Equational Reasoning of x86 Assembly Code

Kevin Coogan and Saumya Debray
University of Arizona, Tucson, AZ

Assembly Code is Source Code

- **Commercial libraries** often do not come with source code, but there security and correctness influence the overall system.
- **Operating system modules** may include hand-written assembly code to interface to specific hardware.
- **Malware** is often written in assembly to take advantage of security holes.

Assembly Code Issues

x86 assembly presents interesting challenges during analysis

```
1438: mov ecx, [esi]
1439: sub ebp, 0x4
1440: pushf
1441: mov [ebp+0x0], eax
...
2017: shr eax, cl
...
2039: add [ebp+0x4], eax
...
2314: mov al, [esi]
2315: inc esi
2316: movzx eax, al
2317: jmp dword near [eax*4+0x405091]
```

Assembly Code Issues

Many instructions have implicit functionality, such as the subtract operation affecting the value of the eflags register, or the push and pop operations changing the value of the stack pointer.

```
1438: mov ecx, [esi]
1439: sub ebp, 0x4
1440: pushf
1441: mov [ebp+0x0], eax
...
2017: shr eax, cl
...
2039: add [ebp+0x4], eax
...
2314: mov al, [esi]
2315: inc esi
2316: movzx eax, al
2317: jmp dword near [eax*4+0x405091]
```

Assembly Code Issues

User registers in x86 may be referenced by different names. For example, the “cl” register is also the lower 8 bits of the “ecx” register. These dependencies must be handled in analysis.

```
1438: mov ecx, [esi]
1439: sub ebp, 0x4
1440: pushf
1441: mov [ebp+0x0], eax
...
2017: shr eax, cl
...
2039: add [ebp+0x4], eax
...
2314: mov al, [esi]
2315: inc esi
2316: movzx eax, al
2317: jmp dword near [eax*4+0x405091]
```

Assembly Code Issues

- x86 assembly code is typically much larger than source code, in terms of number of instructions.
- When code is intentionally obfuscated, the number of instructions may be arbitrarily large.
- The example from VMProtect on the next slide converted 2 instructions into over 800.

Obfuscated Malware

```
1000: dec ebx
1001: jnz 0x01000
```

$\Longrightarrow$

```
1438: mov ecx, [esi]
1439: sub ebp, 0x4
1440: pushf
1441: mov [ebp+0x0], eax
...
2017: shr eax, cl
...
2039: add [ebp+0x4], eax
...
2314: mov al, [esi]
2315: inc esi
2316: movzx eax, al
2317: jmp dword near [eax*4+0x405091]
```

Solution must handle:

- Implicit functionality
- Dependencies due to accessing partial values
- Increase in overall size of trace

Implicit Functionality

We handle implicit functionality by converting each instruction into an equivalent set of equations

- push eax

- esp := esp – 4
- valueAt(0x1000) := eax

Accessing Partial Values

There are several cases where partial accesses of values can occur. Refer to the paper for full details.

```
valueAt(esi) := 4
ecx := valueAt(esi)

...

eax := eax >> cl
```

Accessing Partial Values

In all cases, we instrument the equations so that a unique and precise definition for each use is available.

```
valueAt(esi) := 4
ecx := valueAt(esi)
cl := Restrict(ecx, 0x000000ff)
...

eax := eax >> cl
```

Simplification

If we want to know something about the calculation of eax, we can look at the values used to calculate it…

```
valueAt(esi) := 4
ecx := valueAt(esi)
cl := Restrict(ecx, 0x000000ff)
...

eax := eax >> cl
```

Simplification

… and substitute the right hand side of the definition at that point.

```
valueAt(esi) := 4
ecx := valueAt(esi)

...

eax := eax >> Restrict(ecx, 0x000000ff)
```

Simplification

We can repeat this process for all uses…

```
valueAt(esi) := 4
ecx := valueAt(esi)

…

eax := eax >> Restrict(ecx, 0x000000ff)
```

Simplification

…over and over…

```
valueAt(esi) := 4

…

eax := eax >> Restrict(valueAt(esi), 0x000000ff)
```

Simplification

```
valueAt(esi) := 4

...

eax := eax >> Restrict(valueAt(esi), 0x000000ff)
```

Simplification

…and simplify as we go…

```
…

eax := eax >> Restrict(4, 0x000000ff)
```

Simplification

Until we have calculated a simplified expression for our variable of interest.

```
...

eax := eax >> 4
```

Uses for equational reasoning

- Our system is very general, but we had a specific application in mind.
- For virtualization-obfuscated malware, we wanted to find conditional control flow statements in a dynamic trace.
- On next slide, we see one example of our results. The original 800+ line example is analyzed, and revealed to contain a conditionally calculated target address.

Handling Indirection

By applying our reasoning to all memory accesses, we identify conditional control flow hidden by multiple layers of indirection.

```
1438: mov ecx, [esi]
1439: sub ebp, 0x4
1440: pushf
1441: mov [ebp+0x0], eax
...
2017: shr eax, cl
...
2039: add [ebp+0x4], eax
...
2314: mov al, [esi]
2315: inc esi
2316: movzx eax, al
2317: jmp dword near [eax*4+0x405091]
```

$\rightarrow$

eax_{2056} := 0x0012ff74 + …
Flag(...atoi(0x003548e1)…
shr 0x04)

esi_{2314} := 0x00405765

eax_{2317} := 0x0000002d

Equational Reasoning

- Handles challenges of x86 assembly code
- Handles indirection in general
- Works well on obfuscated malware

Questions ?