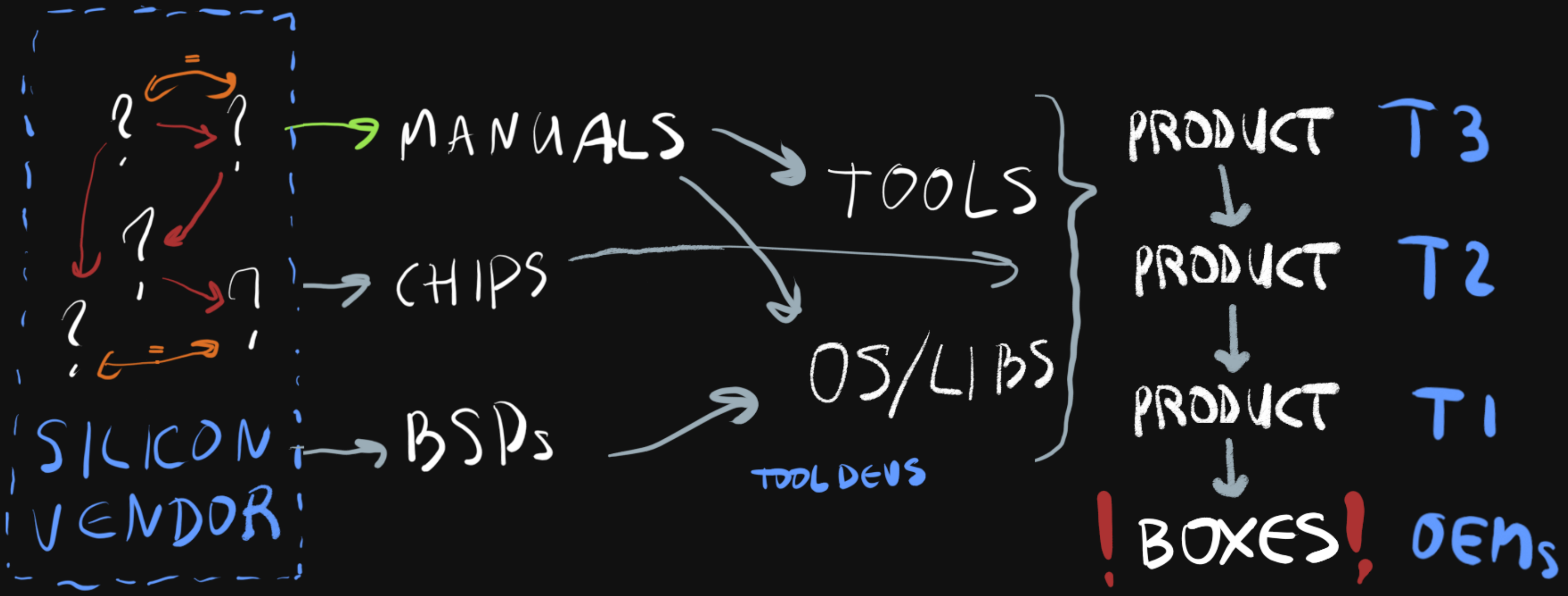
Sum Total of ISA Knowledge

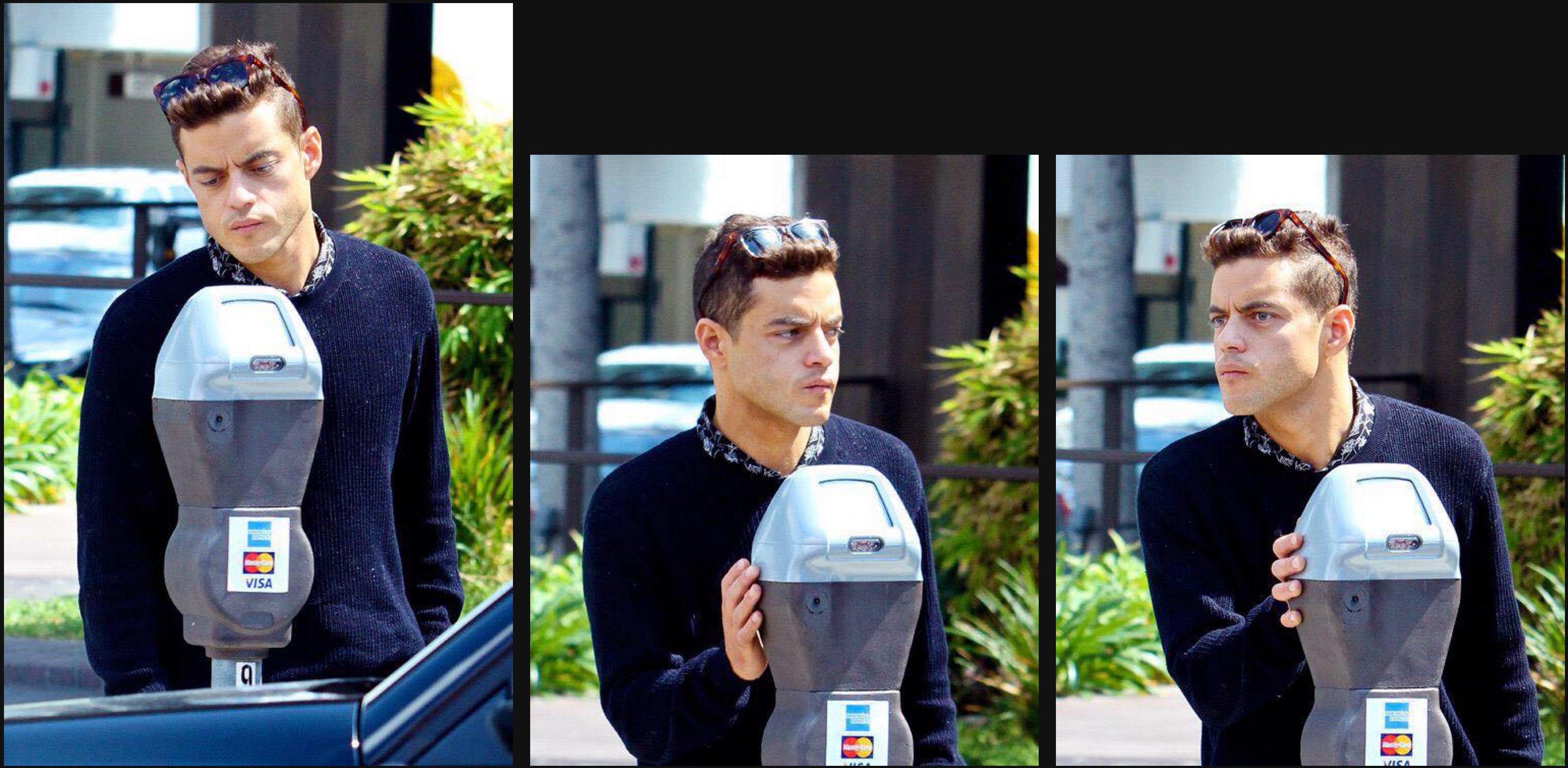
Analyzing Your Static Analysis Tools

@alexkropivny

Unsolicited Firmware Archeology and You



SILICON VENDOR
MANUALS
CHIPS
BSPs
TOOLS
OS/LIBS
TOOL DEVS
PRODUCT
T3
PRODUCT
T2
PRODUCT
T1
!BOXES!
OEMs



"I bet I can hack this"

LINUX ON *		LOW LEVEL
DUMP	✓	DUMP
↓		↓
RE	✓	RE
↓		↓
SHELLCODE	✗	RWX DEBUGGER
↓		↓
SHELL	✗	PATCHES

References

- VMU hackery - full workflow example
- Nexmon - long-term toolchain example

Tools to assist static portion of workflow:

- angr
- Triton (obfuscated interpreters?)
- miasm2 or amoco (pure Python)
- KLEE (if you have source)
- bincat / BAP / Manticore / ...

Manual Static Analysis Automation

Types of Failures

1. False positives discovering more false positives (sev: high)
2. Underapproximations makes you re-visit code (sev: annoying)
3. Script stomped over manually-entered markup (sev: only happens once)

Useful Automation

- Instruction length disassembler
- All control flow effects
- Constant propagation (sometimes)

Useful Automation

- Command/state machine tables (fancy switches)

```
}
[0x32] =
{
    uint8_t _ff = 0xff
    void* p = hook__0x32
}
[0x33] =
{
    uint8_t _ff = 0xff
    void* p = hook__0x33
}
[0x34] =
{
    uint8_t _ff = 0xff
    void* p = hook__0x34
}
[0x35] =
{
    uint8_t _ff = 0xff
    void* p = hook__0x35
}
[0x36] =
```

```
struct state charger__cold_boot =
{
    struct code_ptr name =
    {
        uint8_t _ff = 0xff
        void* at = _"Cold Boot"
    }
    void* handlers[4] =
    {
        [0x0] = charger__cold_boot_0
        [0x1] = charger__cold_boot_1
        [0x2] = state_transition__nop
        [0x3] = state_transition__nop
    }
}
struct state charger__init_charger =
{
```

Lifter Problems: System Code

- Uncommon instruction classes
- Once-per-boot setup features
- Shared memory bus: FIFOs, control flags, DMA

Lifter Problems: Abstractions

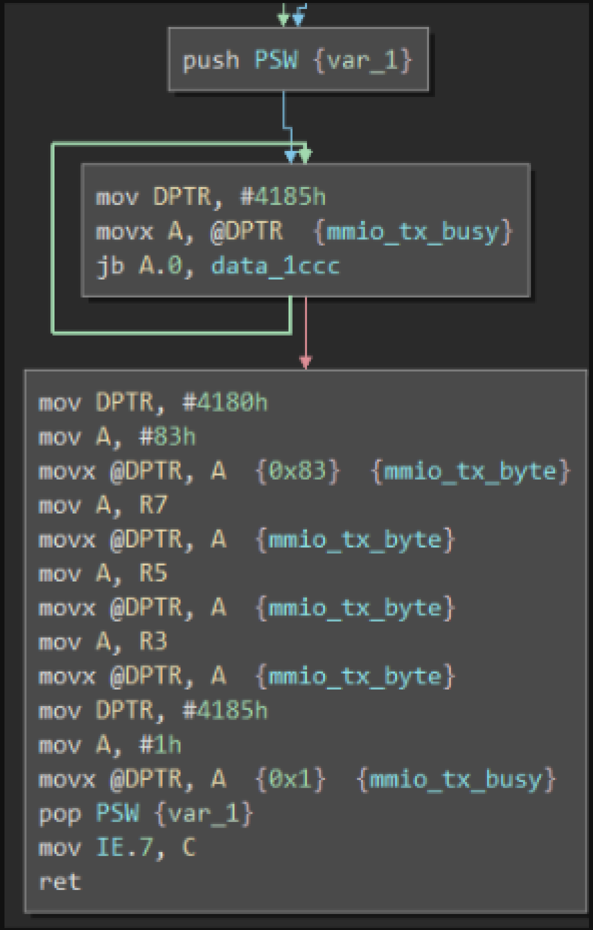
- Flattening memory spaces
- Aliasing with registers (or other memory)
- Inter- vs intra-procedural analysis
- C memory and stack model

Examples

```
    AUXR1 EQU 0A2H

    MOV   DPTR,#SOURCE
    INC   AUXR1
    MOV   DPTR,#DEST
LOOP:
    INC   AUXR1
    MOVX A,@DPTR
    INC   DPTR
    INC   AUXR1
    MOVX @DPTR,A
    INC   DPTR
    JNZ   LOOP
    INC   AUXR1
```

Examples

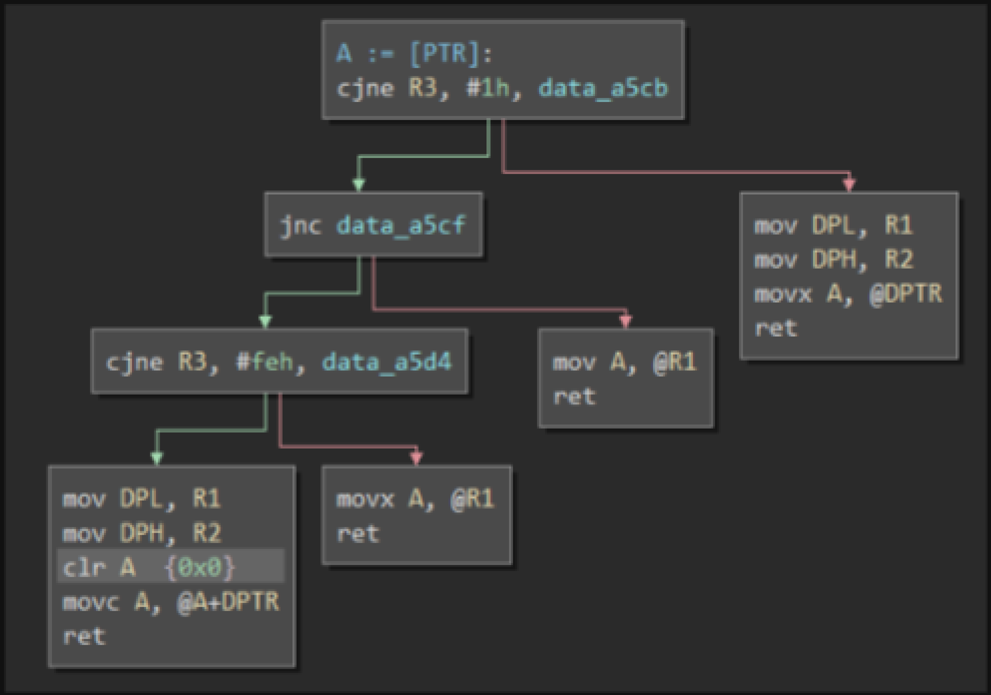

push PSW {var_1}
mov DPTR, #4185h
movx A, @DPTR {mmio_tx_busy}
jb A.0, data_1ccc
mov DPTR, #4180h
mov A, #83h
movx @DPTR, A {0x83} {mmio_tx_byte}
mov A, R7
movx @DPTR, A {mmio_tx_byte}
mov A, R5
movx @DPTR, A {mmio_tx_byte}
mov A, R3
movx @DPTR, A {mmio_tx_byte}
mov DPTR, #4185h
mov A, #1h
movx @DPTR, A {0x1} {mmio_tx_busy}
pop PSW {var_1}
mov IE.7, C
ret

Examples

```
Segments:
r-x  0x00002000-0x00008000  {Code}
r-x  0x00008000-0x00010000  {Code}
r-x  0x00010000-0x00018000  {Code}
r-x  0x00018000-0x00020000  {Code}
r-x  0x00020000-0x00028000  {Code}
rw-  0x80ff000080-0x80ff000100
rw-  0xda1a000000-0xda1a000100
rw-  0xda7a000000-0xda7a010000

Sections:
0x00002000-0x00008000  .code  {Code}
0x00008000-0x00010000  .page0  {Code}
0x00010000-0x00018000  .page1  {Code}
0x00018000-0x00020000  .page2  {Code}
0x00020000-0x00028000  .page3  {Code}
0x80ff000080-0x80ff000100  .special_function_registers
0xda1a000000-0xda1a000020  .register_banks  {Writable d
0xda1a000020-0xda1a000030  .data_bitwise_access  {Writa
0xda1a000030-0xda1a000080  .data  {Writable data}
0xda1a000080-0xda1a000100  .data_indirect_only  {Writab
0xda7a000000-0xda7a010000  .xram  {Writable data}
```

Examples


A := [PTR]:
cjne R3, #1h, data_a5cb
jnc data_a5cf
mov DPL, R1
mov DPH, R2
movx A, @DPTR
ret
cjne R3, #feh, data_a5d4
mov A, @R1
ret
mov DPL, R1
mov DPH, R2
clr A {0x0}
movc A, @A+DPTR
ret
movx A, @R1
ret

Examples

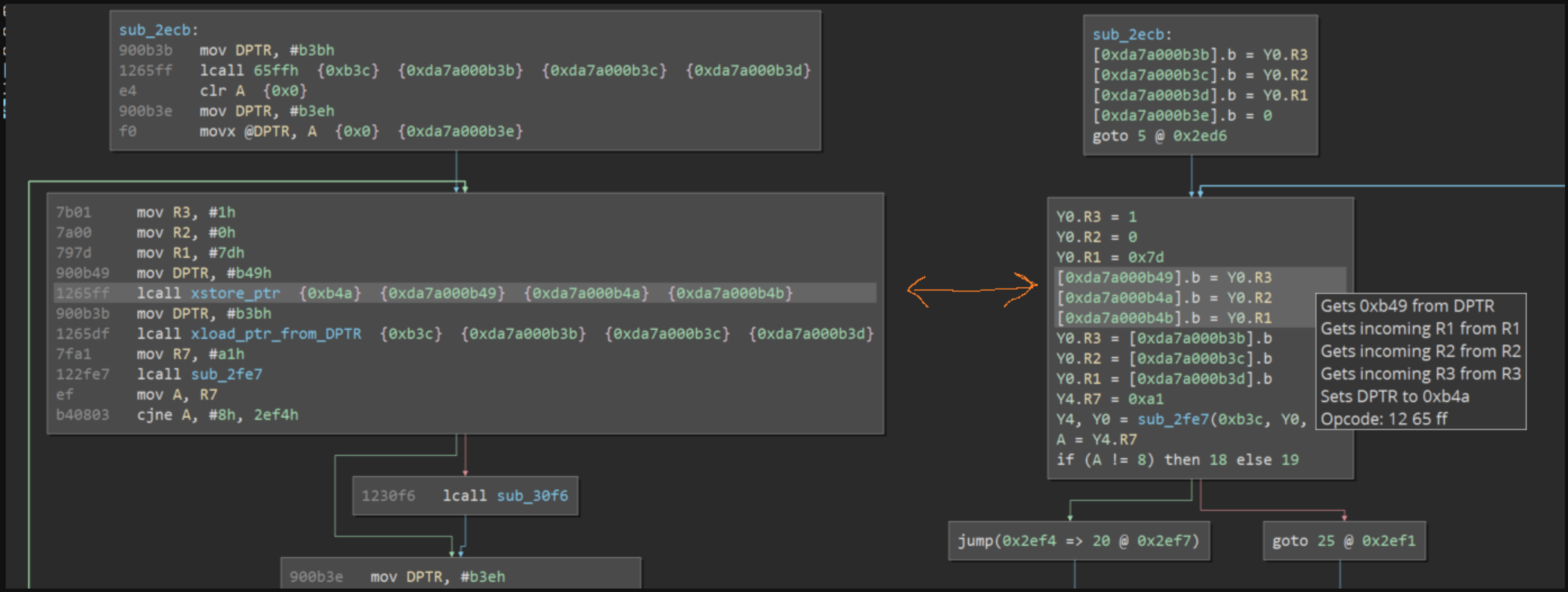
```
x[DPTR] := next word in code:
mov R0, DPL
mov B, DPH
pop DPH
pop DPL
lcall x[R0:B++] := c[DPTR++]
lcall x[R0:B++] := c[DPTR++]
lcall x[R0:B++] := c[DPTR++]
lcall x[R0:B++] := c[DPTR++]
clr A  {0x0}
jmp @A+DPTR
```

```
x[DPTR] := next word in code:
R0 = DPL
B = DPH
DPTR = pop
call(x[R0:B++] := c[DPTR++])
[0xda7a000000 + R0 + (B << 8)].b = [DPTR].b
DPTR = 1 + DPTR
R0 = 1 + R0
call(x[R0:B++] := c[DPTR++])
[0xda7a000000 + R0 + (B << 8
DPTR = 1 + DPTR
R0 = 1 + R0
call(x[R0:B++] := c[DPTR++])
[0xda7a000000 + R0 + (B << 8
DPTR = 1 + DPTR
R0 = 1 + R0
call(x[R0:B++] := c[DPTR++])
[0xda7a000000 + R0 + (B << 8
DPTR = 1 + DPTR
R0 = 1 + R0
A = 0
jump(A + DPTR)
```

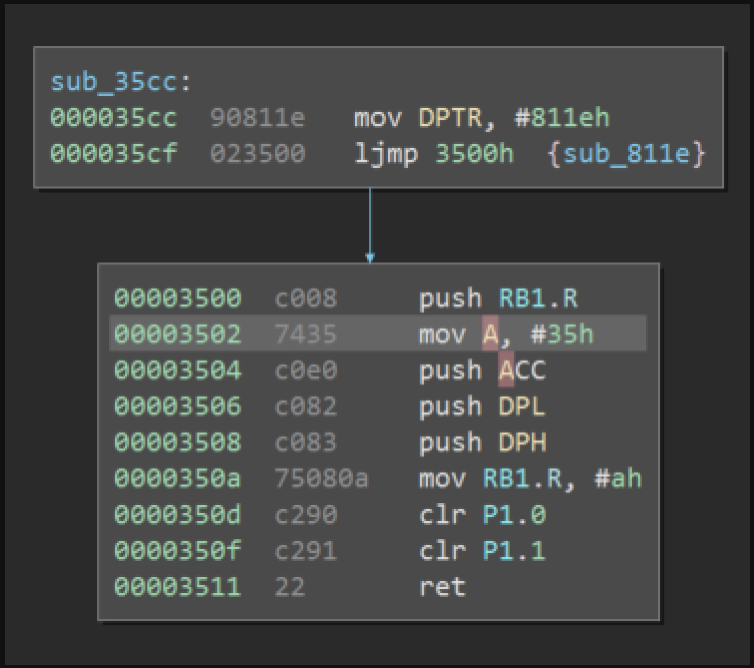
int16_t x[R0:B++] := c[DPTR++]()

```
x[R0:B++] := c[DPTR++]:
clr A  {0x0}
movc A, @A+DPTR
inc DPTR
xch A, DPH
xch A, B
xch A, DPH
xch A, R0
xch A, DPL
xch A, R0
movx @DPTR, A
inc DPTR
xch A, DPH
xch A, B
xch A, DPH
xch A, R0
xch A, DPL
xch A, R0
ret
```

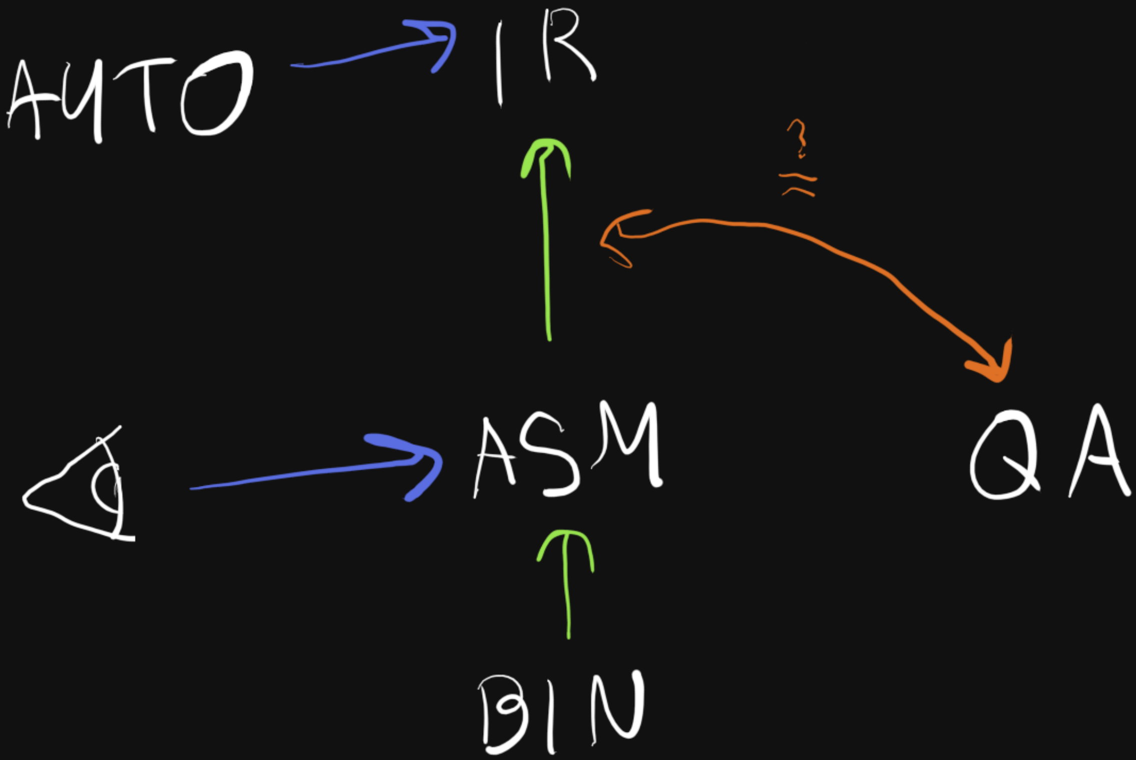
Examples


sub_2ecb:
900b3b mov DPTR, #b3bh
1265ff lcall 65ffh {0xb3c} {0xda7a000b3b} {0xda7a000b3c} {0xda7a000b3d}
e4 clr A {0x0}
900b3e mov DPTR, #b3eh
f0 movx @DPTR, A {0x0} {0xda7a000b3e}
7b01 mov R3, #1h
7a00 mov R2, #0h
797d mov R1, #7dh
900b49 mov DPTR, #b49h
1265ff lcall xstore_ptr {0xb4a} {0xda7a000b49} {0xda7a000b4a} {0xda7a000b4b}
900b3b mov DPTR, #b3bh
1265df lcall xload_ptr_from_DPTR {0xb3c} {0xda7a000b3b} {0xda7a000b3c} {0xda7a000b3d}
7fa1 mov R7, #a1h
122fe7 lcall sub_2fe7
ef mov A, R7
b40803 cjne A, #8h, 2ef4h
1230f6 lcall sub_30f6
900b3e mov DPTR, #b3eh
sub_2ecb:
[0xda7a000b3b].b = Y0.R3
[0xda7a000b3c].b = Y0.R2
[0xda7a000b3d].b = Y0.R1
[0xda7a000b3e].b = 0
goto 5 @ 0x2ed6
Y0.R3 = 1
Y0.R2 = 0
Y0.R1 = 0x7d
[0xda7a000b49].b = Y0.R3
[0xda7a000b4a].b = Y0.R2
[0xda7a000b4b].b = Y0.R1
Y0.R3 = [0xda7a000b3b].b
Y0.R2 = [0xda7a000b3c].b
Y0.R1 = [0xda7a000b3d].b
Y4.R7 = 0xa1
Y4, Y0 = sub_2fe7(0xb3c, Y0,
A = Y4.R7
if (A != 8) then 18 else 19
Gets 0xb49 from DPTR
Gets incoming R1 from R1
Gets incoming R2 from R2
Gets incoming R3 from R3
Sets DPTR to 0xb4a
Opcode: 12 65 ff
jump(0x2ef4 => 20 @ 0x2ef7)
goto 25 @ 0x2ef1

Examples


sub_35cc:
000035cc 90811e mov DPTR, #811eh
000035cf 023500 ljmp 3500h {sub_811e}
00003500 c008 push RB1.R
00003502 7435 mov A, #35h
00003504 c0e0 push ACC
00003506 c082 push DPL
00003508 c083 push DPH
0000350a 75080a mov RB1.R, #ah
0000350d c290 clr P1.0
0000350f c291 clr P1.1
00003511 22 ret

Planned Workflow

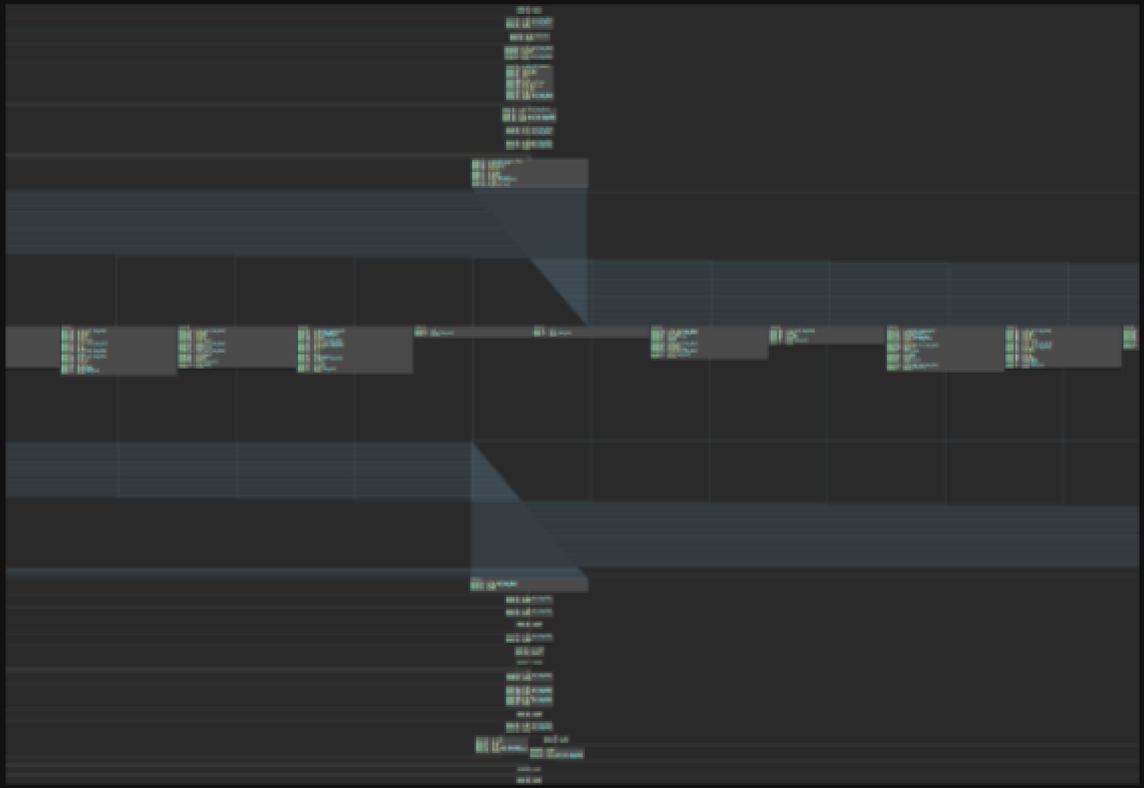


QA by Concrete Execution

Sources of Information

- Emulators!
- Hacker tools

Emulator Architecture

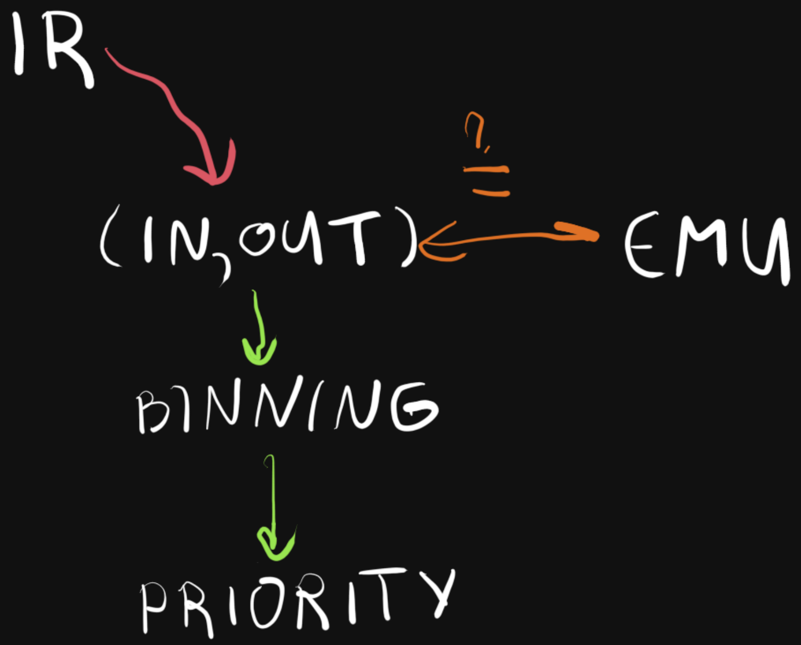


Emulator Architecture

```
{Case 0x22}
bl      pop_pc
ldr     r8, [r7]  {i8051_icount}
b       0x321430
```

```
{Case 0xb2}
ldr     r1, data_321f0c  {opcode_arg_base}
ldr     r0, [r9]  {opcode_mask}
add     r3, r3, #0x2
ldr     r1, [r1]  {opcode_arg_base}
strh    r3, [r4,  #0x2]  {data_1ef26ae}
mov     r3, #0x1
and     r2, r2, r0
strb    r3, [r4,  #0x8]  {0x1}  {data_1ef26b4}
ldrb    r6, [r1, r2]
mov     r0, r6
bl      bit_address_r
and     r1, r0, #0x1
eor     r1, r1, #0x1
mov     r0, r6
bl      bit_address_w
mov     r3, #0
strb    r3, [r4,  #0x8]  {0x0}  {data_1ef26b4}
ldr     r8, [r7]  {i8051_icount}
b       0x321430
```

Fuzzing A vs B



- explore on commonly-occuring instructions
- bin differences on instruction opcodes
- prioritize on registers affected

QA by Symbolic Execution

```
ii = lift.instruction_at(bv, here)       # 'swap' MCS-51 instructi
emu = lift.function(current_function)    # 'swap_a' function on AR

s = ii.solver()
emu.constrain(s)
s.add(z3.And(ii['A'][0] == emu['mem'][0][0x1ef2608],
             ii['A'][-1] != emu['mem'][-1][0x1ef2608]))

print s.check()  # sat
print s.model()[x['A'][0]].sexpr(), ':',
print s.model()[x['A'][-1]].sexpr()
```

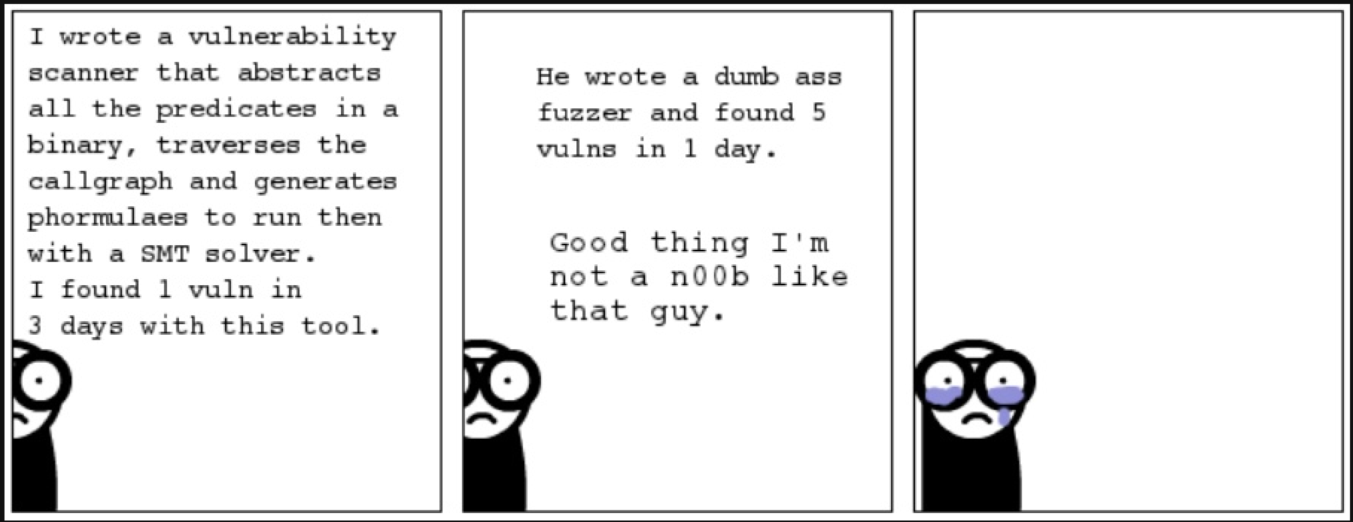
```
[0x1ef2608])); print s.check(); print
[0]].sexpr(), '->', s.model()[x['A'][-
sat
#x01 -> #x10

>>>
```

```
swap_a:
ldr     r3, data_31b034  {data_1ef2600}
ldrb    r2, [r3,  #0x8]  {data_1ef2608}
lsr     r1, r2, #0x4
orr     r2, r1, r2, lsl  #0x4
strb    r2, [r3,  #0x8]  {data_1ef2608}
bx      lr
```

```
Y4 &= Y0:
mov A, R7
anl A, R3
mov R7, A
mov A, R6
anl A, R2
mov R6, A
mov A, R5
anl A, R1
mov R5, A
mov A, R4
anl A, R0
mov R4, A
ret
```

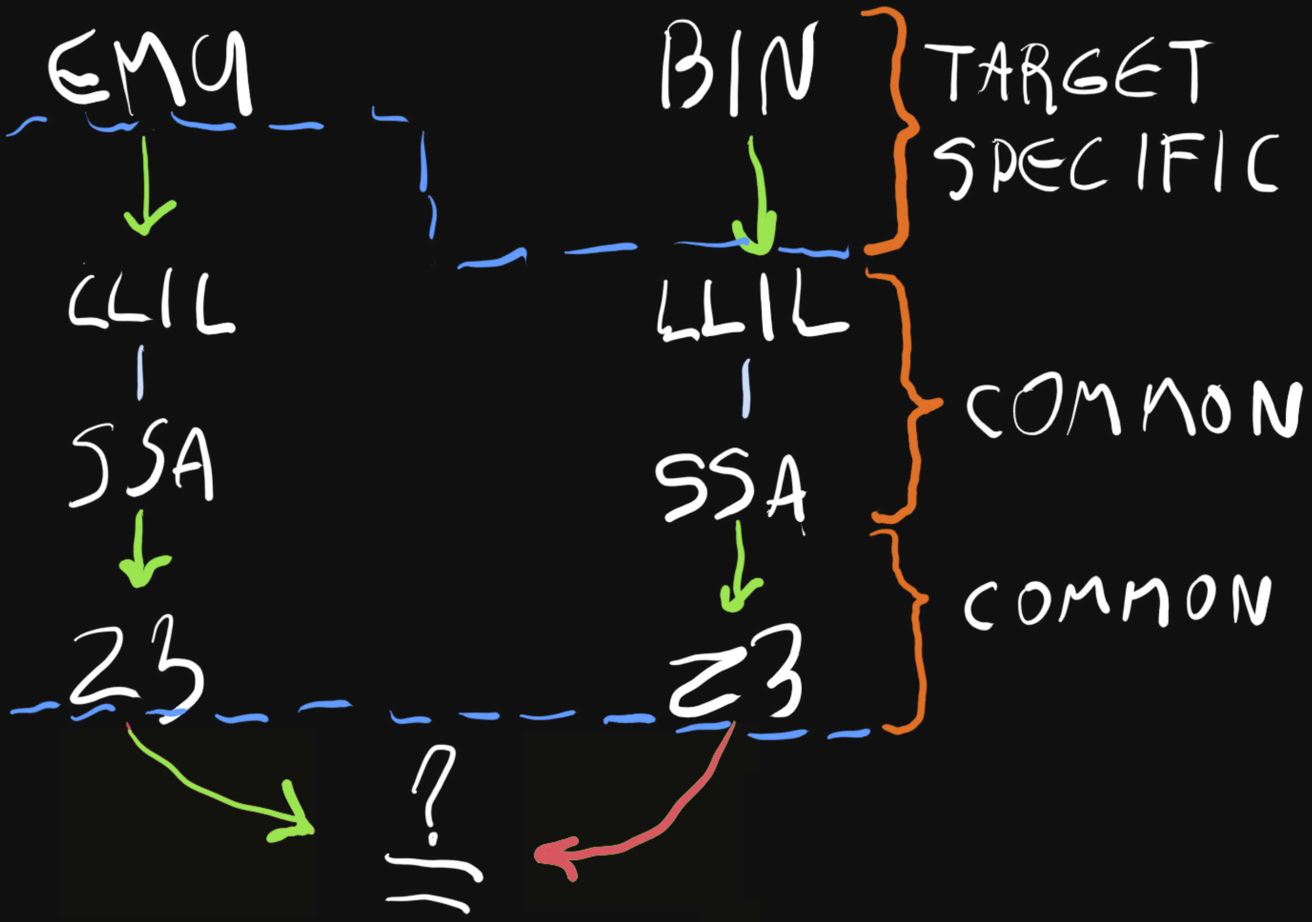
```
x = lift.function(current_function)
summary = x['Y4'][-1] != x['Y4'][0] & x['Y0'][0]
s = x.solver()
s.assert_and_track(summary, 'not-equivalent')
print s.check() # unsat
s.unsat_core()  # [not-equivalent]
```


I wrote a vulnerability scanner that abstracts all the predicates in a binary, traverses the callgraph and generates phormulaes to run then with a SMT solver.
I found 1 vuln in 3 days with this tool.
He wrote a dumb ass fuzzer and found 5 vulns in 1 day.
Good thing I'm not a n00b like that guy.

Program Analysis is a Search Problem

- Fast backtracking vs slow complex search
- Specialized algorithms vs generic solver
- Heuristics compensating for generic solver
- Checking results of $\exists$ search vs $\forall$ search
- Approximating state coverage via path coverage

Workflow and Correctness



References

- MeanDiff - comparison of several major lifters in F#
- Automatic Generation of Peephole Superoptimizers - ambitious academic work
- Fuzzing and Patch Analysis: SAGEly Advice - equivalence checking experiments
- Hi-Fi Tests for Lo-Fi Emulators - emulator comparison (would AFL do better?)

Literature reviews to pull terminology from:

- A Survey of Symbolic Execution Techniques
- A Vocabulary of Program Slicing-Based Techniques
- Mechanizing Proof: Computing, Risk, and Trust for a fun historical perspective

What Went Right & What Went Wrong

1. Approximations:
    - acceptable, but validate major assumptions
2. Partial lifting:
    - acceptable and commonplace
3. Emulator-as-oracle:
    - less partial, needs a map to lifted model
4. Full equivalence checking versus emulator:
    - hampered by 2 and 3, but sometimes works

Example Tools

- i8051 - minimum viable processor module for 8051
- STC - (WIP) attempt at generic lifter analysis tools
- slides (PDF render, reveal.js with notes)