A Lightweight Software Control System for Cyber Awareness and Security

Michele Co, Clark L. Coleman, Jack W. Davidson, Sudeep Ghosh, Jason D. Hiser, John C. Knight and Anh Nguyen-Tuong

University of Virginia
The Problem

- Modern society critically relies on properly functioning software
  - Transportation, power distribution, communication, water and sanitation systems
- Software is not and cannot be made perfect
  - Rigorous testing/validation still yield flawed software
  - Software bugs/vulnerabilities are a growing problem
- A method for improving the resilience of software is needed
Software Dynamic Translation (SDT)

- **The programmatic modification of a running program’s binary instructions**

If built correctly, SDT systems are reliable, transparent, low memory overhead, very fast and act as a control system.
Overview

- Introduction
- Overview
- Software Dynamic Translation
  - Strata
- Summary
Strata Control System Framework

**Build Time**
- Strata Software Dynamic Translator Library
- Application Code (Binary)
- Control Logic Library
- Application-specific Goals & Response Actions
- Sensor & Actuator Database
- Binary Rewriter

**Run Time**
- Fragment Cache
- Translated Application Code
-原件 Application Code
- Software Dynamic Translator (Strata)
- Controlled Application
- Actuators
- Sensors
- Goals & Response Actions
- Control Logic
Strata Virtual Machine

Application Binary

Context Capture

Dynamic Translator

New PC

Cached?

Context Switch

Finished?

New Fragment

Fetch

Decode

Translate

Next PC

Strata
Strata Virtual Machine

The Takeaway:
- Strata's fragment construction process for basic blocks ending in conditional branches
  - Fragment linking avoids excess overhead related to reentering the translator
This takes FOREVER, right?

UltraSPARC-Ill Results

Average overhead: 4% integer benchmarks: 8%
Worst-case overhead, most indirect branches: 33%

Evaluating Fragment Construction Policies for SDT Systems in VEE’06
Strata as a Control System

Application Controlled by MEDS

Translated Application Code

Original Application Code

Control Logic

Machine State Updater

Policy Selector

Crash History Database

Failed Policies

Viable Policies

Goals:
1) No Buffer Overflows
2) Avoid program restarts
Summary

• Software protections critical infrastructures and needs to be monitored
• Software Dynamic Translation provides a useful mechanism for such monitoring
  – Detecting memory errors
  – Detecting tampering
A Lightweight Software Control System for Cyber Awareness and Security

Questions?

Jason Hiser
University of Virginia
Backup Slides
Obfuscation and Anti-Tampering

- Anti-tampering (AT): making a program hard to (meaningfully) modify
- Obfuscation (Obf): making a program hard to understand
- Why?
  - Protect Intellectual Property (IP)
    - Preventing reverse engineering or code extraction
    - Digital watermarking and fingerprinting
  - Digital Rights Management (DRM)
  - Security
    - Anti-virus
    - Anti-hacker
    - Insider threats
  - Obfuscation used to hide AT techniques
Limitations of Previous Obf/AT Work

• Static
  – Applied once at software build time
  – Often NP-Hard if only static information is used, but dynamic information easily breaks many techniques

• Slow
  – High runtime/memory overhead

• Special Hardware
  – Trusted network connection with bounded time
  – Can trust that the CPU will generate result in bounded time

• Unrealistic threat model
  – OS, network, or memory trusted
  – Known optimal algorithms to calculate program checksums
The Problem with all Obf/AT work, even mine!

• For an *arbitrary* application, strong Obf/AT is impossible!
  – The player holds all the cards, eventually they will figure out how the application works and how to change it (Barak’01)

• The good news…
  – Sufficient to make an attacker’s job harder than re-writing the application
  – Some functions can be obfuscated (Wee’05, Hohenberger’07)
Goals

• Significantly stronger Obf/AT algorithms
  – Not just against static attacks, but against realistic hybrid dynamic/static attacks

• No reliance on custom, trusted HW/SW
  – Should work on machine currently in your office

• Efficient runtime overhead
  – 100x slowdown is unacceptable
    • If necessary, configurable tradeoff between protection level and overhead, perhaps on per-module basis
Guards (Atallah’02)

- Code segments that check some property of the program and react based on the outcome
  - Most commonly, check that the code is unchanged and subtly fail if tampering is detected.
  - Each guard can protect other guards to form a network
Guards Example

```c
chksm=0;
for(int i=start;i<end;i++)
    chksm+=*(int*)i;
%ebp+=chksm;
```

• **Advantages**
  – Provides circular protection
  – Reasonable overhead

• **Disadvantages**
  – Applied once at link time
  – Execution of guard may reveal its location
Guards with SDT

- Advantages: Guards copied to F$ differently in each run of the program, execution of guard does not reveal its location in the application text.

- Disadvantage: Can attempt to attack guards one at a time and guards still look the same during each execution of the program, even if at different locations.
Guards with SDT

- Advantages: Guards copied to F$ differently in each run of the program, execution of guard does not reveal its location in the application text.
- Disadvantage: guards still look the same during each execution of the program
Addressing Shortcomings

- Flush the F$ periodically
  - Move the guards around
- Encrypt the application code and decrypt on demand
  - Hides app. code from static disassembly/analysis
    - Hide key with white-box AES techniques
  - To succeed in an attack, encryption blocks must be modified as a unit
    - One-off changes, attacking guards one at a time, or playing what-if games with single instructions will fail!
But, isn’t the entire app. just in F$?

Preliminary Results: Case Study

SDT does well to start with, no more than 45% of application text in the F$!

Flushing helps

• Flushing every 1 sec. => less than 10% of app. text in F$
• Flushing every 0.1 sec. => less than 3% of app. text in F$
Dynamic transforms

- Apply dynamic Obf/AT transforms on fragments
  - Application never runs the same way twice
  - Guards appear different each time they execute!

- Examples
  - Dynamic disassembly resistance
  - Dynamic control flow graph obfuscation
  - Dynamic guards
  - Instruction morphing
  - Algebraic Identities
Dynamic Disassembly Resistance

• Goal – make it harder to disassemble F$

8048330: 80 3d ee ac 08 41 00 74 02 8d 84 c3 34 12 84 80

8048330: 80 3d ee ac 08 41 00  cmpb $0x0, *(0x4108acee)
8048337: 74 02                   je     804833b
8048339: 8d 84                   .byte 0x8d 0x84
804333b: c3                        ret
804333c: 34 12 84 80               ...

Lightweight transform performed randomly for each frag build
(Dynamic) Opaque Predicates

- Runtime generate predicates which are hard to decrypt after generation

```c
a = a->next;
b = b->next;
c = c->next;
if(a == b)
    ...
else if(b == c)
    ...
```

![Diagram of graph structures]
Surely all this must take forever!

Preliminary Results: Flushing + Encryption

- Flushing every 1 sec. => 3% slower than no flushing
- Flushing every 0.1 sec => Lots of slowdown.. but, maybe we can improve that
  - Selectively flushing
  - Using extra CPU’s in a multi-core machine
How well does dynamic Obf/AT protect applications?

• Continuing evaluation ongoing as part of CyberTrust’07 Grant
Overview

• Introduction
• Overview
• Strata
  – SDT Concepts
• SDT Applications
  – Obf/AT
• Related Work and Summary
Related Work - SDTs

• SDT Applications
  – Security policy enforcement (Code Diversity, Program Shepherding)
  – Software migration (Apple’s Rosetta)
  – Dynamic instrumentation (PIN, FIST)
  – Dynamic patching and debugging (Arachne)

• SDT Optimizations
  – Dynamic optimizers (Dynamo/DynamoRIO, JITs)
    • Bala, Duesterwald, Bruening, Suganuma, Arnold, …
  – Trace selection:
    • NET (Deusterwald’00)
    • LEI (Hohenberger’05)
  – Code cache management (Hazelwood’06)

• Many many more..
Related Work – Obf/AT

• Guards (Atallah ’02)
  – Breaking guards (Wurster’05),
  – Self-modifying guards (Giffin’05)
• Opaque Predicates (Collberg’98)
• Data Obfuscation (Collberg’98)
• Control flow flattening (Wang’00)
• Dynamic code mutation (Madou’05)
• So many others…
Summary

- **Software dynamic translation**
  - Efficient, powerful technology to dynamically modify programs
  - Low overhead
    - Recent optimizations yield only 4% slower than native execution for Spec2k benchmarks!

- **Obfuscation and anti-tampering**
  - Important for DRM/IP/Security
  - Current technology has many shortcomings against realistic threat models
  - Combining previous static techniques with SDT yields significantly stronger Obf/AT protection
Optimizing Software Dynamic Translation
(for Program Obfuscation and Anti-tampering)

Jason D. Hiser
http://www.cs.virginia.edu/~jdh8d

Questions?
Experimental Setup

• Strata running on 3 machines
  – Opteron 244, 1.8GHz, Linux, gcc 4.0
  – Xeon, 2.8GHz, Linux, gcc 3.3
  – UltraSPARC-Illl, 500MHz, Solaris 5.9, SUNWspro cc

• Results compared to no SDT

• Indirect branches handled efficiently with indirect branch translation cache mechanism
Fast Returns

- Translate call instructions to push fragment cache return address instead of application ret. addr.
  + Copy return instructions directly to F$
  = Fast returns

**Advantages:** Very fast, minimal F$ space

**Disadvantages:** May break some programs with nonstandard usage of call instruction.

**Alternatives:** Use IBTC/Sieve or Return Cache
How to Handle IBs, Option 3: Inline Mappings

- Instructions emitted at each branch to perform translation
- No hashing – compare app. address against inlined addresses

---

**Application Binary**

\[
\ldots
r1 = \ldots
\]

\[
\ldots
\]

\[
\text{L0: }
\]

\[
\ldots
\]

**Fragment Cache**

\[
\ldots
r1 = \ldots
\]

\[
\ldots
\]

\[
\text{save t0}
\]

\[
t0 = \text{APPADDR}_1
\]

\[
\text{if (r1 == t0)}
\]

\[
\text{jmp FRAGADDR}_100
\]

\[
\text{restore t0}
\]

\[
t0 = \text{APPADDR}_2
\]

\[
\text{if (r1 == t0)}
\]

\[
\text{jmp FRAGADDR}_120
\]

\[
\text{restore t0}
\]

\[
<\text{backing mechanism}>
\]
Indirect Branch Translation Cache

- Table in memory
  - **Advantage**: Small code footprint & minimal branches
  - **Disadvantage**: Memory accesses & data cache pressure
  - Other considerations
    - Uses two temporary registers & comparison

- Many options
  - Sharing (one for all branches or one per branch)
  - Appropriate size (number of entries)
  - Resizing (dynamically adjust size)
  - Reprobing (where to look on collision)
  - Lookup code placement
    - Inline in fragment or a separate “function”
Sieve

- Table as an instruction sequence
  - **Advantage**: Fewer data memory accesses
  - **Disadvantage**: More branches and possibly pressure on instruction cache
  - Other considerations
    - Uses *one* temporary register
    - Uses an *address-sized* constant compared to register

- Options
  - Table size
  - Others possible, but seem to not matter

16K-Entries
Back to Indirect Branches (IB)

How necessary is this? Aren’t indirect branches pretty rare?
The “rarity” of IBs
How to Handle IBs, Option 1: Indirect Branch Translation Cache

- Mapping done with **table in data memory** (memory accesses)
  - Table entry: `<AppAddr, FragAddr>`
- Table indexed by application address

---

**Application Binary**

```
L0:
  ...
  r1 = ...
  ...
  jmp r1
  ...
```

**Fragment Cache**

```
...
  r1 = ...
  ...
  save t0, t1
  t0 = hash(r1)
  if (IBTC[t0].AppAddr == r1)
    t1 = IBTC[t0].FragAddr
    jmp t1
    restore t0, t1
  else
    jmp translator
```
How to Handle IBs, Option 2: Sieve

- Mapping done by *executing instruction sequence*

---

**Sieve Table**

- **Addr16**
- **Addr10**

**Dispatch**

- **Addr8**
- **Addr4**

**Return To Translator**

- **Addr16**
- **Addr10**

**Bucket5**

- **Addr16**
- **Addr12**

**Bucket3**

- **Addr10**

**Bucket4**

- **Addr4**
- **Addr10**

**Bucket2**

- **Addr8**

**Frag10**

- **Frag99**
- **Frag111**
- **Frag16**
- **Frag204**

**Fragment Cache**
Combined: Inline Mapping

• Inlining mappings at indirect
  – **Advantage**: No hashing, no mem. access, min. branches
  – **Disadvantage**: Code growth & hit cost depends on hit entry
  – Other considerations
    • Possibly one register and constant address comparison to register

• Options
  – Number of inline entries
    • Should the translator decide the amount of inlining?
  – Target to inline
  – Execution point when that target should be selected
  – Backing mechanism to use (what to do on a miss)

It depends..
IBTC Vs. Sieve
UltraSPARC-IIIi

Sieve: 2 instructions to generate address-sized constant, more control transfers
IBTC Vs. Sieve
Pentium IV Xeon

- Sieve: 1 instruction to generate address-sized constant
- Sieve: No need to save/restore eflags for 16k-entries => Big win!

Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems in CGO’07
Why SDT for Obf/AT?

• Efficient: 2-10% overhead
• Monitors program execution
  – Dynamically apply Obf/AT transformations
  – Malicious user first has to figure out the SDT, then the application
    • Ever try to debug a program running under a simulator without source code?!
    – The SDT can protect the application, and the application can protect the SDT, circular level of trust
Unconditional Direct Branches

Elide direct branches (and calls) to avoid extra instructions
F$ Inefficiencies

- Each conditional branch transfers control to a trampoline
  + Trampolines patched to jump directly to target fragment
  = 2 F$ branches executed for every one executed branch in the original program!
- Patched trampolines leave wasted F$ space – reduced locality?
- Possible code duplication
  - 100 calls to strcpy() executed lead to 100 copies of the first basic block of strcpy thanks to partial inlining and unconditional branch eliding
Improving Performance

- **Advantages**
  - One branch in F$ for most branches in application text
  - Trampoline pool improves locality
  - Trampolines can be recycled

- **Disadvantages**
  - May translate unrequested basic blocks (waste of time and F$ space)
Fragment Construction Policies

1) Conditional branch policies
2) Unconditional branch policies
3) Call policies
   • Partial inlining
   • Lazy vs. eager target translation
4) Fragment alignment
5) Trampoline placement
Conditional Branch Handling

- Always stop translating
- Always continue translating
- Stop if…
  - Target already translated
  - Fall through already translated
  - Target OR fall through translated
  - Target AND fall through translated
Conditional Branches
Opteron 244

“Always continue” reduces overhead from 39% to 28% for integer benchmarks
Partial Inlining/Branch Eliding

- **Advantages**
  - Provides opportunity for optimization
  - Eliminates call/branch instructions

- **Disadvantages**
  - Increased code duplication
  - Calls not matched with return instructions => Bad branch predictor performance!
"No partial inlining" reduces overhead from 24% to 10% for integer benchmarks.