Dynamic Taint Analysis versus Obfuscated Self-Checking

Sebastian Banescu **Samuel Valenzuela** Marius Guggenmos
Mohsen Ahmadvand Alexander Pretschner

Annual Computer Security Applications Conference (ACSAC) - December 6-10, 2021

Introduction

- **Context:** Self-checking used as software protection against tampering
- **Problem:** Automated attack using taint-analysis on self-checking exists
- **Idea:** Use code obfuscation to hide self-checking
- **RQ:** Can obfuscation protect against dynamic taint-analysis attacks on self-checking?

- **Our work:**
  - Evaluate dynamic taint-analysis attack on popular obfuscations
  - Improve most resilient protection

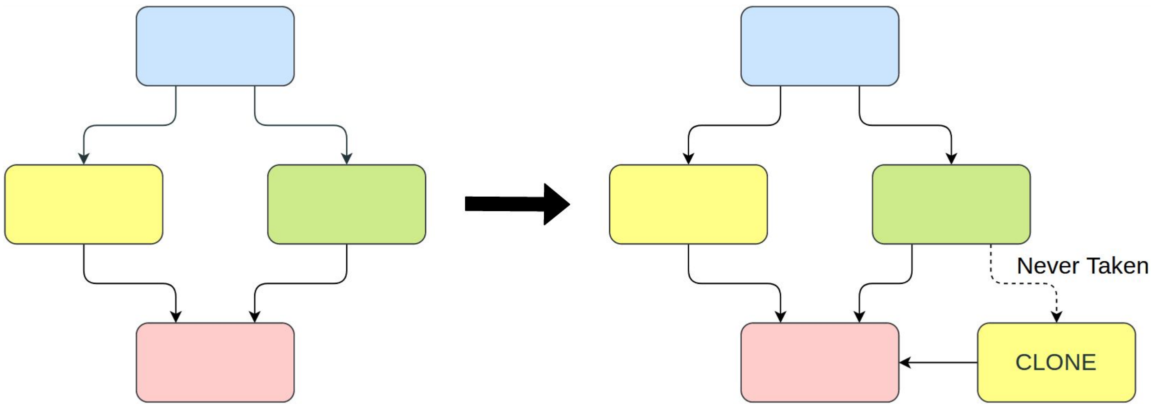
Self-Checksumming: Software Tamper Protection

- Detects and responds to tampering
- Inserts code guards in program
- Example:

```
...
int actual = compute_checksum(...);
if(actual != expected) {
  response_mechanism();
}
...
```

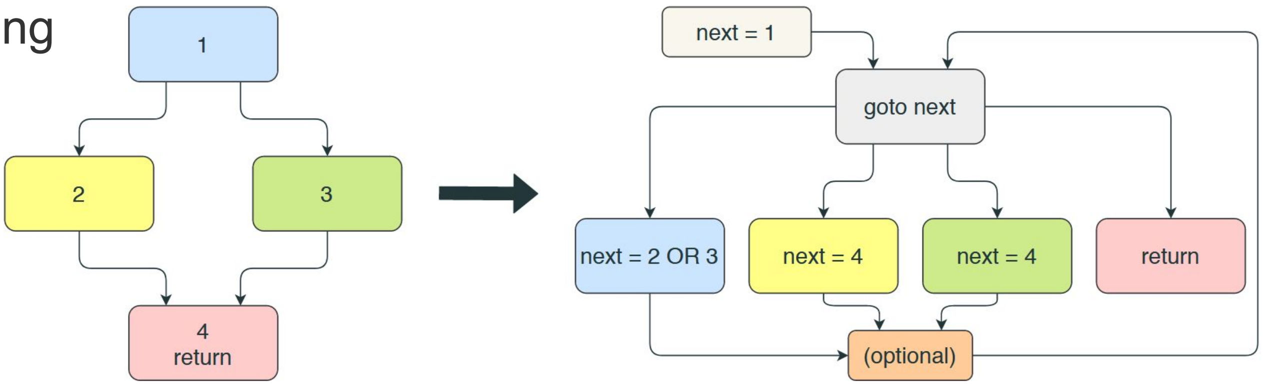
Obfuscations Used To Hide Self-Checking (1)

- Instruction Substitution   a = b | c   ->   a = (b & c) | (b ^ c)
- Control Flow Indirection   jmp 0x123   ->   a = 0x123
  jmp a
- Opaque Predicates



Obfuscations Used To Hide Self-Checking (2)

- Control Flow Flattening



- Virtualization

Replaces instructions with emulator

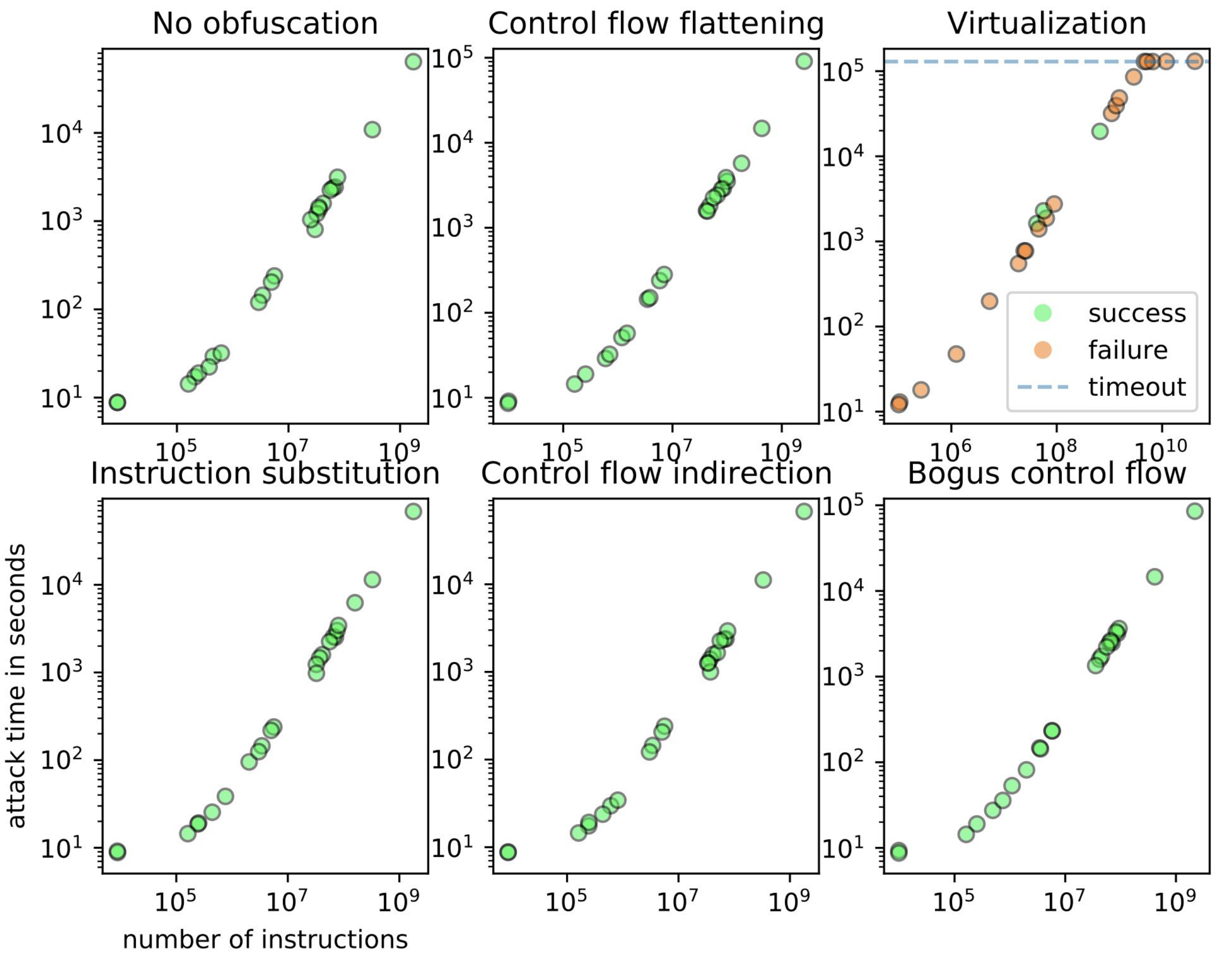
Attacking Self-Checksumming on Machine Code

Input: executable binary (+ command line arguments to be applied)

- **Step 1:** generate execution trace of binary
- **Step 2:** taint program's executable memory
- **Step 3:** perform dynamic taint analysis on emulated instructions
- **Step 4:** filter out and patch tainted control flow instructions

Output: patched executable binary bypassing all encountered code guards

Evaluation of the First Attack

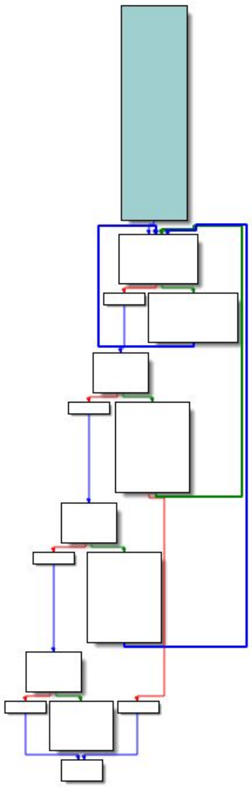


VirtSC – Original Implementation

- Function used:

```
void func() {
  // code guard here
  other_func();
}
```

- VirtSC: LLVM pass combining self-checksumming and virtualization obfuscation
- VirtSC doesn't read code from executable memory, but rather read-only data section
  - ➢ Taint analysis doesn't notice self-checksumming
- Code guard implemented as virtualized instruction



Original VirtSC

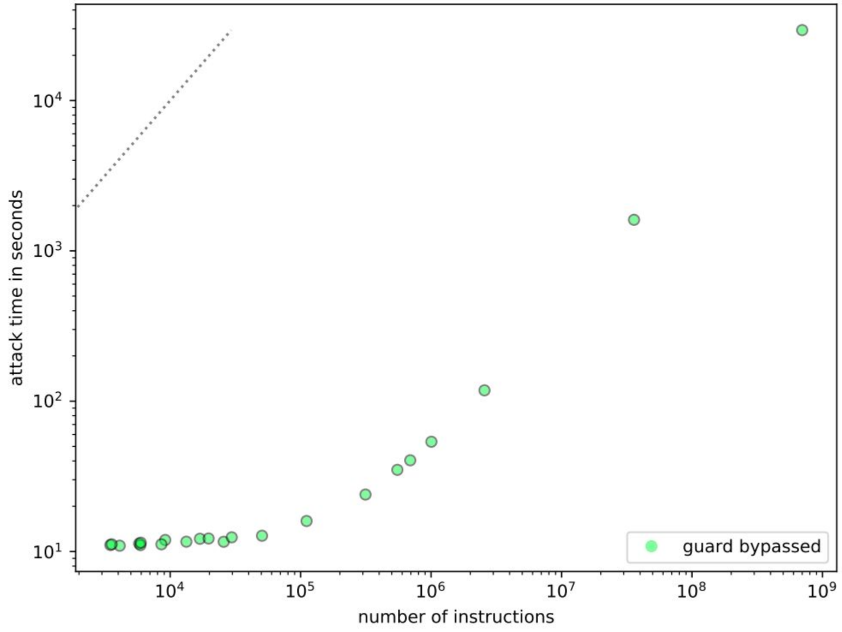
Improved Attack on VirtSC

Example trace of VirtSC:

```
movabs rdi, 0x401528 ; code array address
mov eax, 0x25 ; code array length
...
call 0x400690 ; hash function call
...
ret ; hash function return
...
cmp cx, ax ; checksum comparison
je 0x40102f
```

Evaluation of Improved Attack on Original VirtSC

Attack results and magnitude



Key Insights:

- Bypassed all guards
- Drawback: attack duration
- Disk space for trace could become problematic as well
- Issues are of rather technical nature

Updated VirtSC: Improving Original VirtSC

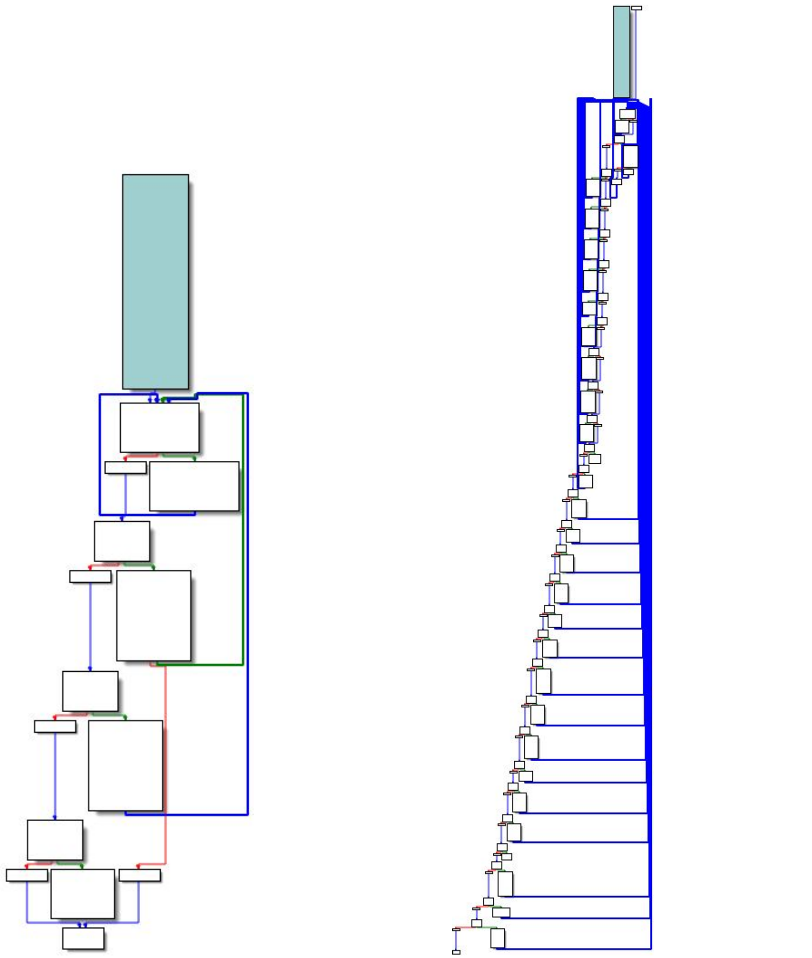
- Code guards’ instructions are virtualized as well
- **Result:**
  - Virtualized instructions inside & outside code guards
  - Code guards not bundled in machine code anymore

VirtSC – Version Comparison

- Function used:

```
void func() {
  // code guard here
  other_func();
}
```

- Code guard length in code array:
  2 vs. 110



Original VirtSC vs. Updated VirtSC

Conclusion and Future Work

- **Summary:**
  - Compared obfuscation techniques combined with self-checksumming
  - Automated attack against original VirtSC
  - VirtSC‘s security update
- **Key Insights:**
  - Virtualization obfuscation complicates dynamic taint analysis
  - Inlined code guards are harder to attack
- **Future work:** optimize performance overhead by avoiding placement of guards in hot code