Binary Rewriters

Charles Averill

Dallas Hackers' Association

November 2025

Binary Rewriters

Binary rewriters are very cool and have numerous applications in security and software development.

As you might expect, a rewriter takes in a binary and performs some transformation on it, resulting in a new binary with (potentially) different behavior.

Generally, the steps are:

1. Disassemble the binary
2. Transform the assembly
3. Assemble into a new binary

Why? What if I want to ensure that the Office binary I downloaded meets my own personal security specifications?

Binary Rewriters

Binary rewriters are very cool and have numerous applications in security and software development.
As you might expect, a rewriter takes in a binary and performs some transformation on it, resulting in a new binary with (potentially) different behavior.
Generally, the steps are:

1. Disassemble the binary
2. Transform the assembly
3. Assemble into a new binary

Why? What if I want to ensure that the Office binary I downloaded meets my own personal security specifications?

Binary Rewriters

Binary rewriters are very cool and have numerous applications in security and software development.
As you might expect, a rewriter takes in a binary and performs some transformation on it, resulting in a new binary with (potentially) different behavior.
Generally, the steps are:

1. Disassemble the binary
2. Transform the assembly
3. Assemble into a new binary

Why? What if I want to ensure that the Office binary I downloaded meets my own personal security specifications?

Binary Rewriters

Binary rewriters are very cool and have numerous applications in security and software development.
As you might expect, a rewriter takes in a binary and performs some transformation on it, resulting in a new binary with (potentially) different behavior.
Generally, the steps are:

1. Disassemble the binary
2. Transform the assembly
3. Assemble into a new binary

Why? What if I want to ensure that the Office binary I downloaded meets my own personal security specifications?

Binary Rewriters

Binary rewriters are very cool and have numerous applications in security and software development.
As you might expect, a rewriter takes in a binary and performs some transformation on it, resulting in a new binary with (potentially) different behavior.
Generally, the steps are:

1. Disassemble the binary
2. Transform the assembly
3. Assemble into a new binary

Why? What if I want to ensure that the Office binary I downloaded meets my own personal security specifications?

Relocation

Transforming machine code is not easy!
Rewriting a source program is easy, we do it all the time:

```
int f(x);
int g(x) {
    return 3 + f(x);
}

// After rewrite

int f_new(x) {
    printf("Calling␣f(%d)\n", x);
    return f(x);
}

int g(x) {
    return 3 + f_new(x);
}
```

Relocation

In machine code, jump addresses are hard-coded into each instruction:

```
0484: add t0, t0, 5
0488: bne t0, 0490
048C: ret
0490: ...
```

So if we naïvely insert or remove any instructions...

```
0484: call printf
0488: add t0, t0, 5
048C: bne t0, 0490
0490: ret
0494: ...
```

The behavior of our code is completely different!

Relocation

```
new_bin = []
addr_map = []
for i, instr in enumerate(orig_binary):
    if instr is `Jmp`:
        new_bin.append(Print("Jumping to " + instr.target))
        new_bin.append(instr)
        addr_map.append(i)
        i += 2
    else:
        new_bin.append(instr)
        addr_map.append(i++)

for i, instr in enumerate(new_bin):
    if instr is `Jmp`:
        new_bin[i] = Jmp(addr_map[instr.target])
```

Control-Flow Integrity

Control-Flow Integrity is the modern solution to ROP attacks. Basically, build a whitelist of control-flow graph edges, and before any jump, check whether the jump is in the whitelist.

```
valid_targets = build_control_flow_graph(binary)

for instr in binary:
    if instr is `Jmp`:
        insert_before(instr, check_target(instr.target))

function check_target(tgt):
    if tgt not in valid_targets[current_function]:
        raise_security_exception()
```

Profiling

I do this for my research on timing! (but at the source level ☹) Inject reads from hardware timing registers like TSC before and after targeted operations to determine their execution time.

See my profiling instrumentation (this is the same as what the binary rewriter would do, but more convenient. sorry!)

```
for instr in binary:
    insert_before(instr, x <- read(tsc))
    insert_after(instr, x <- read(tsc))
    insert_after(instr, Print(x))
```

Profiling

I do this for my research on timing! (but at the source level ☹) Inject reads from hardware timing registers like TSC before and after targeted operations to determine their execution time.

See my profiling instrumentation (this is the same as what the binary rewriter would do, but more convenient. sorry!)

```
for instr in binary:
    insert_before(instr, x <- read(tsc))
    insert_after(instr, x <- read(tsc))
    insert_after(instr, Print(x))
```

Inlined Reference Monitors

IRMs are chunks of coded injected into binaries to enforce arbitrary security policies at arbitrary points in execution.
For example, insert the following before any file write operation:

```
if not write_perms(file):
    deny_action("permission denied", context)
    goto recovery_path
```

- Sandbox security-critical operations
- Memory safety enforcement
- Information flow control
- Intrusion detection