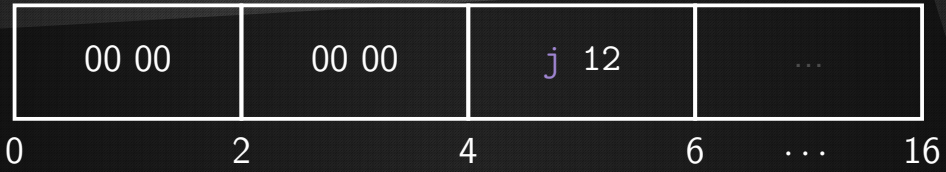
Solution: use 16-byte chunks




```
sra             s4,zero,s3
amoand.d.rl     zero,s4,(sp)
amoor.d.aq      a6,tp,(sp)
lui             a0,0x59714
li              s4,12
sra             t1,a0,s4
addiw           t1,t1,-10
amoor.w.aq      t5,t1,(sp)
```

tp	0000	0000	A031	0004
s4	000c	0000	0000	0000
a0	4000	5971	0000	0000
t1	9714	0005	0000	0000

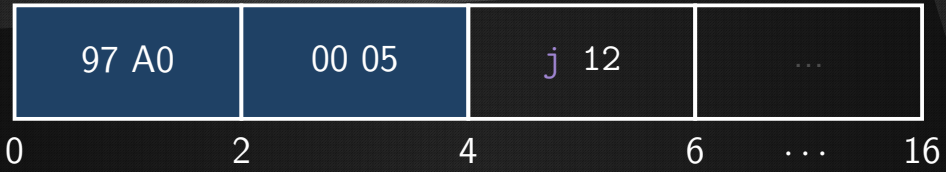
Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	000c 0000 0000 0000
a0	4000 5971 0000 0000
t1	970a 0005 0000 0000

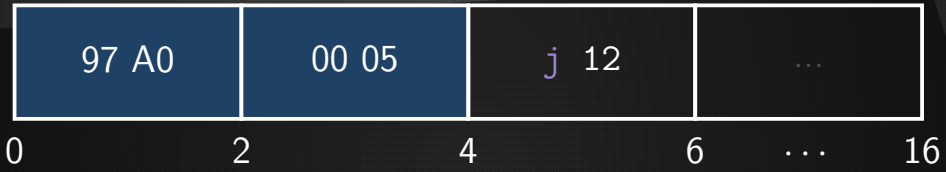
Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	000c 0000 0000 0000
a0	4000 5971 0000 0000
t1	970a 0005 0000 0000

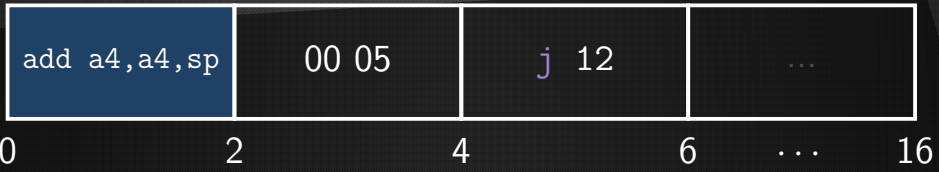
Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	000c 0000 0000 0000
a0	4000 5971 0000 0000
t1	970a 0005 0000 0000

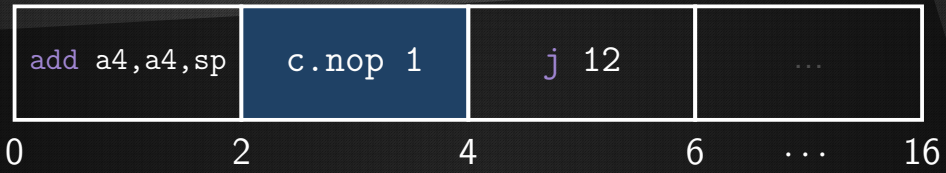
Solution: use 16-byte chunks




```
sra          s4,zero,s3
amoand.d.rl  zero,s4,(sp)
amoor.d.aq   a6,tp,(sp)
lui          a0,0x59714
li           s4,12
sra          t1,a0,s4
addiw        t1,t1,-10
amoor.w.aq   t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	000c 0000 0000 0000
a0	4000 5971 0000 0000
t1	970a 0005 0000 0000

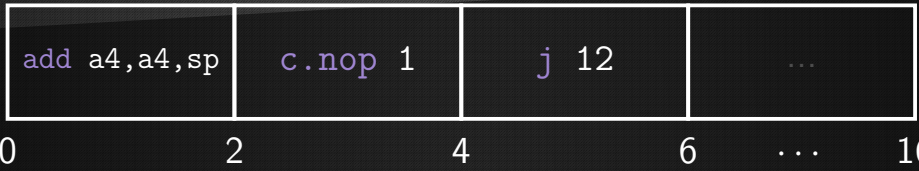
Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	000c 0000 0000 0000
a0	4000 5971 0000 0000
t1	970a 0005 0000 0000

Solution: use 16-byte chunks




```
sra           s4,zero,s3
amoand.d.rl   zero,s4,(sp)
amoor.d.aq    a6,tp,(sp)
lui           a0,0x59714
li            s4,12
sra           t1,a0,s4
addiw         t1,t1,-10
amoor.w.aq    t5,t1,(sp)
```

tp	0000 0000 A031 0004
s4	000c 0000 0000 0000
a0	4000 5971 0000 0000
t1	970a 0005 0000 0000

The majik table

Make bruteforce great again

- Iterate on all instructions sequences in the form `lui`, `sra`, `addiw`
- Make sure that bytes 2 and 3 are nop-like
- Keep the shortest sequence for each 2-byte value
- Write your stage 2 with the available 2-byte values

0f10 0000		a031	…
7d77	0005	a031	…
0a97	0005	a031	…
c145	0001	a031	…
9a05	0001	a031	…
3a85	0005	a031	…
ba86	0005	a031	…
4041	0001	a031	…
a287	0005	a031	…
9183	0001	a031	…
0905	0001	a031	…
3d8c	005e	a031	…
00c3	0001	a031	…
0507	0005	a031	…
f915	0001	a031	…
c1f1	02f2	a031	…
0f10 0000		a031	…
8296	0003	a031	…

fence.i		jmp 12	…
lui a4,0xfffff	c.nop 1	jmp 12	…
add a4,a4,sp	c.nop 1	jmp 12	…
li a1,16	nop	jmp 12	…
slli a1,a1,0x6	nop	jmp 12	…
mv a0,a4	c.nop 1	jmp 12	…
mv a3,a4	c.nop 1	jmp 12	…
lw s0,4(a0)	nop	jmp 12	…
mv a5,s0	c.nop 1	jmp 12	…
srli a5,a5,0x4	nop	jmp 12	…
addi a0,a0,2	nop	jmp 12	…
xor s0,s0,a5	c.slli zero,0x17	jmp 12	…
sw s0,0(a4)	nop	jmp 12	…
addi a4,a4,1	c.nop 1	jmp 12	…
addi a1,a1,-2	nop	jmp 12	…
bnez a1,-256	slli t0,t0,0x1c	jmp 12	…
fence.i		jmp 12	…
jalr a3	n/a	jmp 12	…

fence.i		jmp 12	...
lui a4,0xfffff	c.nop 1	jmp 12	...
add a4,a4,sp	c.nop 1	jmp 12	...
li a1,16	nop	jmp 12	...
slli a1,a1,0x6	nop	jmp 12	...
mv a0,a4	c.nop 1	jmp 12	...
mv a3,a4	c.nop 1	jmp 12	...
lw s0,4(a0)	nop	jmp 12	...
mv a5,s0	c.nop 1	jmp 12	...
srli a5,a5,0x4	nop	jmp 12	...
addi a0,a0,2	nop	jmp 12	...
xor s0,s0,a5	c.slli zero,0x17	jmp 12	...
sw s0,0(a4)	nop	jmp 12	...
addi a4,a4,1	c.nop 1	jmp 12	...
addi a1,a1,-2	nop	jmp 12	...
bnez a1,-256	slli t0,t0,0x1c	jmp 12	...
fence.i		jmp 12	...
jalr a3	n/a	jmp 12	...

0f10 0000		a031	…
7d77	0005	a031	…
0a97	0005	a031	…
c145	0001	a031	…
9a05	0001	a031	…
3a85	0005	a031	…
ba86	0005	a031	…
4041	0001	a031	…
a287	0005	a031	…
9183	0001	a031	…
0905	0001	a031	…
3d8c	005e	a031	…
00c3	0001	a031	…
0507	0005	a031	…
f915	0001	a031	…
c1f1	02f2	a031	…
0f10 0000		a031	…
8296	0003	a031	…

0f10 0000		a031	…
7d77	0005	a031	…
0a97	0005	a031	…
c145	0001	a031	…
9a05	0001	a031	…
3a85	0005	a031	…
ba86	0005	a031	…
4041	0001	a031	…
…			
c1f1	02f2	a031	…
0f10 0000		a031	…
8296	0003	a031	…

The stage 2

0f10 0000		a031	...
7d77	0005	a031	...
0a97	0005	a031	...
c145	0001	a031	...
9a05	0001	a031	...
3a85	0005	a031	...
ba86	0005	a031	...
4041	0001	a031	...
...			
c1f1	02f2	a031	...
0f10 0000		a031	...
8296	0003	a031	...

The stage 2

0f10 0000		jmp 12	...
7d77	0005	jmp 12	...
0a97	0005	jmp 12	...
c145	0001	jmp 12	...
9a05	0001	jmp 12	...
3a85	0005	jmp 12	...
ba86	0005	jmp 12	...
4041	0001	jmp 12	...
...			
c1f1	02f2	jmp 12	...
0f10 0000		jmp 12	...
8296	0003	jmp 12	...

The stage 2

0f10 0000		jmp 12	...
7d77	c.nop 1	jmp 12	...
0a97	c.nop 1	jmp 12	...
c145	nop	jmp 12	...
9a05	nop	jmp 12	...
3a85	c.nop 1	jmp 12	...
ba86	c.nop 1	jmp 12	...
4041	nop	jmp 12	...
	...		
c1f1	slli t0,t0,0x1c	jmp 12	...
0f10 0000		jmp 12	...
8296	n/a	jmp 12	...

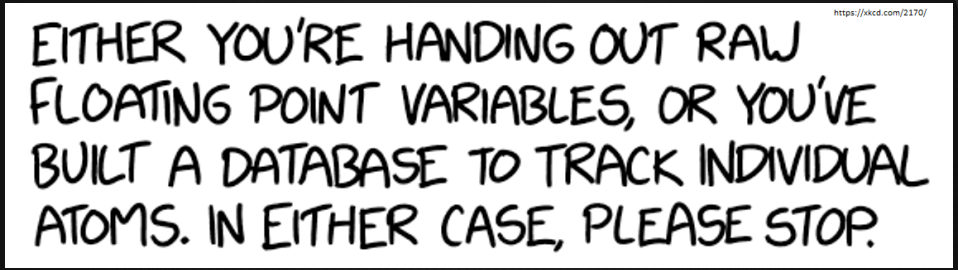
The stage 2

fence.i		jmp 12	...
lui a4,0xfffff	c.nop 1	jmp 12	...
add a4,a4,sp	c.nop 1	jmp 12	...
li a1,16	nop	jmp 12	...
slli a1,a1,0x6	nop	jmp 12	...
mv a0,a4	c.nop 1	jmp 12	...
mv a3,a4	c.nop 1	jmp 12	...
lw s0,4(a0)	nop	jmp 12	...
...			
bnez a1,-256	slli t0,t0,0x1c	jmp 12	...
fence.i		jmp 12	...
jalr a3	n/a	jmp 12	...

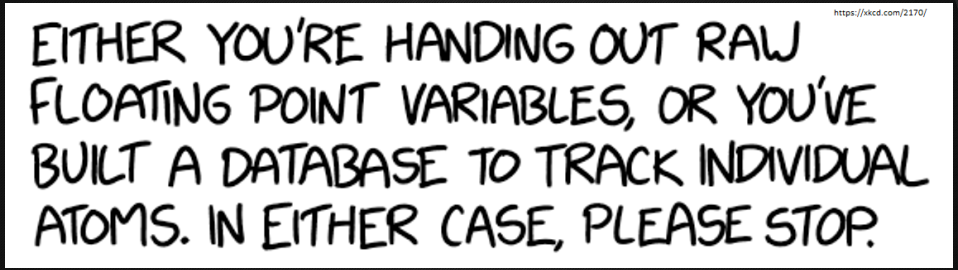
http://givemerootaccess/: /Demo

/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo
/demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo /demo

Fish'n'chips: Floating points for a ticking bomb

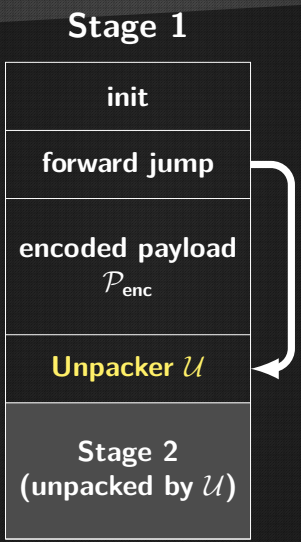


Fish'n'chips: Floating points for a ticking bomb



- (tick) gives us floating point stores!

Fish'n'chips: Adapt our architecture



Fish'n'chips: Adapt our architecture



Fish'n'chips: Adapt our architecture



$$F = \pm mantissa \times 2^{exponent}$$

Fish'n'chips: Adapt our architecture



$$F = \pm mantissa \times 2^{exponent}$$

- Load **some** floating point value from memory into a register (alphanumeric binary representation)
- Do **some** computation
- Register now contains a chunk of our Stage 2, store it to memory

Fish'n'chips: Adapt our architecture



$$F = \pm mantissa \times 2^{exponent}$$

- Load **some** floating point value from memory into a register (alphanumeric binary representation)
- Do **some** computation
- Register now contains a chunk of our Stage 2, store it to memory

Some?

Fish'n'chips: What are our tools?

- Floating point loads `fld`
- Floating point stores `fsw`, `fsd`, `fsq`
- Quad to double conversion `fcvt.d.q`
- Sign manipulation `fabs.q`, `fsgnjx.q`
- Fused multiply-add variants `fmadd`, `fmsub`, `fmnadd`, `fmnsub`

Fish'n'chips: What are our tools?

- Floating point loads `fld`
- Floating point stores `fsw`, `fsd`, `fsq`
- Quad to double conversion `fcvt.d.q`
- Sign manipulation `fabs.q`, `fsgnjx.q`
- Fused multiply-add variants `fmadd`, `fmsub`, `fmnadd`, `fmnsub`

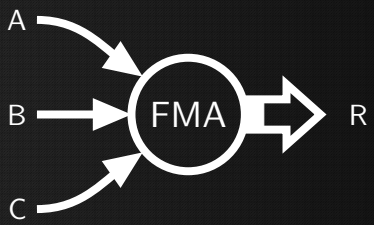
Fused multiply add

$$R := \pm A \times B \pm C$$

`fmsub.d ft6, fs2, ft4, fa0` ft6 := fs2 × ft4 − fa0

Fish'n'chips: Arbitrary binary values

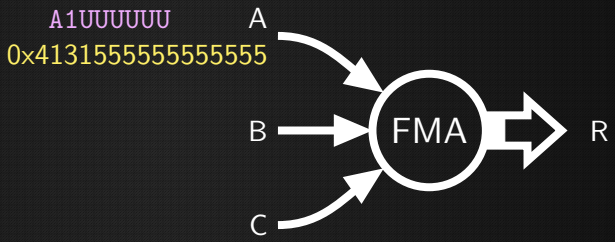
We want to store `0xABCD`



Fish'n'chips: Arbitrary binary values

We want to store `0xABCD`

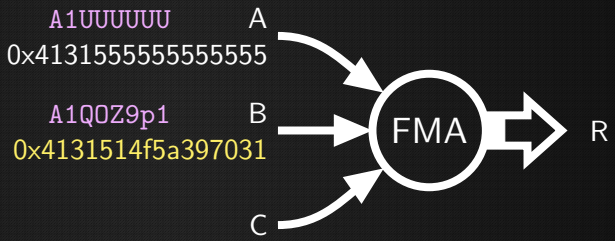
1. Fix *A*
   (for whole Stage 1)



Fish'n'chips: Arbitrary binary values

We want to store `0xABCD`

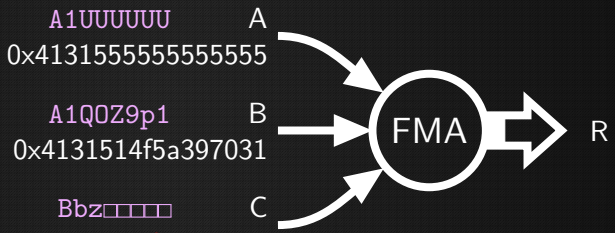
1. Fix A
   (for whole Stage 1)
2. Bruteforce on multiplication
   (Draw randomized B's)



Fish'n'chips: Arbitrary binary values

We want to store `0xABCD`

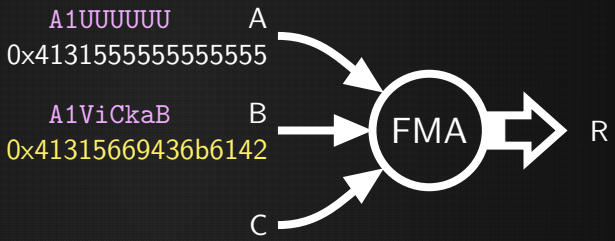
1. Fix A
   (for whole Stage 1)
2. Bruteforce on multiplication
   (Draw randomized B's)
3. Solve on addition
   ($\exists$ an alnum C?)



Fish'n'chips: Arbitrary binary values

We want to store `0xABCD`

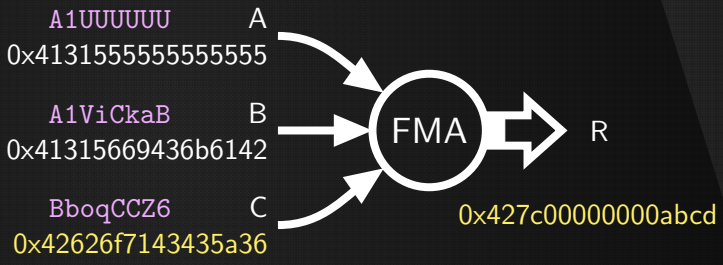
1. Fix A
   (for whole Stage 1)
2. Bruteforce on multiplication
   (Draw randomized B's)
3. Solve on addition
   ($\exists$ an alnum C?)



Fish'n'chips: Arbitrary binary values

We want to store `0xABCD`

1. Fix A
   (for whole Stage 1)
2. Bruteforce on multiplication
   (Draw randomized B's)
3. Solve on addition
   ($\exists$ an alnum C?)



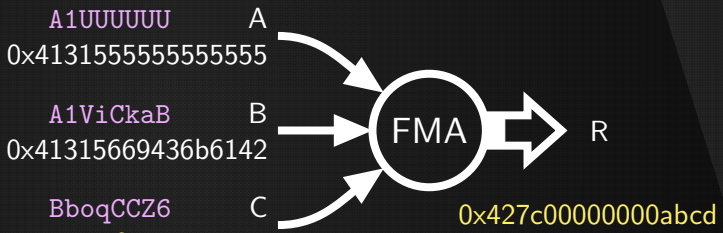
Fish'n'chips: Arbitrary binary values

We want to store `0xABCD`

1. Fix A
   (for whole Stage 1)
2. Bruteforce on multiplication
   (Draw randomized B's)
3. Solve on addition
   ($\exists$ an alnum C?)

Great news

We get all 16 bit values!



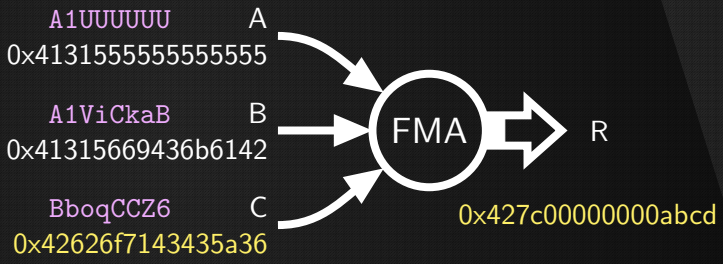
Fish'n'chips: Arbitrary binary values

We want to store `0xABCD`

1. Fix A
   (for whole Stage 1)
2. Bruteforce on multiplication
   (Draw randomized B's)
3. Solve on addition
   ($\exists$ an alnum C?)

~~Great news~~
~~We get all 16 bit values!~~

Greater news
We get all **48** bit values!



Fish'n'chips: 'Demo

'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo
'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo 'demo

666. Conclusion

Conclusion

a. The yoga of writing constrained shellcodes
- Avoid filters, fool IDS and humans, target specific apps
- x86/x86_64 has a mature tooling environment, but it's mostly laptops/servers

b. RISC-V is gaining momentum and we need to keep up
- We show that it is possible to write alphanumeric (with `#`, `/`, or `'`) shellcodes even on very constrained instruction sets
- Automatically. For arbitrary shellcodes.

c. More than tricks and techniques
- The methods can be transported to other architectures, everything is detailed in the paper "*RISC-V: #AlphanumericShellcoding*", all the code is open source

That's all folks

Now get hashing and slashing and ticking for fun and profit!

Code and documentation on:
`https://xn--fda.fr/riscv-alphanumeric-shellcoding/`
Short link: `bit.do/riscv`

Your friendly neighbourhood hackers

`hb.dc27@xn--fda.org` `georges-axel.jaloyan@ens.fr` `remi.geraud@ens.fr`