Combating Control Flow Linearization

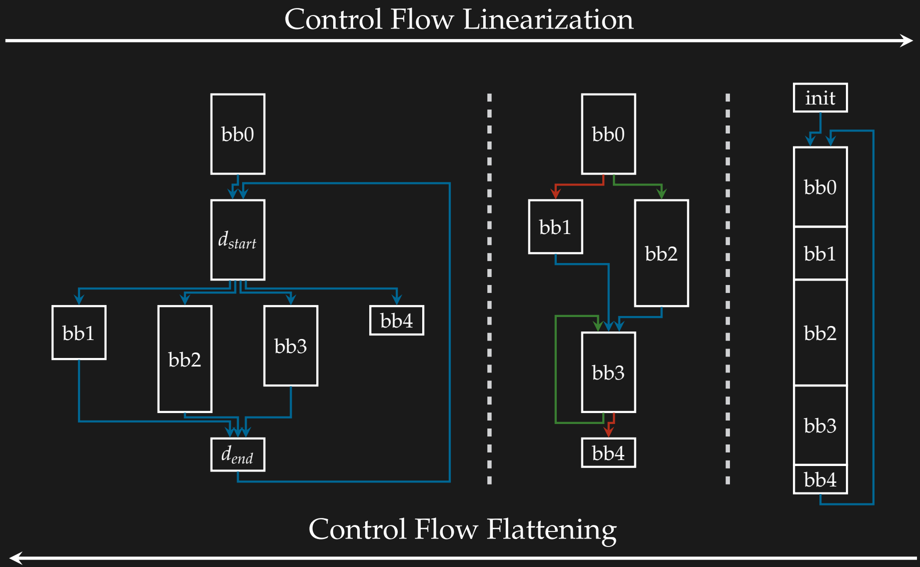
Julian Kirsch, Clemens Jonischkeit, Thomas Kittel,
Apostolis Zarras & Claudia Eckert

Technical University of Munich

May 31, 2017

Introduction

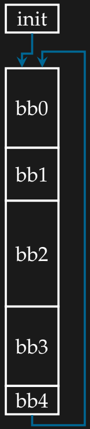
Control Flow Linearization



Background

The M/o/Vfuscator

- `x86(-64)` is **Turing complete**.
- Question: **Smallest subset** of `x86(-64)` instructions that still is Turing complete?



Background

The M/o/Vfuscator

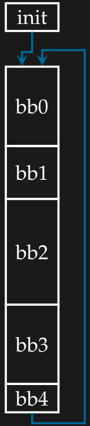
- `x86(-64)` is **Turing complete**.
- Question: **Smallest subset** of `x86(-64)` instructions that still is Turing complete?
- Answer: **1** instruction — `mov` [1]

⇒ The M/o/Vfuscator: C to `x86/mov` compiler [2]

⇒ Control flow Linearization by instruction substitution

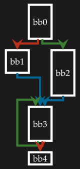
[1] Mov Is Turing Complete. Stephen Dolan. 2013.

[2] The M/o/Vfuscator: Turning the `mov` Instruction into a Soul-Crushing RE Nightmare. Christopher Domas. 2015.



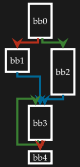
```
mov     eax, dword_83F55B8
mov     edx, 8804857Eh
mov     dword_81F5440, eax
mov     dword_81F5444, edx
mov     eax, 0
mov     ecx, 0
mov     edx, 0
mov     al, byte ptr dword_81F5440
mov     ecx, off_804FA50[eax*4]
mov     dl, byte ptr dword_81F5444
mov     dl, [ecx+edx]
mov     dword_81F5430, edx
mov     al, byte ptr dword_81F5440+1
mov     ecx, off_804FA50[eax*4]
mov     dl, byte ptr dword_81F5444+1
mov     dl, [ecx+edx]
mov     dword_81F5434, edx
⋮
mov     eax, dword_81F5430
mov     edx, dword_81F5434
mov     eax, off_804C4F0[eax*4]
mov     eax, [eax+edx*4]
mov     dword_81F5430, eax
⋮
mov     eax, off_804C4F0[eax*4]
mov     eax, [eax+edx*4]
mov     dword_81F5430, eax
```

Effectiveness against the angr ( ) symbolic execution engine:

Firmalice [3]:	**Clean**	**Obfuscated**
		
# Basic Blocks Executed	37	99,999
Analysis Time (s)	5.1	1704.3
Explored Paths	2	1
Executable Size (bytes)	5400	5,962,776

[3] Firmalice – Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. Yan Shoshitaishvili et al. . 2015

Effectiveness against the angr ( ) symbolic execution engine:

Firmalice [3]:	**Clean**	**Obfuscated**
		
# Basic Blocks Executed	37	**99,999**
Analysis Time (s)	5.1	**1704.3**
Explored Paths	2	1
Executable Size (bytes)	5400	5,962,776

[3] Firmalice – Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. Yan Shoshitaishvili et al. . 2015

Combating Control Flow Linearization

A Bird's Eye Perspective

Technical University of Munich

1. Find **critical data structures** indicating the linearized program's execution state.
2. Infer basic block **labels** using backward taint analysis and constraint solving.
3. Find and identify types of **control flow** changing instructions.
4. **Patch** binary to reconstruct control flow.

Combating Control Flow Linearization

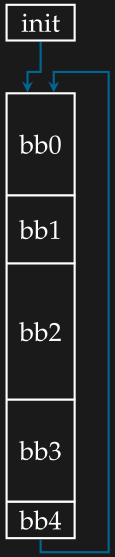
Example: Program with Linearized Control Flow

```c
#define DEFVAR(TYPE, NAME)      TYPE NAME[2] = { 0 }
#define TRUVAL(X) (X[1])
#define ASSIGN(VAR, VAL, CONDVAR, CONDNUM) \
  do { VAR[TRUVAL(CONDVAR) == CONDNUM] = VAL; } while (0)

void nop(void) { return; }

int main(int argc, char **argv) {
    DEFVAR(size_t, state); DEFVAR(size_t, cmp);
    DEFVAR(uint64_t, fac);
    DEFVAR(size_t, i); DEFVAR(size_t, j);
    void (*my_exit[2])(int)                = { nop, exit };
    int (*my_printf[2])(const char *, ...)   = { nop, printf };

    do {
        ASSIGN(i,     1,                      state, 0);
        ASSIGN(j,     atoi(argv[1]),          state, 0);
        ASSIGN(fac,   1,                      state, 0);
        ASSIGN(state, 1,                      state, 0);
        ASSIGN(fac,   TRUVAL(fac) * TRUVAL(i), state, 1);
        ASSIGN(i,     TRUVAL(i) + 1,          state, 1);
        ASSIGN(cmp,   TRUVAL(i) > TRUVAL(j),   state, 1);
        ASSIGN(state, 2,                      cmp,   1);
        my_printf[TRUVAL(state) == 2]("%llu\n", TRUVAL(fac));
        my_exit[TRUVAL(state) == 2](0);
    } while (1);
}
```



Combating Control Flow Linearization

Example: Program with Linearized Control Flow

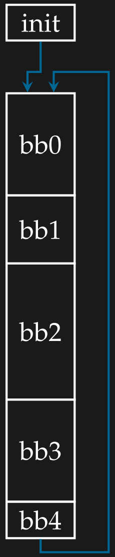
Technical University of Munich

```c
#define DEFVAR(TYPE, NAME)      TYPE NAME[2] = { 0 }
#define TRUVAL(X) (X[1])
#define ASSIGN(VAR, VAL, CONDVAR, CONDNUM) \
  do { VAR[TRUVAL(CONDVAR) == CONDNUM] = VAL; } while (0)

void nop(void) { return; }

int main(int argc, char **argv) {
    DEFVAR(size_t, state); DEFVAR(size_t, cmp);
    DEFVAR(uint64_t, fac);
    DEFVAR(size_t, i); DEFVAR(size_t, j);
    void (*my_exit[2])(int)                = { nop, exit };
    int (*my_printf[2])(const char *, ...)  = { nop, printf };

    do {
        ASSIGN(i,     1,                        state, 0);
        ASSIGN(j,     atoi(argv[1]),            state, 0);
        ASSIGN(fac,   1,                        state, 0);
        ASSIGN(state, 1,                        state, 0);
        ASSIGN(fac,   TRUVAL(fac) * TRUVAL(i),  state, 1);
        ASSIGN(i,     TRUVAL(i) + 1,            state, 1);
        ASSIGN(cmp,   TRUVAL(i) > TRUVAL(j),    state, 1);
        ASSIGN(state, 2,                        cmp,   1);
        my_printf[TRUVAL(state) == 2]("%llu\n", TRUVAL(fac));
        my_exit[TRUVAL(state) == 2](0);
    } while (1);
}
```



Combating Control Flow Linearization

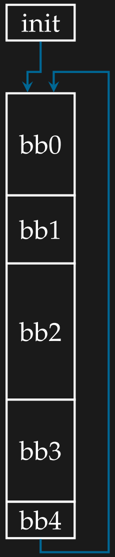
Example: Program with Linearized Control Flow

```c
#define DEFVAR(TYPE, NAME)      TYPE NAME[2] = { 0 }
#define TRUVAL(X) (X[1])
#define ASSIGN(VAR, VAL, CONDVAR, CONDNUM) \
  do { VAR[TRUVAL(CONDVAR) == CONDNUM] = VAL; } while (0)

void nop(void) { return; }

int main(int argc, char **argv) {
    DEFVAR(size_t, state); DEFVAR(size_t, cmp);
    DEFVAR(uint64_t, fac);
    DEFVAR(size_t, i); DEFVAR(size_t, j);
    void (*my_exit[2])(int)                 = { nop, exit };
    int (*my_printf[2])(const char *, ...)   = { nop, printf };

    do {
        ASSIGN(i,     1,                         state, 0);
        ASSIGN(j,     atoi(argv[1]),             state, 0);
        ASSIGN(fac,   1,                         state, 0);
        ASSIGN(state, 1,                         state, 0);
        ASSIGN(fac,   TRUVAL(fac) * TRUVAL(i),   state, 1);
        ASSIGN(i,     TRUVAL(i) + 1,             state, 1);
        ASSIGN(cmp,   TRUVAL(i) > TRUVAL(j),     state, 1);
        ASSIGN(state, 2,                         cmp,   1);
        my_printf[TRUVAL(state) == 2]("%llu\n", TRUVAL(fac));
        my_exit[TRUVAL(state) == 2](0);
    } while (1);
}
```



Combating Control Flow Linearization

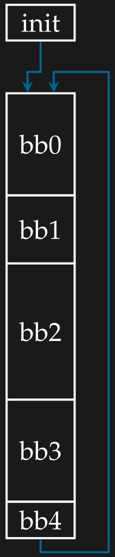
Example: Program with Linearized Control Flow


```c
#define DEFVAR(TYPE, NAME)      TYPE NAME[2] = { 0 }
#define TRUVAL(X) (X[1])
#define ASSIGN(VAR, VAL, CONDVAR, CONDNUM) \
  do { VAR[TRUVAL(CONDVAR) == CONDNUM] = VAL; } while (0)

void nop(void) { return; }

int main(int argc, char **argv) {
    DEFVAR(size_t, state); DEFVAR(size_t, cmp);
    DEFVAR(uint64_t, fac);
    DEFVAR(size_t, i); DEFVAR(size_t, j);
    void (*my_exit[2])(int)                = { nop, exit };
    int (*my_printf[2])(const char *, ...)   = { nop, printf };

    do {
        ASSIGN(i,     1,                        state, 0);
        ASSIGN(j,     atoi(argv[1]),            state, 0);
        ASSIGN(fac,   1,                        state, 0);
        ASSIGN(state, 1,                        state, 0);
        ASSIGN(fac,   TRUVAL(fac) * TRUVAL(i), state, 1);
        ASSIGN(i,     TRUVAL(i) + 1,            state, 1);
        ASSIGN(cmp,   TRUVAL(i) > TRUVAL(j),   state, 1);
        ASSIGN(state, 2,                        cmp,   1);
        my_printf[TRUVAL(state) == 2]("%llu\n", TRUVAL(fac));
        my_exit[TRUVAL(state) == 2](0);
    } while (1);
}
```



- Shortcut: Check `state` **only once** per basic block, store result in variable ON, set on to `false` **after each** basic block:

```
#define DEFVAR(TYPE, NAME)      TYPE NAME[2] = { 0 }
#define TRUVAL(X) (X[1])
#define ASSIGN(VAR, VAL, CVAR, CNUM) do { VAR[TRUVAL(CVAR) == CNUM] = VAL; } while (0)
#define ASSIGN_FAST(VAR, VAL, ON) do { VAR[TRUVAL(ON)] = VAL; } while (0)

int main(int argc, char **argv) {
    DEFVAR(size_t, state); DEFVAR(uint64_t, fac); DEFVAR(uint8_t, on);
    DEFVAR(size_t, i); DEFVAR(size_t, j);

    do {
        ASSIGN     (on,    1,                          state, 0);
        ASSIGN_FAST(i,     1,                          on);
        ASSIGN_FAST(j,     atoi(argv[1]),              on);
        ASSIGN_FAST(fac,   1,                          on);
        ASSIGN_FAST(state, 1,                          on);
        TRUVAL(on) = 0;
        ASSIGN     (on,    1,                          state, 1);
        ASSIGN_FAST(fac,   TRUVAL(fac) * TRUVAL(i), on);
        /* ... */
    } while (1);
}
```

⇒ **Observation**: ON becomes the **most accessed** data structure in linearized code:

```
    do {
        ASSIGN     (on,    1,                            state, 0);
        ASSIGN_FAST(i,     1,                            on);
        ASSIGN_FAST(j,     atoi(argv[1]),                on);
        ASSIGN_FAST(fac,   1,                            on);
        ASSIGN_FAST(state, 1,                            on);
        TRUVAL(on) = 0;
        ASSIGN     (on,    1,                            state, 1);
        ASSIGN_FAST(fac,   TRUVAL(fac) * TRUVAL(i),      on);
        /* ... */
    } while (1);
}
```

⇒ ON **trivial to detect** by linear sweep disassembly or **frequency analysis** of memory access patterns

Combating Control Flow Linearization

A Bird's Eye Perspective

Technical University of Munich

1. Find **critical data structures** indicating the linearized program's execution state. ✓
2. Infer basic block **labels** using backward taint analysis and constraint solving.
3. Find and identify types of **control flow** changing instructions.
4. **Patch** binary to reconstruct control flow.

- **Observation**: Any given basic block writes `1` to ON if the `state` variable **equals the respective block's LABEL**:

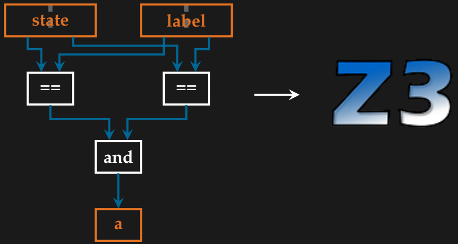
```
/* ... */
ASSIGN(on,    1,                          state, 0);
/* ... */
ASSIGN(on,    1,                          state, 1);
/* ... */
```

⇒ Reconstruct predicate used to access ON (i.e. `a = state == 0` in `on[a] = 1`) using backwards **taint analysis**.

⇒ Build up **syntax tree** of arithmetic / logic operations applied to the `state` variable.

⇒ Constrain formula to be `1`, and solve system using an **SMT solver** (z3).

⇒ Result: List of basic block **labels** + location of **state**



Combating Control Flow Linearization

A Bird's Eye Perspective

1. Find **critical data structures** indicating the linearized program's execution state. ✓
2. Infer basic block **labels** using backward taint analysis and constraint solving. ✓
3. Find and identify types of **control flow** changing instructions.
4. **Patch** binary to reconstruct control flow.

Combating Control Flow Linearization

Identifying Types of Control Flow Changing Instructions

- Employ **backwards taint analysis** on positions writing to `state`.
- **Four** different cases for predicate and value written:

Jump / Call:

```
*stack_ptr-- = label_0;
state[TRUEVAL(on)] = label_1;
```

Conditional Jump:

```
state[condition] = label;
```

Indirect jump:

```
x = ...
state[TRUVAL(on)] = x;
```

Return:

```
x = *stack_ptr++;
state[TRUVAL(on)] = x;
```

⇒ **Location** of `stack_ptr`

⇒ (Un-)conditional **edges** of the control flow graph

Combating Control Flow Linearization

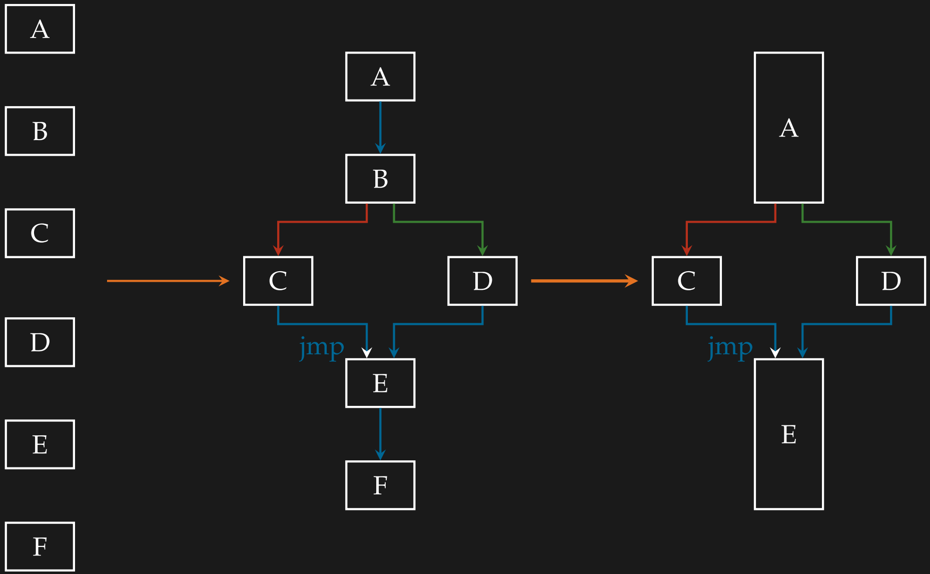
A Bird's Eye Perspective

Technical University of Munich

1. Find **critical data structures** indicating the linearized program's execution state. ✓
2. Infer basic block **labels** using backward taint analysis and constraint solving. ✓
3. Find and identify types of **control flow** changing instructions. ✓
4. **Patch** binary to reconstruct control flow.

Combating Control Flow Linearization

Patching the Binary



Combating Control Flow Linearization

Patching the Binary

Technical University of Munich

1. Find **critical data structures** indicating the linearized program's execution state. ✓
2. Infer basic block **labels** using backward taint analysis and constraint solving. ✓
3. Find and identify types of **control flow** changing instructions. ✓
4. **Patch** binary to reconstruct control flow. ✓

CFL **overhead** in terms of run-time (seconds) and code size (bytes):

	Primes		**Factorial**		**SHA-256**	
	Non-Lin.	Lin.	Non-Lin.	Lin.	Non-Lin.	Lin.
Non-Sub.	0.88 s	5.03 s	< 0.01 s	< 0.01 s	0.02 s	0.4 s
	240 B	928 B	1884 B	1936 B	5672 B	8564 B
Sub.	62.82 s	289.47 s	< 0.01 s	< 0.01 s	8.09 s	60.57 s
	16,957 B	16,957 B	10,684 B	10,684 B	213,740 B	213,740 B

Deobfuscation times of the implementation of our algorithm:

Primes	**Factorial**	**SHA-256**	**AES**
0.47 s	0.213 s	0.824 s	3.68 s

Evaluation

Control Flow Linearization: Cost and Gain

Execution time of the **angr** () symbolic execution engine to **detect** a backdoor in an example executable:

	Clean	Obfuscated	Deobfuscated
# Basic Blocks Executed	37	99,999	87
Execution Time (s)	5.1	1704.3	17.9
Explored Paths	2	1	3
Executable Size (bytes)	5400	5,962,776	5,962,776

Conclusion

Technical University of Munich

- Control Flow Linearization **unsuited** for obfuscating **real time** applications
- Major **challenge** for state of the art **symbolic execution** engines
- Presence of **control data structures** makes deobfuscation easy

Contact

Technical University of Munich

- `mail@kirschju.re`
  `PGP: F949 CFBD 140A 6DD0 71E9 0B8C DC24 396B 6D45 1038`

- **Sources** available – documentation pending :-)
  → Source code:
  `https://github.com/kirschju/demovfuscator`
  → Project website:
  `https://kirschju.re/demov`
  → Combating Control Flow Linearization:
  `https://kirschju.re/static/cfl.pdf`
  → Slides:
  `https://kirschju.re/static/ifip.pdf`

Thanks!