Software Security via Lightweight Process Virtualization

Jack W. Davidson
Department of Computer Science
University of Virginia

Joint Work

- This is joint work with many folks:
  - Bruce Childers^{+}, University of Pittsburgh
  - Michele Co, University of Virginia
  - William Hawkins*, University of Virginia
  - Sudeep Ghosh*, Microsoft
  - Jason Hiser^{+}, University of Virginia
  - John Knight, University of Virginia
  - Abbas Naderi*, University of Virginia
  - Anh Nguyen-Tuong, University of Virginia
  - Mary Lou Soffa, University of Virginia

Outline

- Part I: Brief Biography
- Part II: Introduction to SDT
- Part III: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance
- Part IV: Code security applications

Brief Biography

- Ph.D., University of Arizona, 1981
- Research Activities
  - Areas: Programming languages, compilers, computer security, and computer architecture
- Educational Activities
  - Wrote two introductory programming textbooks
  - Currently developing security curriculum sequence

Brief Biography

- Professor of Computer Science, University of Virginia
  - Located in Charlottesville, VA (100 miles south of Washington, DC). University founded by Thomas Jefferson, 3rd president of the United States. Also home of James Madison and James Monroe

Brief Biography

- Past and current research activities
  - Compilation for embedded systems
    - J. A. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser. Enabling Dynamic Translation in Embedded Systems with Scratchpad Memory. *ACM Transactions on Embedded Computing Systems*, **11**(4), December 2012, Article 89.
    - P. A. Kulkarni, W. Zhao, D. B. Whalley, X. Yuan, R. van Engelen, K. Gallivan, J. D. Hiser, J. W. Davidson, B. Cai, M. W. Bailey, H. Moon, K. Cho, Y. Paek, and D. L. Jones. VISTA: VPO Interactive System for Tuning Applications. *ACM Transactions on Embedded Computing Systems*, **5**(4), November 2006, pp. 819–863.
  - Compilation for many-core machines
    - W. Wang, T. Dey, J. W. Davidson, M. L. Soffa. DraMon: Predicting Memory Bandwdith Usage of Multithreaded Programs with High Accuracy and Low Overhead. Proceedings of the 20th IEEE International Symposium on High-Performance Computer Architecture, Orlando, FL, February 2014, pp. 380–391.
    - T. Dey, W. Wang, J. W. Davidson, M. L. Soffa. ReSense: Mapping Dynamic Workloads of Colocated Multi-threaded Applications Using Resource Sensitivity. *ACM Transactions on Architecture and Code Optimization*, 10(4), December 2013, Article No. 41.

Brief Biography

- Software security
  - S. Ghosh, J. D. Hiser, and J. W. Davidson, What’s the PointISA?, *Proceedings of the 2nd ACM Information Hiding and Multimedia Workshop*, Salzburg, Austria, June 2014, pp. 23–34.
  - S. Ghosh, J. D. Hiser, and J. W. Davidson, Software Protection for Dynamically-generated Code, Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, Rome, Italy, January 2013, pp. 1–12.
  - S. Ghosh, J. D. Hiser, and J. W. Davidson, Replacement Attacks against VM-protected Applications, *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, London, UK, March 2012, pp. 203–214.
  - S. Ghosh, J. D. Hiser, and J. W. Davidson, A Secure and Robust Approach to Software Tamper Resistance, *Proceedings of the 12th Information Hiding Conference*, Calgary, Canada, June 2010, pp. 33–47.
  - B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, and J. Knight, A. Nguyen-Tuong, J. D. Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. *Proceedings of the 15th USENIX Security Symposium*, Vancouver, B. C., Canada, August, 2006, pp. 105–120.

Brief Biography

- Professional Activities
  - Past-chair of ACM Special Interest Group on Programming Languages (SIGPLAN)
  - Member of ACM Executive Council (Co-chair ACM Publications Board)
  - Program Chair, HiPEAC 2018, January2018, Manchester, UK
  - Steering Committee, International Symposium on Code Generation and Optimization (next CGO in February 2018, Vienna, Austria)

Me

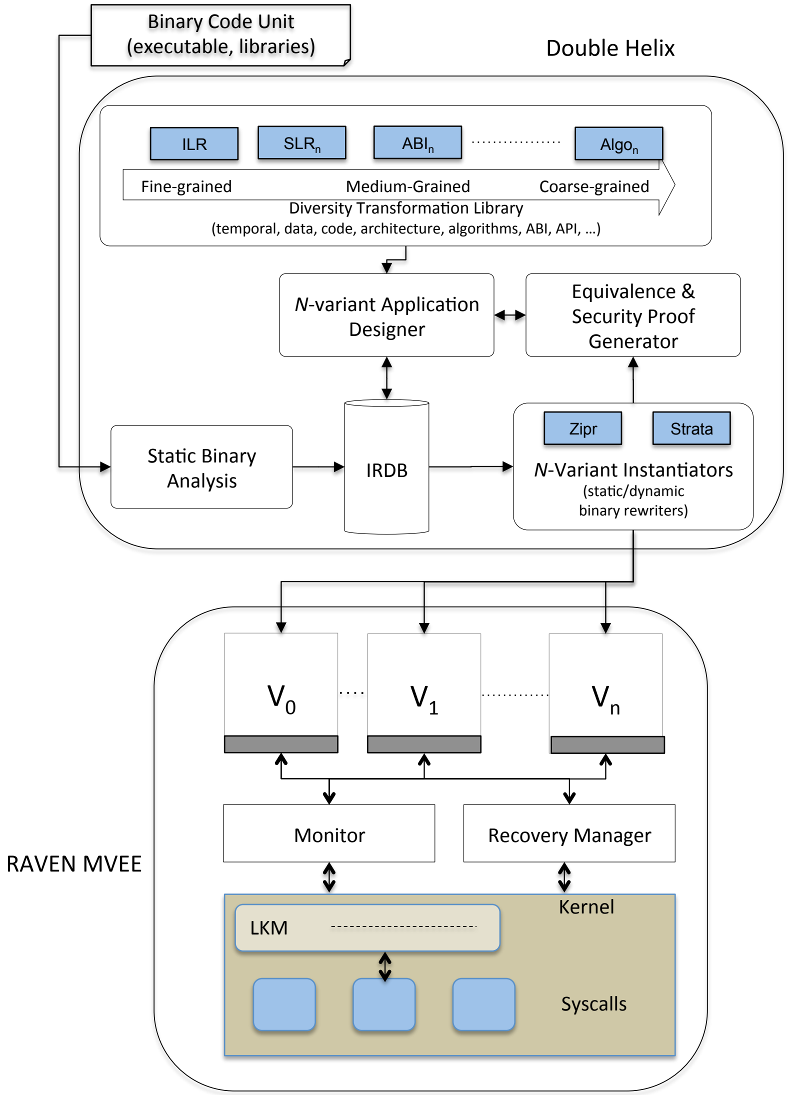
- Awards
  - ACM Fellow
  - IEEE Computer Society Taylor L. Booth Award
  - SIGPLAN Distinguished Service Award
  - Senior Member, IEEE

Computer Security & Modern Compilation

Indo US Engineering Faculty Leadership Institute
Computer Security and Modern Compilation
30 June to 04 July 2008
Global Education Center, Infosys Technologies Ltd., Mysore
Indo-US Collaboration for Engineering Education
IUCEE

Current Projects

- **Name**: Double Helix: High Assurance N-Variant Systems
- **Funding**: DARPA: $5.9M 48 months
- **Summary**: Provide *provable* security using structured diversity. Based on techniques described in our work "N-Variant Systems: A Secretless Framework for Security through Diversity," USENIX Security 2006.



Binary Code Unit (executable, libraries)

Double Helix

ILR

SLR_n

ABI_n

$Algo_n$

Fine-grained

Medium-Grained

Coarse-grained

Diversity Transformation Library

(temporal, data, code, architecture, algorithms, ABI, API, …)

N-variant Application Designer

Equivalence & Security Proof Generator

Static Binary Analysis

IRDB

Zipr

Strata

N-Variant Instantiators (static/dynamic binary rewriters)

V_0

V_1

V_n

Monitor

Recovery Manager

RAVEN MVEE

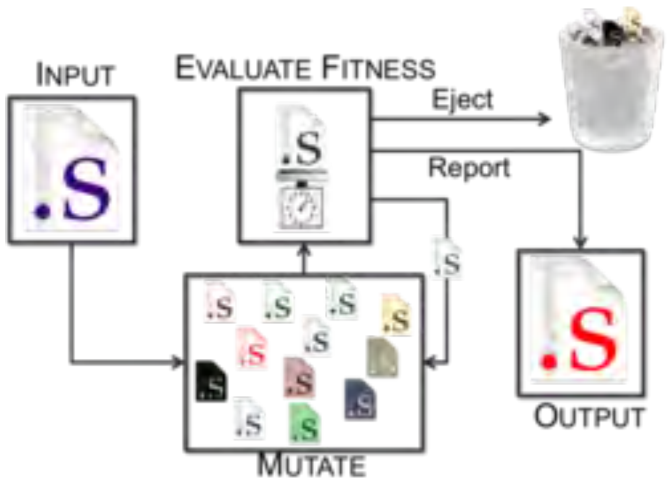
Kernel

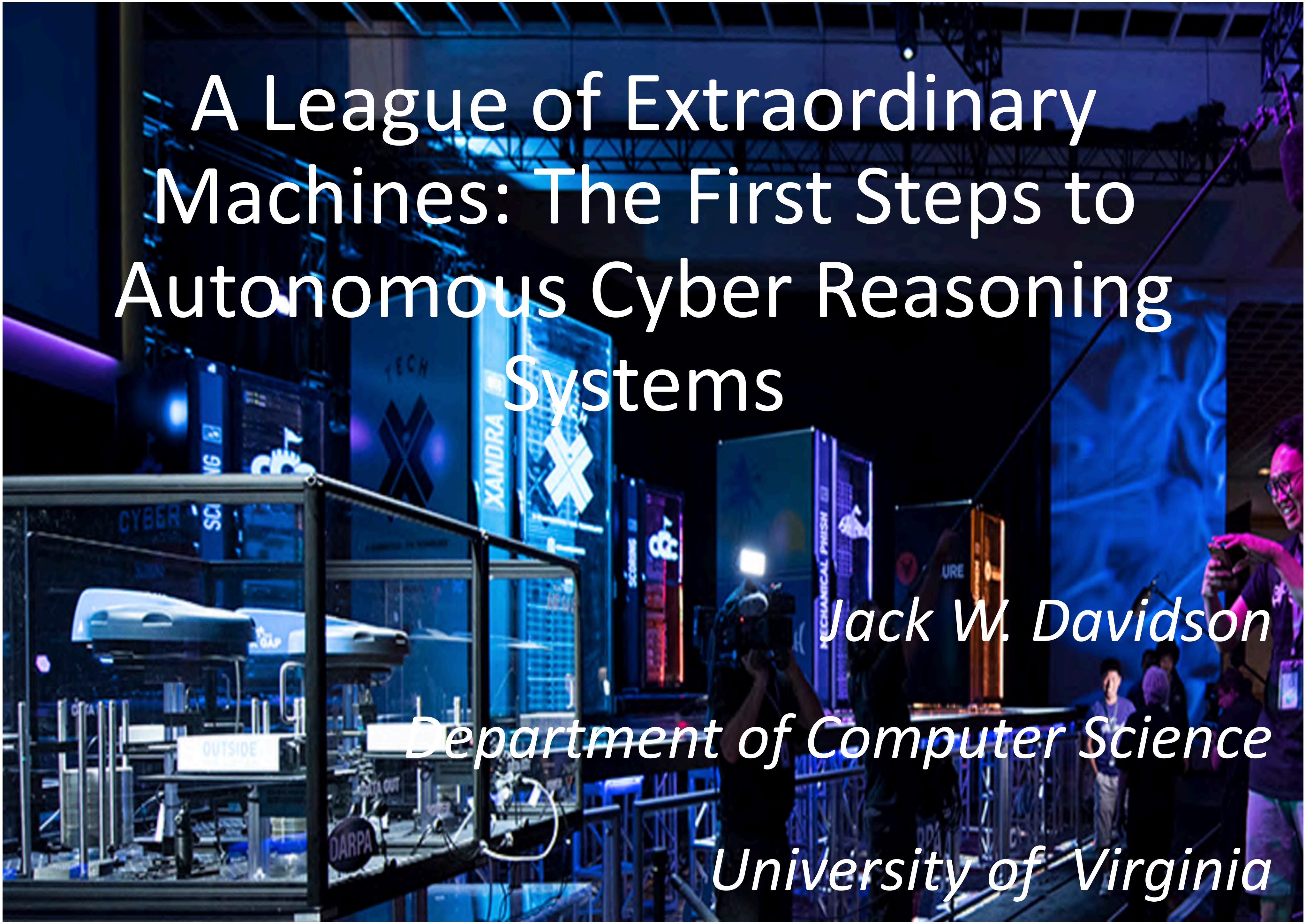
LKM

Syscalls

Current Projects

- **Name**: Trusted and Resilient Mission Operation
- **Funding**: AFRL: $1.25M 12 months
- **Summary**: Improve trust and resilience of cyber physical systems. Use diversity and genetic programming techniques to repair faulty programs.





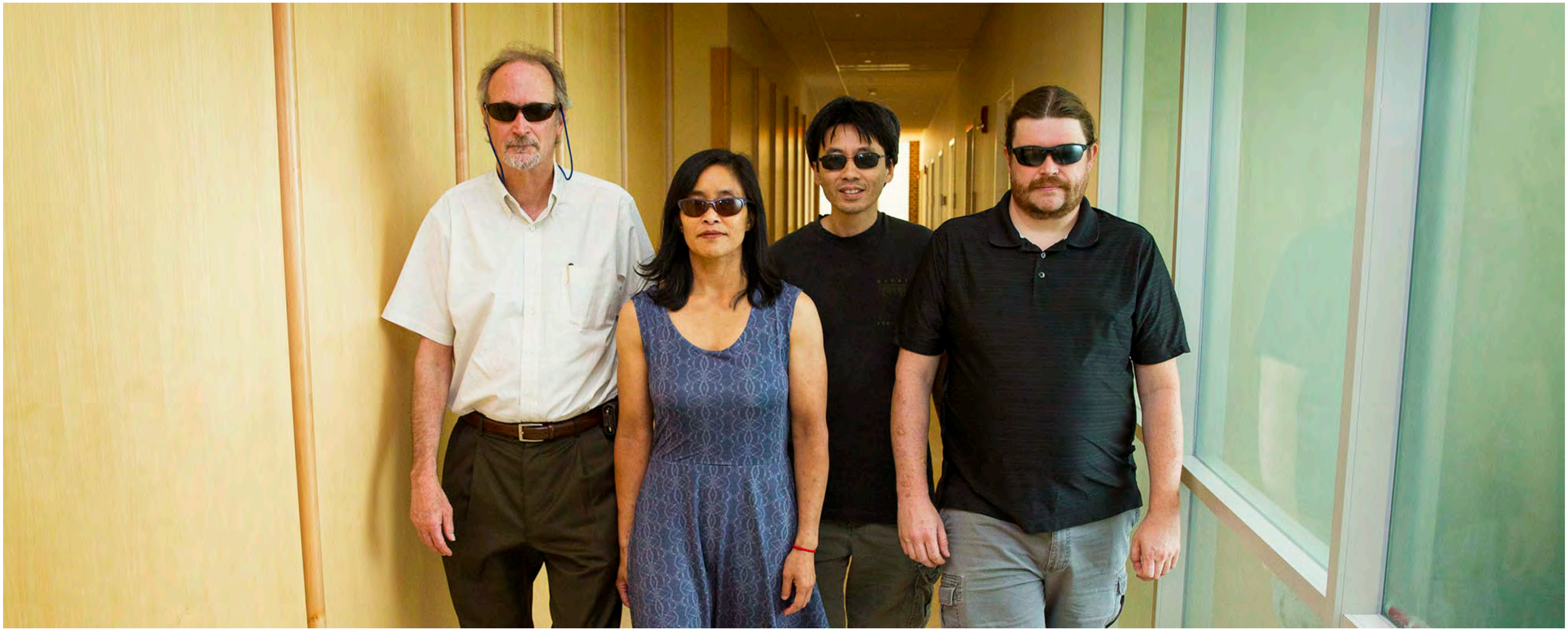
A League of Extraordinary Machines: The First Steps to Autonomous Cyber Reasoning Systems

Jack W. Davidson

Department of Computer Science

University of Virginia

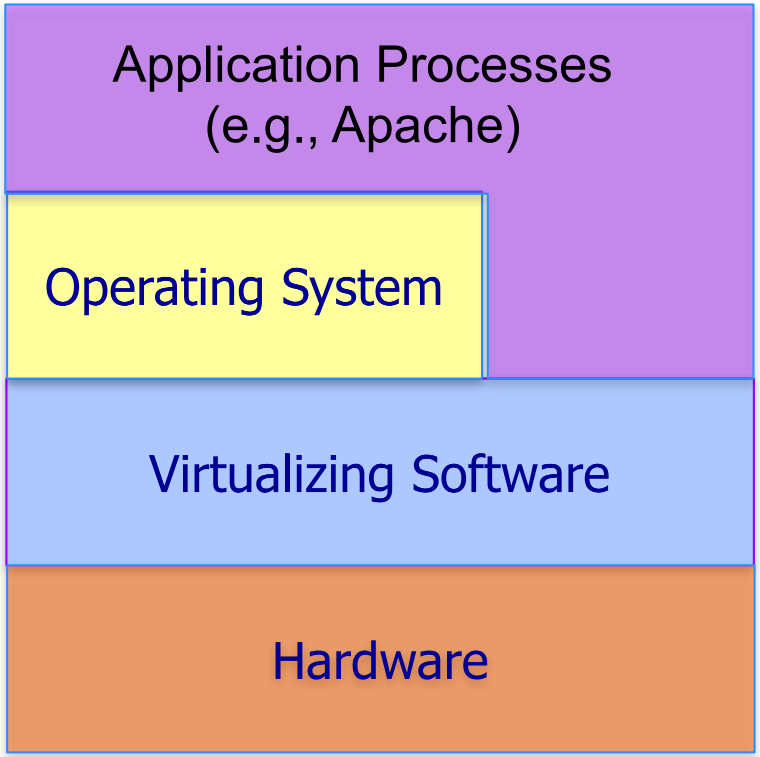
UVa TechX Team



Outline

- Part I: Brief Biography
- Part II: Introduction to SDT
- Part III: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance
- Part III: SDT Code security applications
  - Obfuscating Virtualization
  - Replacement Attacks
  - PointISAs
  - Matryoshka

Virtualization Approaches



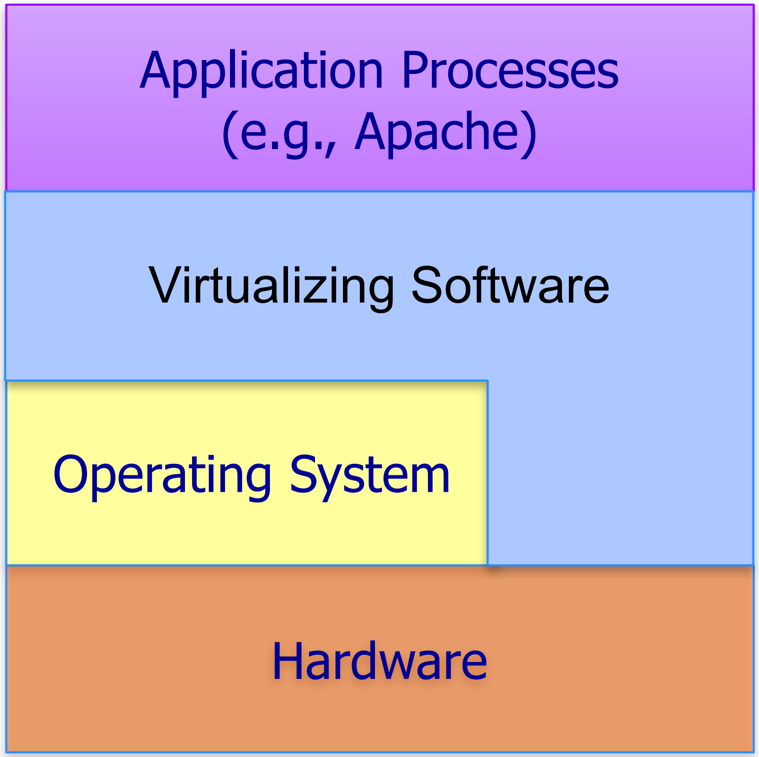
Application Processes (e.g., Apache)

Operating System

Virtualizing Software

Hardware

System-level virtual machine



Application Processes (e.g., Apache)

Virtualizing Software

Operating System

Hardware

Process-level virtual machine

Virtualization is Popular

OUR SERVERS ARE USING TOO MUCH ELECTRICITY. WE NEED TO VIRTUALIZE.
I DID MY PART BY READING ABOUT VIRTUALIZATION IN A TRADE JOURNAL. NOW YOU DO THE SOFTWARE PART.
WHY IS YOUR PART TAKING SO LONG?
scottadams@aol.com
www.dilbert.com
© Scott Adams, Inc./Dist. by UFS, Inc.
© Scott Adams, Inc./Dist. by UFS, Inc.

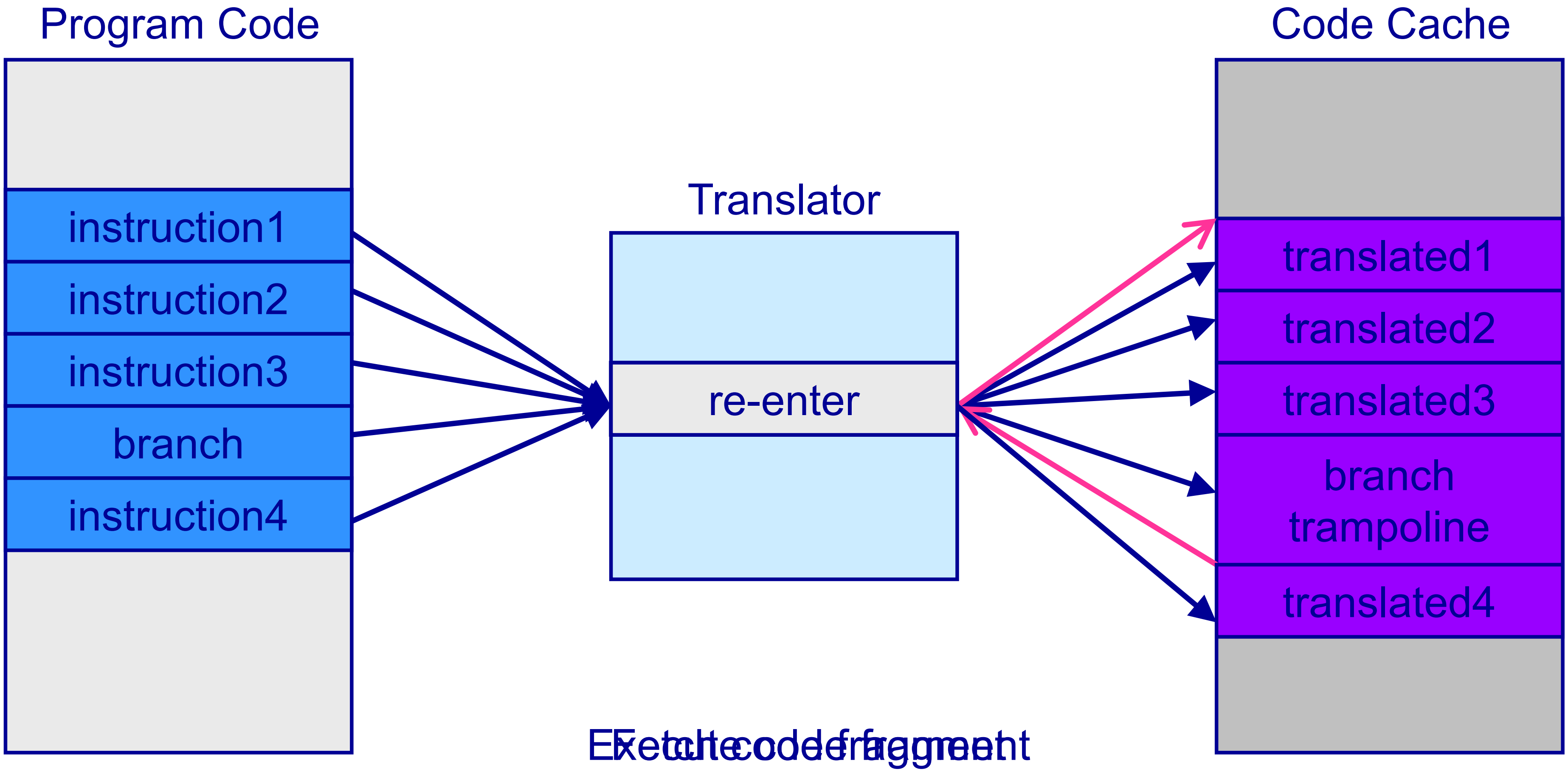
Process-level virtualization

- Software that intercepts, controls and modifies a software application as it runs
- May interpret or directly execute instructions
- Examines *every* instruction before the instruction is permitted to execute
- Translation can be customized to the specific program and goal (e.g., optimization, security, profiling, etc.)

What is SDT?

- Technique that uses software to control execution of another piece of software
- Subsumes:
  - Dynamic optimization / compilation
  - Dynamic binary translation
  - Dynamic instrumentation (e.g., profiling)
  - Process-level virtualization
  - Debugging

SDT Direct Execution & Cache



Fetch code fragment
until end of fragment condition

Execute code fragment
Until branch trampoline

Why Use SDT?

- Improve program performance
  - Adapt program to its execution environment
- Overcome economic barriers
  - Allow one architecture’s binaries to run on another
- Application specific ISA improvements
  - Code decompression, encryption
- Resource management
  - Power, memory footprint, resource protection
- Software engineering and dependability
  - Performance monitoring, fault isolation, security

Challenges

- Significant impediments to SDT uses
  1. Tightly tied to host platform
  2. It’s another layer: Run-time overhead
  3. Debugging translator and translated programs

Close Coupling to ISA & OS

- Steep learning curve
  - Low-level understanding of platform and translator
- Decoding & translating instructions
  - e.g., delay slots? variable length ISA? predicated execution?
- Switching context
  - e.g., how, what & where to save/restore?
- Signal and exception handling
  - e.g., ensure translator maintains control?
- Multi-threading
  - e.g., sharing data structures? locking?

Run-time Overhead

- Translator overhead
  - Fetch, decode, & translate loop
    - Possibly, re-translate already seen code
  - Context save/restore
  - Platform interactions with translated code (cache flushes, branch mispredictions, etc.)

- Translated code overhead
  - Invoking translator after execution (trampolines)
  - Handling control transfers (indirect branches are problems)
  - Program instrumentation needed by translator

Example Applications & SDTs

- Many existing and beneficial SDT uses
- Many systems – typically, re-implement translator for a new use of the technology
  - Architecture study (Shade, Embra)
  - Host virtualization (SimOS, VMware, Flex86)
  - Binary translation (DAISY, FX!32, Transmeta CMS, Co-designed VMs, Rosetta, Walkabout)
  - Code optimization (Dynamo, LLVM, ADORE)
  - Dynamic instrumentation and analysis (Pin, INSOP, Valgrind)
  - Code security (Dynamo/RIO)

Outline

- Part I: Brief Biography
- Part II: Introduction to SDT
- Part III: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance
- Part III: SDT Code security applications
  - Obfuscating Virtualization
  - Replacement Attacks
  - PointISAs
  - Matryoshka

Why do we need an SDT infrastructure?

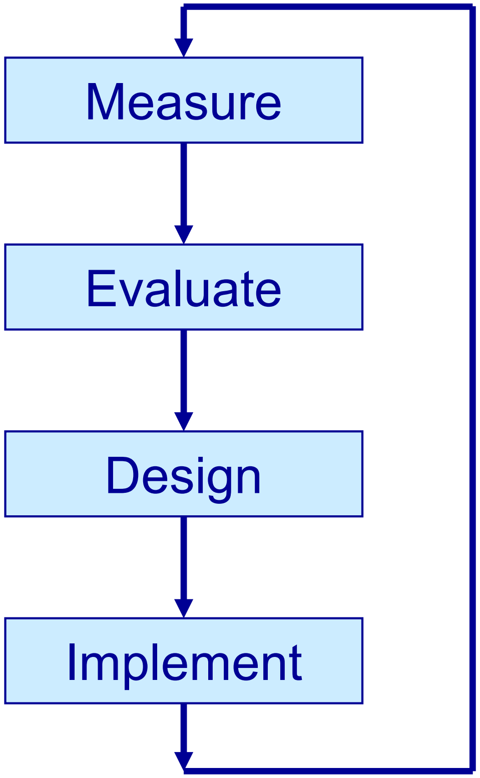
- Simple answer: make task-specific SDTs easier to develop
  - Allows experimentation with novel SDT systems
  - Accelerates research
  - Provides a model for other SDT systems
- How?
  - Factor out code needed across many SDTs
  - Allow for SDT reuse and composition
  - Provide support for multiple architectures to ease retargeting
  - Provide efficient translation mechanisms

Strata

- Infrastructure designed for building SDTs
- Designed with extensibility in mind
  - Optimization
  - Code compression
  - Performance monitoring
  - Dynamic resource management
  - Security
- Provides:
  - Platform independent common services
  - Target interface that abstracts target-specific support functions
  - Target-specific support functions

Challenges

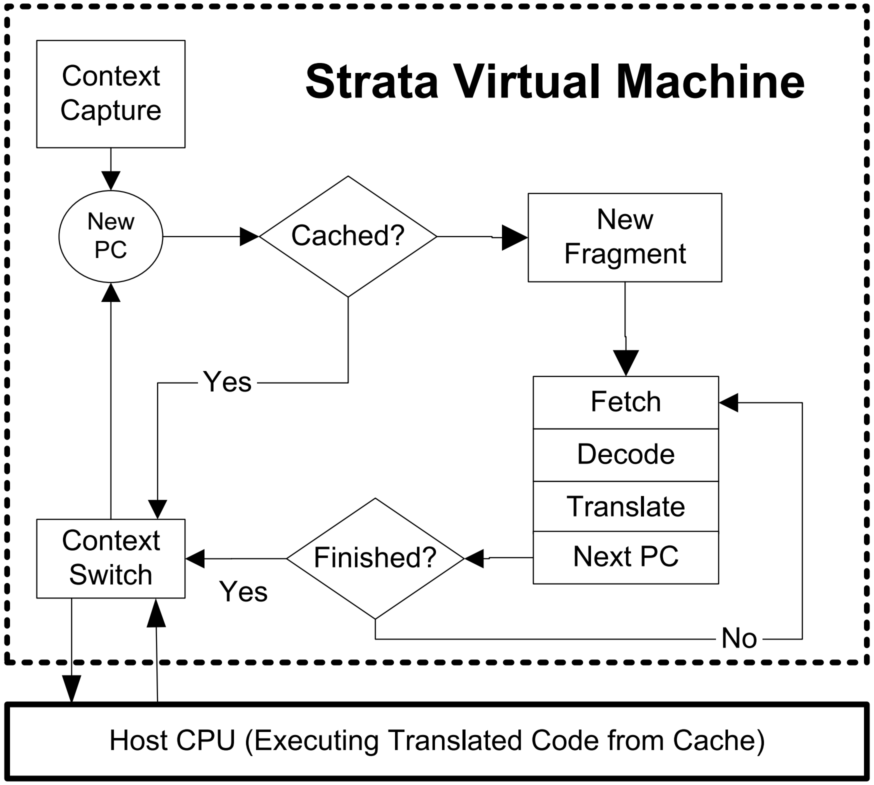
- Reconfigurability
  - Experiment with SDT implementation techniques
    - Memory management issues
    - Translation issues
  - Adapt for new uses
    - Optimization
    - Security
    - Performance Monitoring
    - Code Compression



Challenges

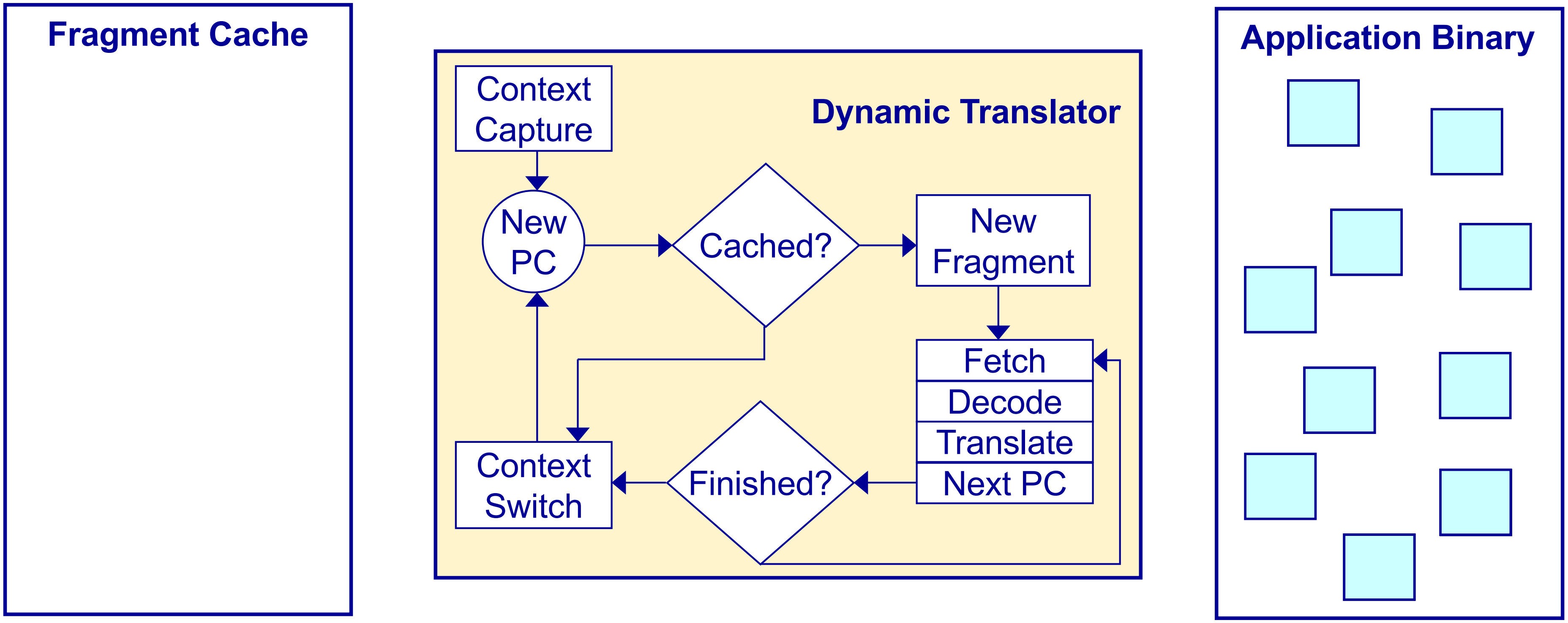
- Retargetability
  - Instruction-set architecture
    - Instruction decoding
    - Instruction semantics
    - Alignment
  - Operating system
    - Calling convention
    - Memory management and consistency
    - System calls
    - Signal and exception handling
    - Thread semantics
- Performance
  - Run-time overhead
  - Memory overhead

Strata Virtual Machine



- Base VM implements a simple SDT
- Programmer implements new SDTs by customizing the VM
- VM is customized by overriding functions in the target interface

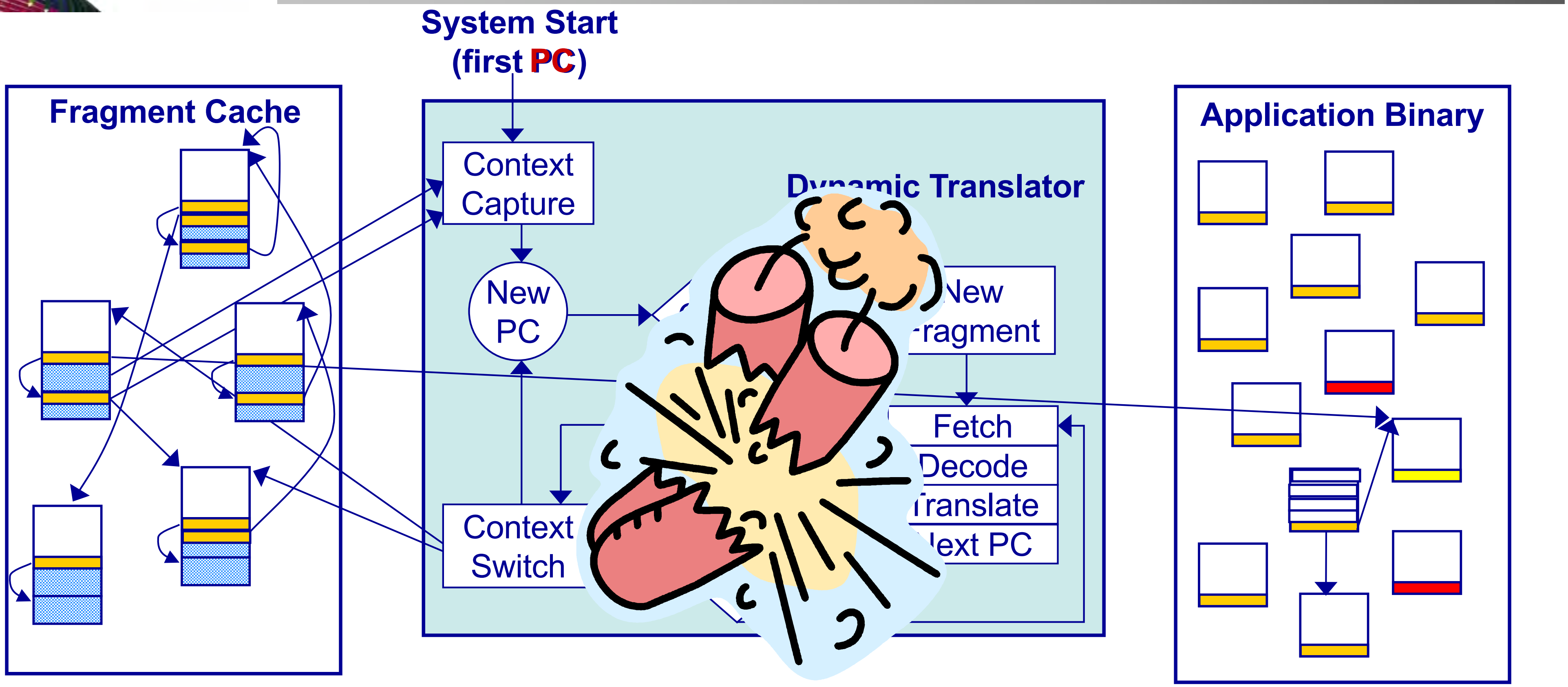
Strata Operation



Outline

- Part I: Brief Biography
- Part II: Introduction to SDT
- Part III: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance
- Part III: SDT Code security applications
  - Obfuscating Virtualization
  - Replacement Attacks
  - PointISAs
  - Matryoshka

Strata Virtual Machine



The Takeaway:

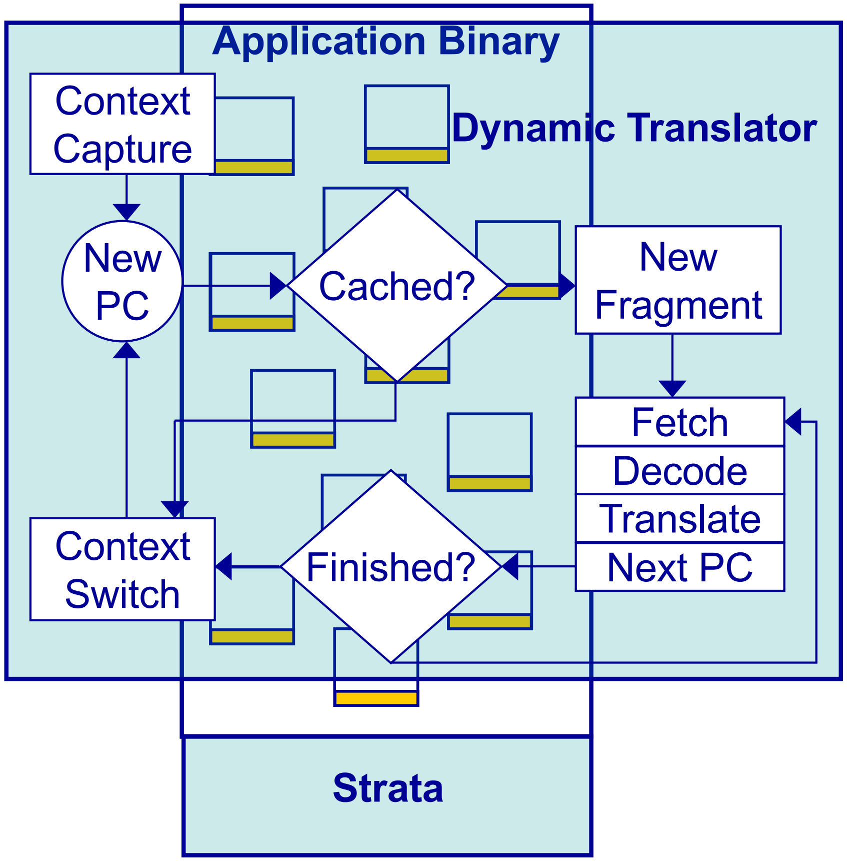
▨ Trampoline

- Strata's fragment construction process for basic blocks ending in conditional branches
- Fragment linking avoids excess overhead related to reentering the translator

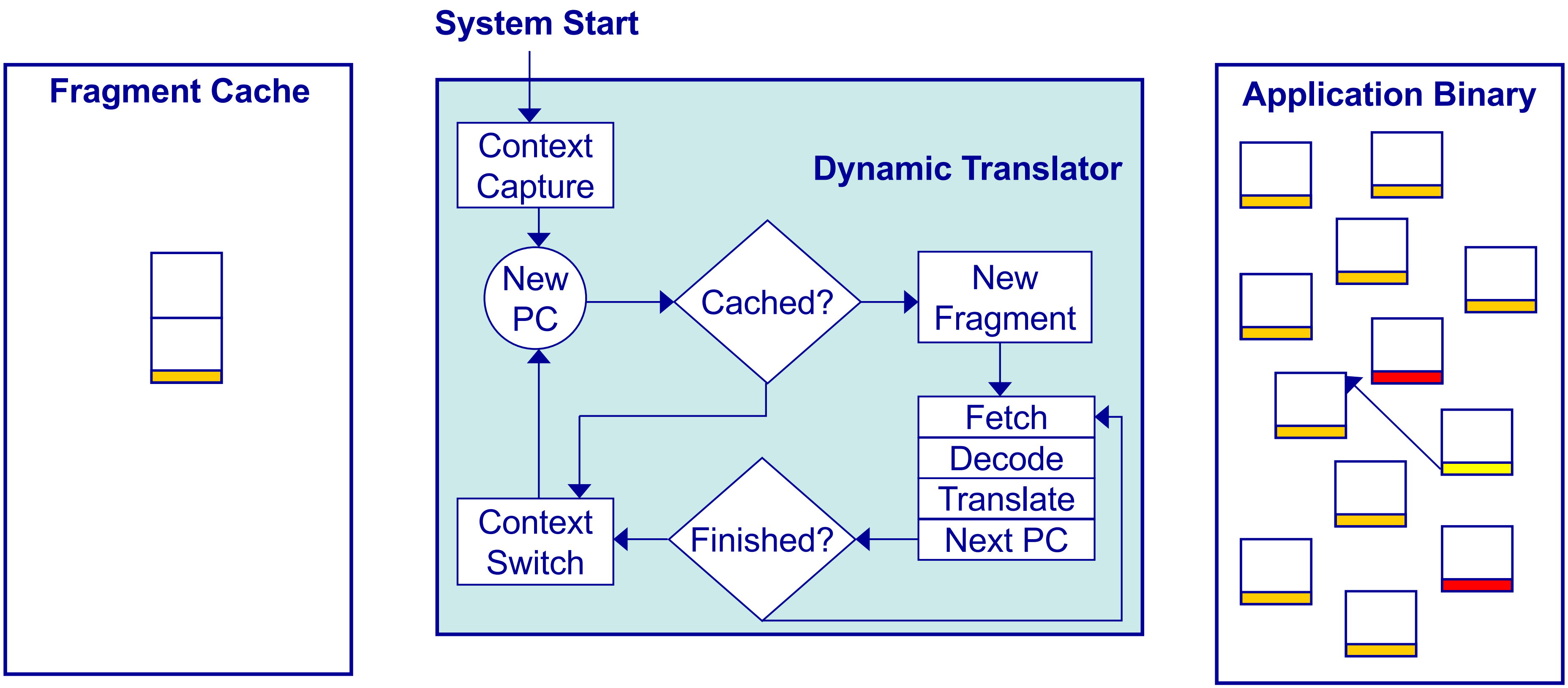
□ Non-control instruction

🟨 Direct Conditional branch

Strata Virtual Machine



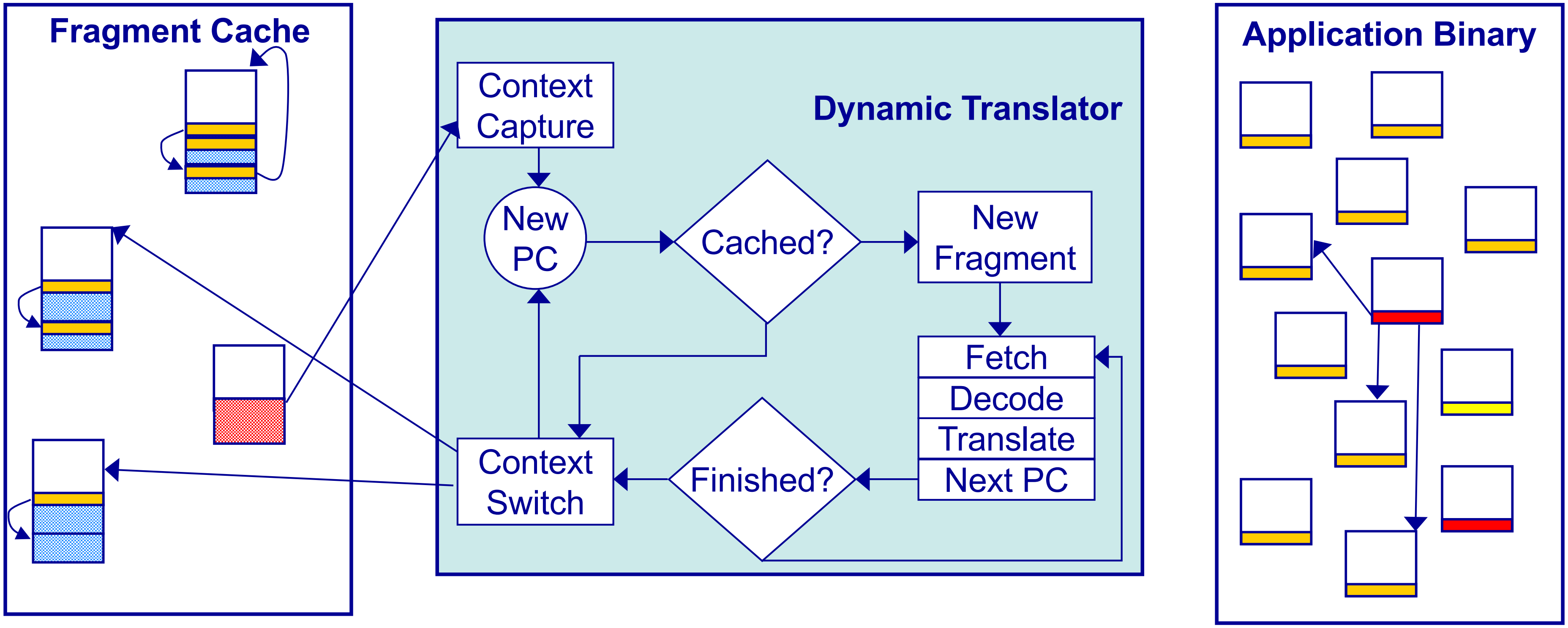
Unconditional Direct Branches



Elide direct branches (and calls) to avoid extra instructions

🟨 Direct Unconditional branch

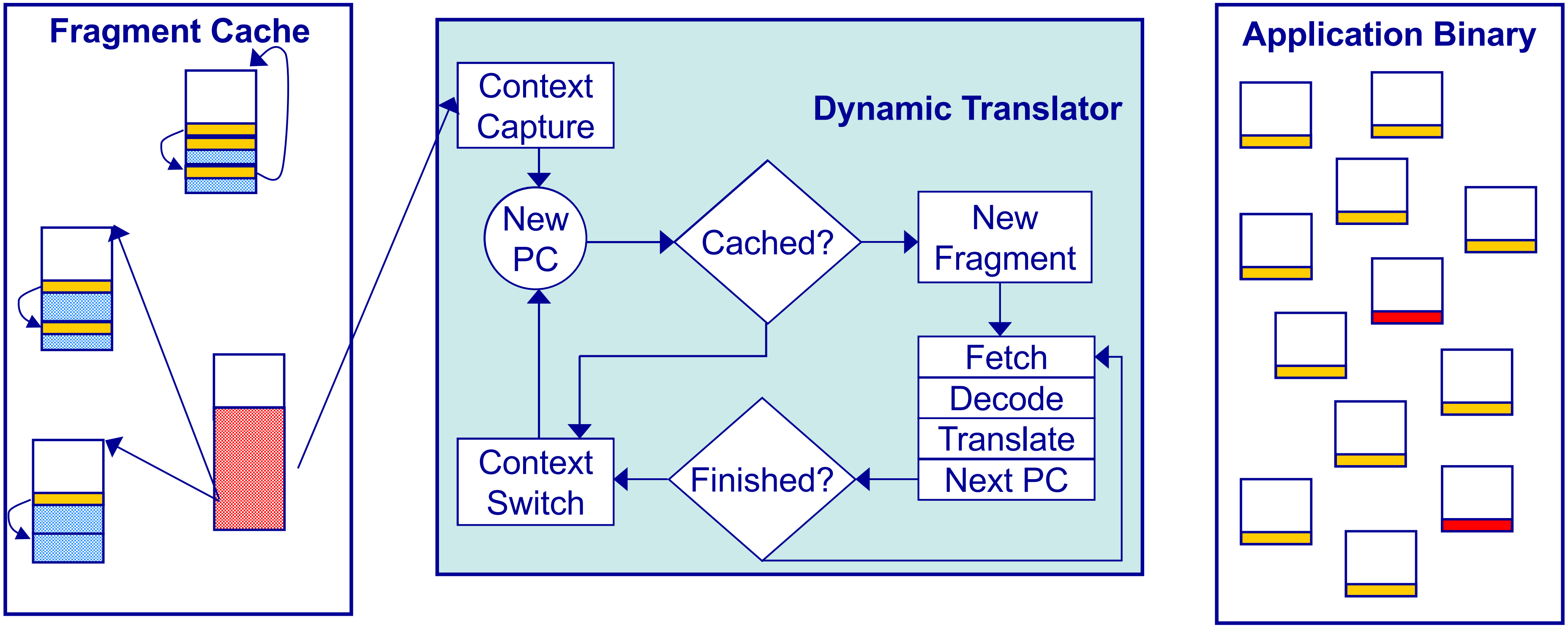
Handling Indirect Branches



🟥 Indirect branch

▨ IB Trampoline

Improving Indirect Branches Handling



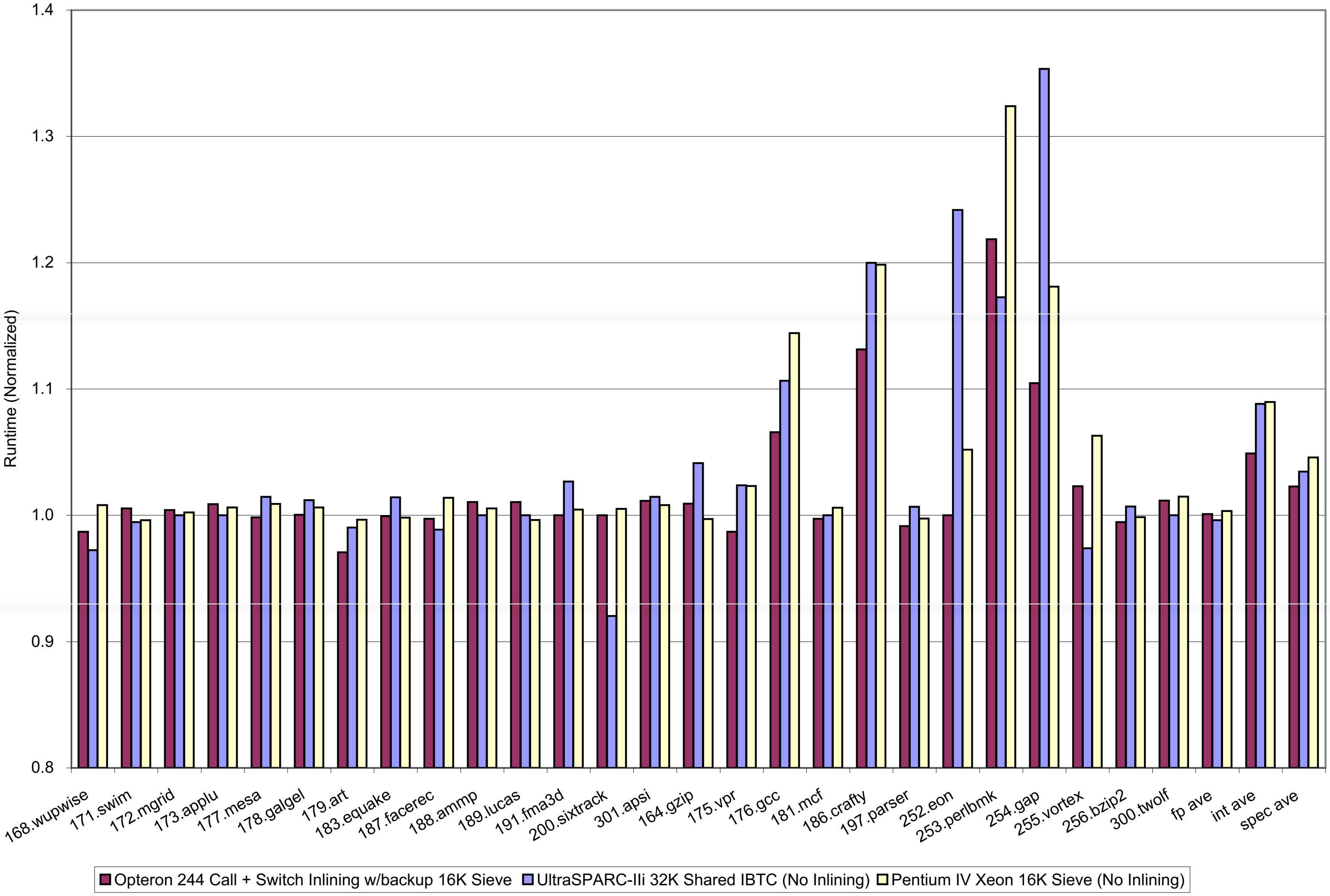
Embed lookup and mapping of application address into fragment cache

- Minimize amount of context to save & restore
- Efficient if done properly: *Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems* in CGO'07

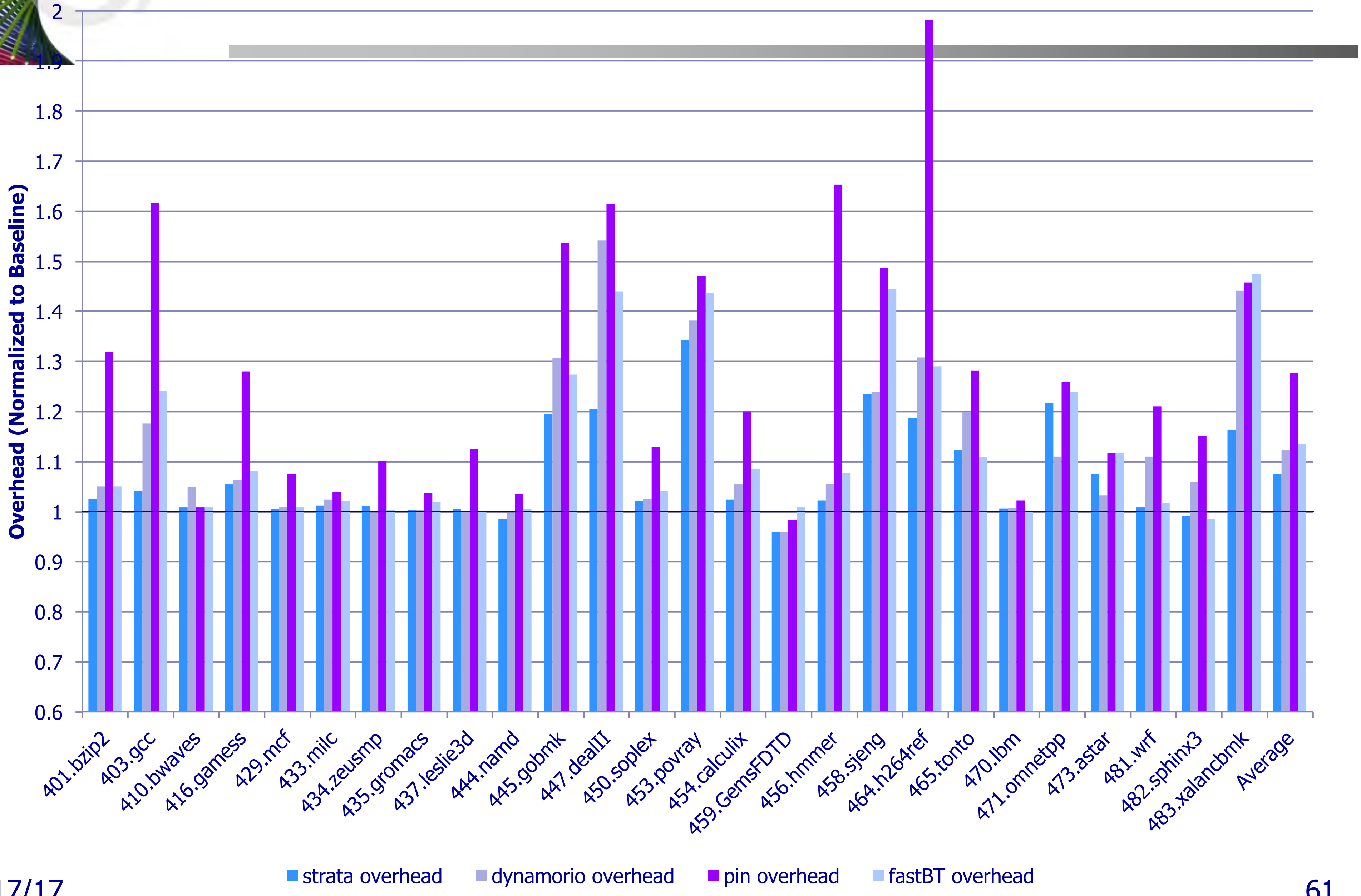
Outline

- Part I: Brief Biography
- Part II: Introduction to SDT
- Part III: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance
- Part III: SDT Code security applications
  - Obfuscating Virtualization
  - Replacement Attacks
  - PointISAs
  - Matryoshka

Performance



Performance

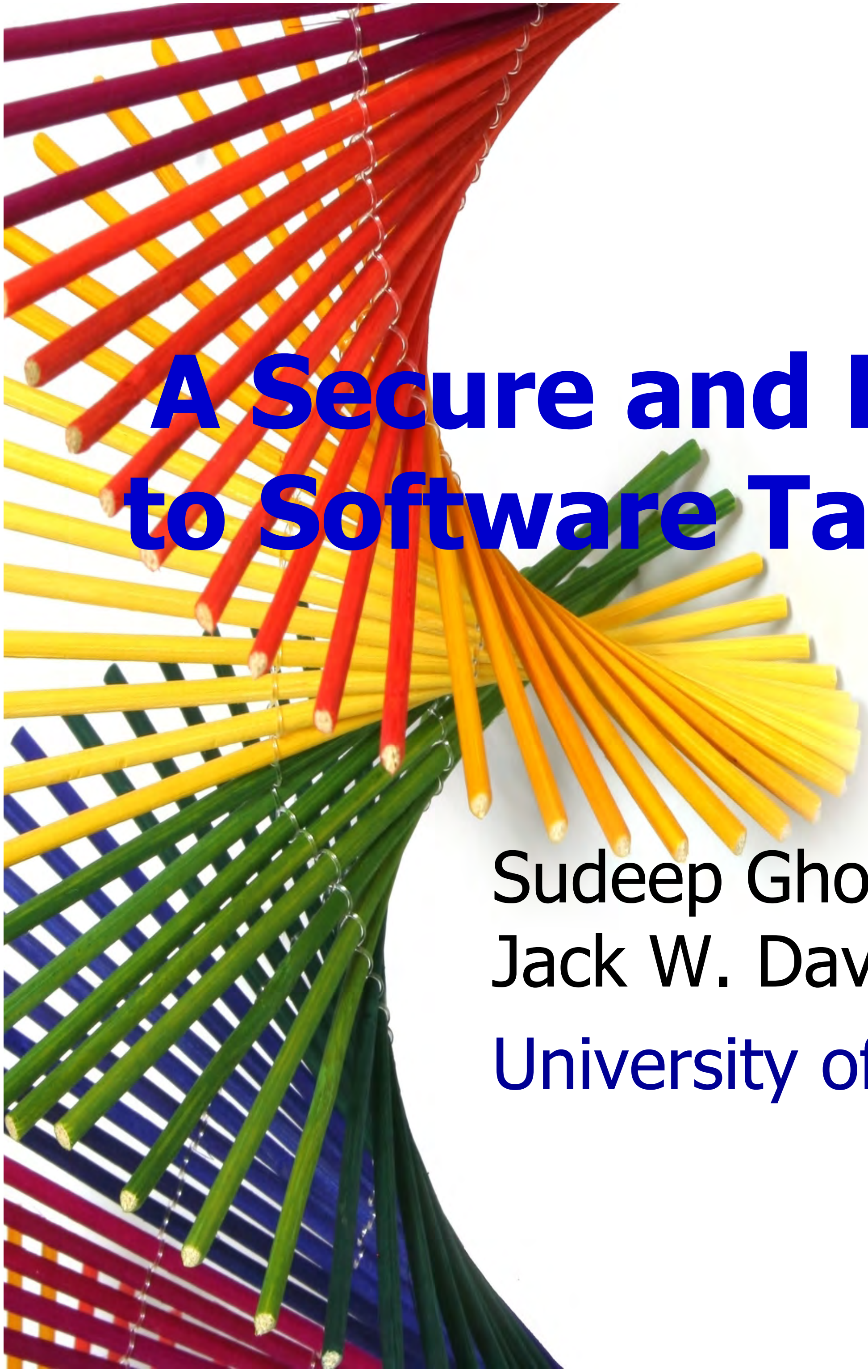


To Delve Further

- J. A. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser. Enabling Dynamic Translation in Embedded Systems with Scratchpad Memory. *ACM Transactions on Embedded Computing Systems*, **11**(4), December 2012, Article 89.
- J. D. Hiser, D. W. Williams, W. Hu, J. W. Davidson, J. Mars and B. R. Childers. Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems. *ACM Transactions on Architecture and Code Optimization*, **8**(2), July 2012, Article 9.
- J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems. *Proceedings of the 2007 International Conference on Code Generation and Optimization*, San Jose, CA, March 2007, pp. 61–73.
- J. Hiser, D. Williams, A. Filipi, J. W. Davidson and B. R. Childers. Evaluating Fragment Construction Policies for Software Dynamic Translation Systems. *Proceedings of the 2006 International Conference on Virtual Execution Environments*, Ottawa, CA, June 2006, pp. 122–132.
- K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson and M. L. Soffa. Retargetable and Reconfigurable Software Dynamic Translation. *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, San Francisco, CA, March 2003, pp. 36–47.

Outline

- Part I: Brief Biography
- Part II: Introduction to SDT
- Part III: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance
- Part III: SDT Code security applications
  - Obfuscating Virtualization
  - Replacement Attacks
  - PointISAs
  - Matryoshka



A Secure and Robust Approach to Software Tamper Resistance

Sudeep Ghosh, Jason D. Hiser, and Jack W. Davidson

University of Virginia

Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE 2006), Ottawa, Ontario

Secure and Practical Defense Against Code-injection Attacks using Software Dynamic Translation

Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans,
John C. Knight, Anh Nguyen-Tuong, Jonathan Rowanhill
Department of Computer Science
University of Virginia
{wh5a,jdh8d,dww4s,atf3r,jwd,evans,jck,an7s,jch8f}@cs.virginia.edu

Abstract

One of the most common forms of security attacks involves exploiting a vulnerability to inject malicious code into an executing application and then cause the injected code to be executed. A theoretically strong approach to defending against any type of code-injection attack is to create and use a process-specific instruction set that is created by a randomization algorithm. Code injected by an attacker who does not know the randomization key will be invalid for the randomized processor effectively thwarting the attack. This paper describes a secure and efficient implementation of instruction-set randomization (ISR) using software dynamic translation. The paper makes three contributions beyond previous work on ISR. First, we describe an implementation that uses a strong cipher algorithm—the Advanced Encryption Standard (AES), to perform randomization. AES is generally believed to be impervious to known attack methodologies. Second, we demonstrate that ISR using AES can be implemented practically and efficiently (considering both execution time and code size overheads) without requiring special hardware support. The third contribution is that our approach detects malicious code before it is executed. Previous approaches relied on probabilistic arguments that execution of non-randomized foreign code would eventually cause a fault or runtime exception.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]:Processors—Interpreters, runtime environments; D.4.6 [**Operating Systems**]: Security and Protection—Invasive Software

General Terms: Security

Keywords: Virtual Execution, Software Dynamic Translation

VEE '06 June 14–16, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1–59593–332–6/06/0006...$5.00.

1. Introduction

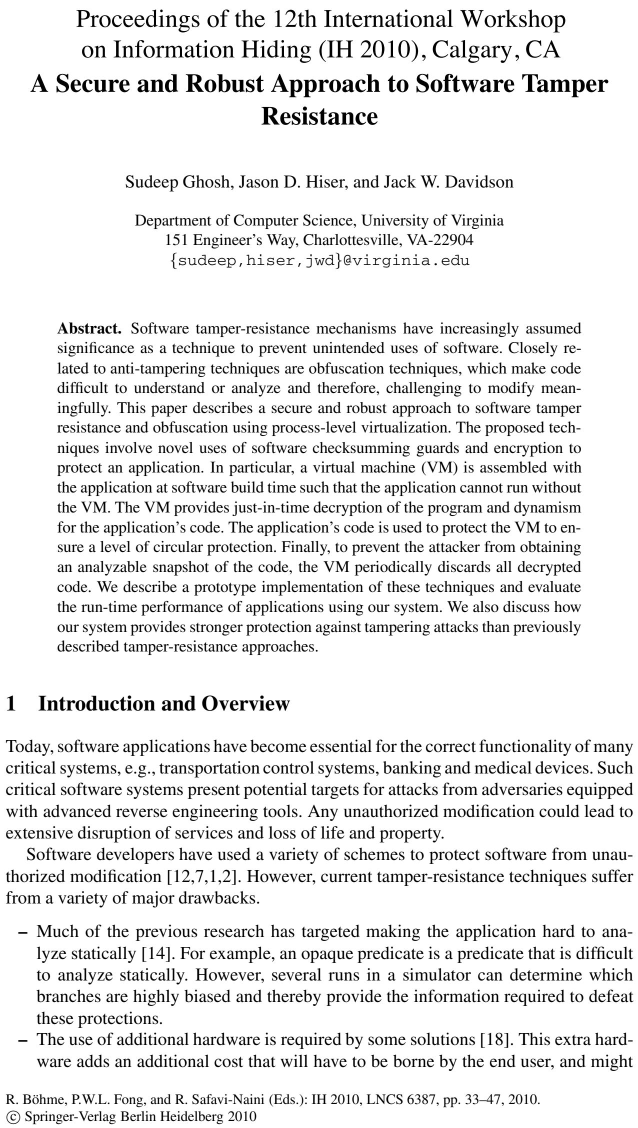
Despite heightened awareness of security concerns, security incidents continue to occur at alarming rates. In 2004, the Department of Homeland Security reported 323 buffer overflow vulnerabilities—an average of 27 new instances per month [13]. The most common attack to exploit a buffer overflow vulnerability is a *code-injection* attack. In a code-injection attack, an attacker exploits a vulnerability, e.g. a buffer overflow, to inject malicious code into a running application and then cause the injected code to be executed. The execution of the malicious code allows the attacker to gain the privileges of the executing program. In the case of programs that communicate over the network, such attacks can be used to break into host systems.

A theoretically strong approach to defending against any type of code-injection attack (irrespective of the vulnerability) is to create and use a process-specific instruction set that is created by a randomization algorithm. Code injected by an attacker who does not know the randomization key will be invalid for the randomized processor thereby thwarting the attack. Such an approach is known as randomized instruction-set emulation (RISE) or instruction-set randomization (ISR) [2, 9]. In this paper, we will use the term ISR exclusively.

The basic operation of an ISR system is as follows. An encryption algorithm (typically XOR'ing the instruction with a key) is applied to an application binary to encrypt the instructions. The encrypted application is executed by an augmented emulator (e.g., Valgrind [17] or Bochs [14]). The emulator is augmented to decrypt the application's instructions before they are executed.

When an attacker exploits a vulnerability to inject code, the injected code is also decrypted before emulation. Unless the attacker knows the encryption key/process, the resulting code will be transformed into, in essence, a random stream of bytes that, when executed, will raise an exception (e.g., invalid opcode, illegal address, etc.).

The security of ISR depends on several factors: the strength of the encryption process, protection of the encryption key, the security of the underlying execution process, and that the decrypted code will, when executed, raise an exception. The practicality of the approach is affected by the overheads in



Proceedings of the 12th International Workshop
on Information Hiding (IH 2010), Calgary, CA

A Secure and Robust Approach to Software Tamper Resistance

Sudeep Ghosh, Jason D. Hiser, and Jack W. Davidson

Department of Computer Science, University of Virginia
151 Engineer's Way, Charlottesville, VA-22904
{sudeep,hiser,jwd}@virginia.edu

Abstract. Software tamper-resistance mechanisms have increasingly assumed significance as a technique to prevent unintended uses of software. Closely related to anti-tampering techniques are obfuscation techniques, which make code difficult to understand or analyze and therefore, challenging to modify meaningfully. This paper describes a secure and robust approach to software tamper resistance and obfuscation using process-level virtualization. The proposed techniques involve novel uses of software checksumming guards and encryption to protect an application. In particular, a virtual machine (VM) is assembled with the application at software build time such that the application cannot run without the VM. The VM provides just-in-time decryption of the program and dynamism for the application's code. The application's code is used to protect the VM to ensure a level of circular protection. Finally, to prevent the attacker from obtaining an analyzable snapshot of the code, the VM periodically discards all decrypted code. We describe a prototype implementation of these techniques and evaluate the run-time performance of applications using our system. We also discuss how our system provides stronger protection against tampering attacks than previously described tamper-resistance approaches.

1 Introduction and Overview

Today, software applications have become essential for the correct functionality of many critical systems, e.g., transportation control systems, banking and medical devices. Such critical software systems present potential targets for attacks from adversaries equipped with advanced reverse engineering tools. Any unauthorized modification could lead to extensive disruption of services and loss of life and property.

Software developers have used a variety of schemes to protect software from unauthorized modification [12,7,1,2]. However, current tamper-resistance techniques suffer from a variety of major drawbacks.

- Much of the previous research has targeted making the application hard to analyze statically [14]. For example, an opaque predicate is a predicate that is difficult to analyze statically. However, several runs in a simulator can determine which branches are highly biased and thereby provide the information required to defeat these protections.
- The use of additional hardware is required by some solutions [18]. This extra hardware adds an additional cost that will have to be borne by the end user, and might

R. Böhme, P.W.L. Fong, and R. Safavi-Naini (Eds.): IH 2010, LNCS 6387, pp. 33–47, 2010.

Threat model

- Adversary has physical access to system (MATE)
- Adversary controls execution environment
  - Execute directly and observe
  - Simulate and observe
  - Provide false inputs
  - Run repeatedly
  - Use sophisticated dynamic analysis tools and algorithmic attacks
- White-box attack where the adversary "holds all the cards"
  - Example, HD protection cracked (http://www.theregister.co.uk/2007/02/14/aacs_hack/)

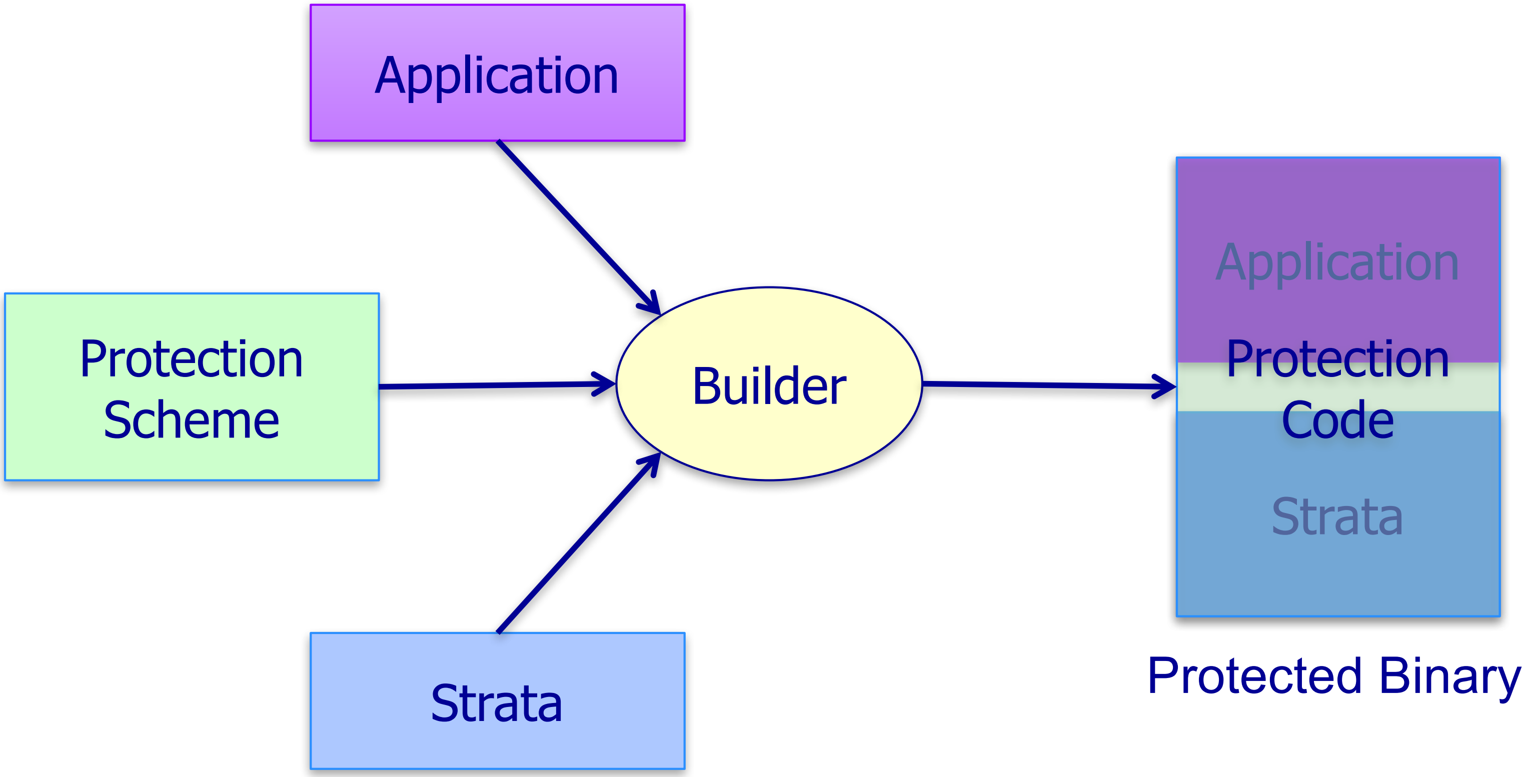
Goal

Use SDT to hamper automated analysis and modification of programs.

PVM-based Protections

- Advantages
  - Allows run-time monitoring and checking of code
  - Enables modular design
  - Retrofit unprotected applications with security mechanisms
  - Enhance and strengthen static protection techniques
- Examples
  - VMProtect (http://www.vmpsoft.com/)
  - Proteus (Anckaert B., et al.: Proteus: virtualization for diversified tamper-resistance. In: *Proceedings of the ACM Workshop on Digital Rights Management)*
  - StarForce (http://www.star-force.com/)

Protection Architecture



Protected Binary

Dynamic diversity

- Use SDT to introduce artificial diversity into applications

- Advantages
  - Binary only (no source needed)
  - Wide range of transformations possible
  - Transformations can be applied (or reapplied) at any point during execution
  - Handles untrusted code (libraries, third party components, etc.)
  - Prevents exploitation of both unintentional and intentional software vulnerabilities
  - Provides facilities for software protection (i.e., code obfuscation and tamper-proofing)

- Disadvantages
  - Can degrade performance (if not done properly)
  - Accountability

Attack Class Coverage*

Attacks Covered	ILX	SLX	HLX	ISR	PCC	DNA	SCFI	Combined
Command Injection Attacks	○	○	○	○	○	●	○	●
Code Injection Attacks	●	⭘	⭘	●	●	○	●	●
Arc Injection Attacks	●	⭘	⭘	○	⭘	○	●	●
ROP Attacks	●	⭘	⭘	○	⭘	○	●	●
Buffer Overflow Attacks	⭘	⭘	⭘	⭘	⭘	○	⭘	●
Number Handling Attacks	⭘	⭘	⭘	○	⭘	○	○	⭘

Legend: ○ Low coverage ⭘ Medium Coverage ● High Coverage

**Covers #1, #2, #3 in MITRE's Top 25 ranking of most dangerous software errors*

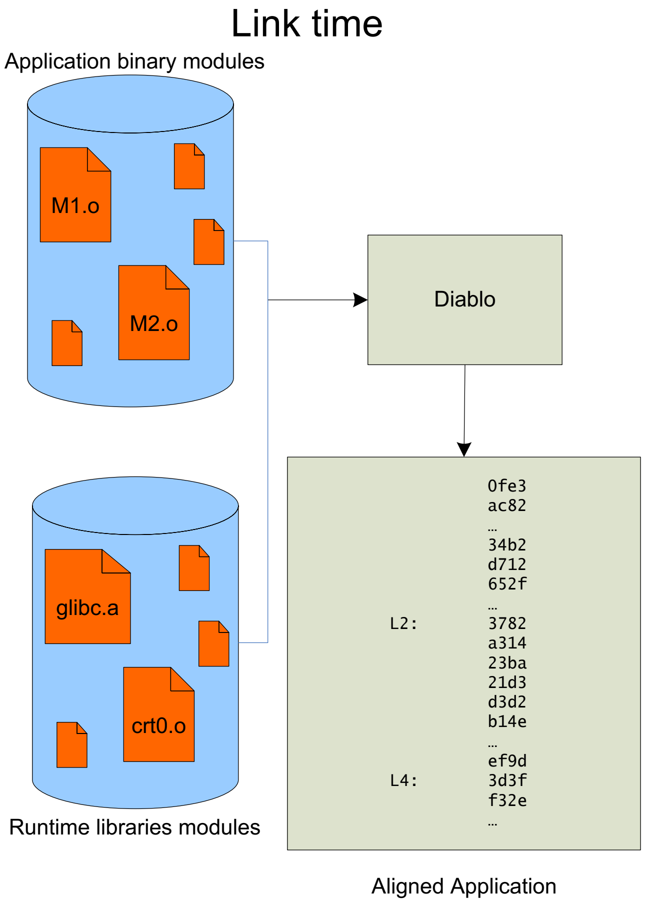
Idea

- What if we could create lots of different machines?
  - Cracker would not know what machine they were attacking
  - Cracker could not easily use static analysis
- Proposed by Harold Thimbleby
  - "Can Viruses Ever be Useful", Computers and Security **10**(2), 1991.
  - Also investigated by Barrantes et al. (Trans. On Info. Security **8**(1)) and Kc et al. (CCS 2003).

Creating Different Machines

- Use encryption to create a new instruction set at load time (random key is generated at load time too)
- Use Strata to decrypt at runtime
- Adversary cannot use static analysis techniques—they must use dynamic analysis
- See “Secure and Practical Defense Against Code-injection Attacks using Software Dynamic Translation”, presented at VEE 2006 in Ottawa, Canada.

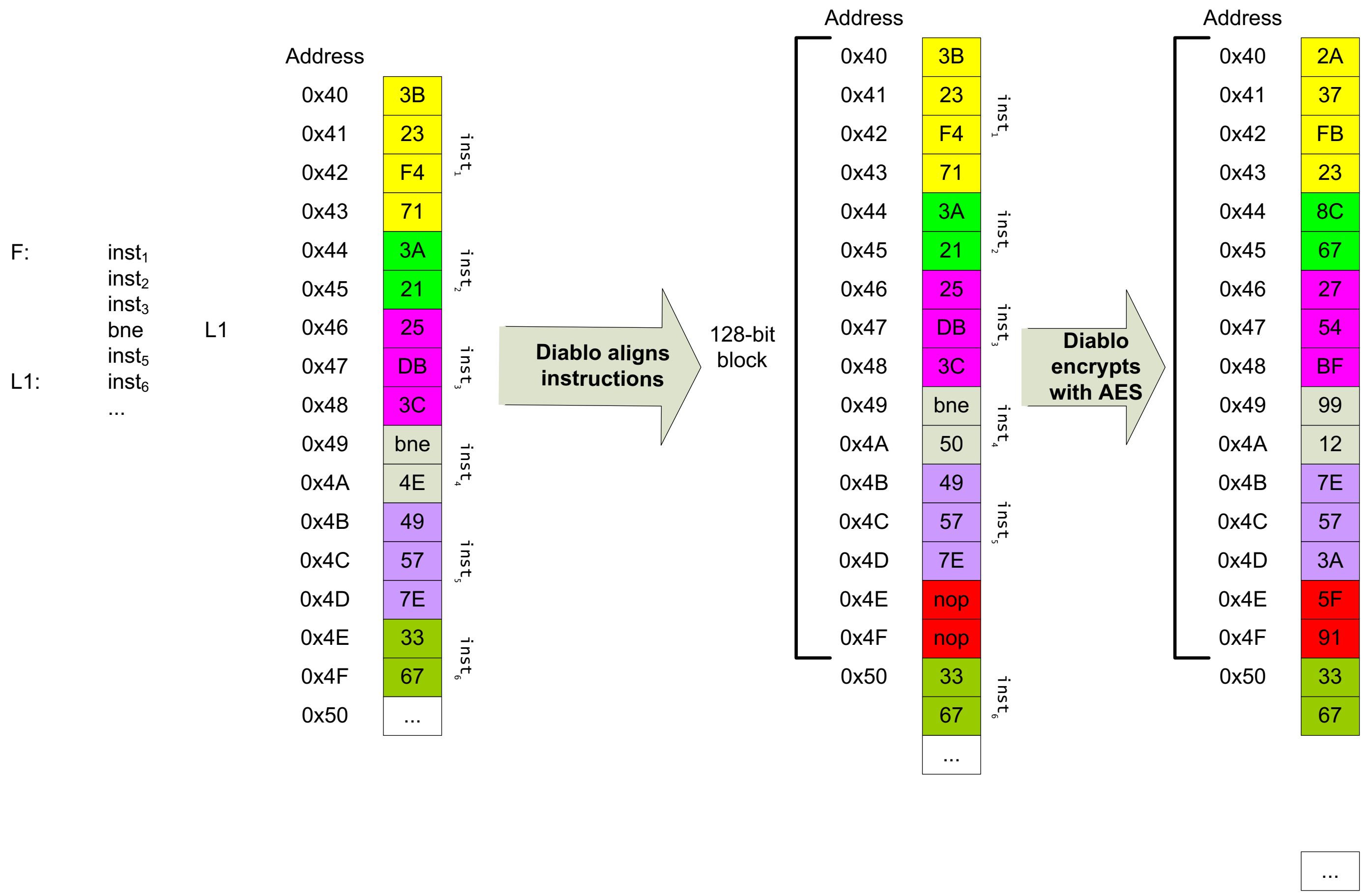
Diablo: Link-time processing



```
        0fe3
        ac82
        …
        34b2
        d712
        652f
        …
L2:     3782
        a314
        23ba
        21d3
        d3d2
        b14e
        …
        ef9d
L4:     3d3f
        f32e
        …
```

- Diablo is a link-time editor developed at Ghent University
- We use Diablo to align x86 instructions so that we can use a strong block encryption algorithm such as AES

Diablo: Link-time processing



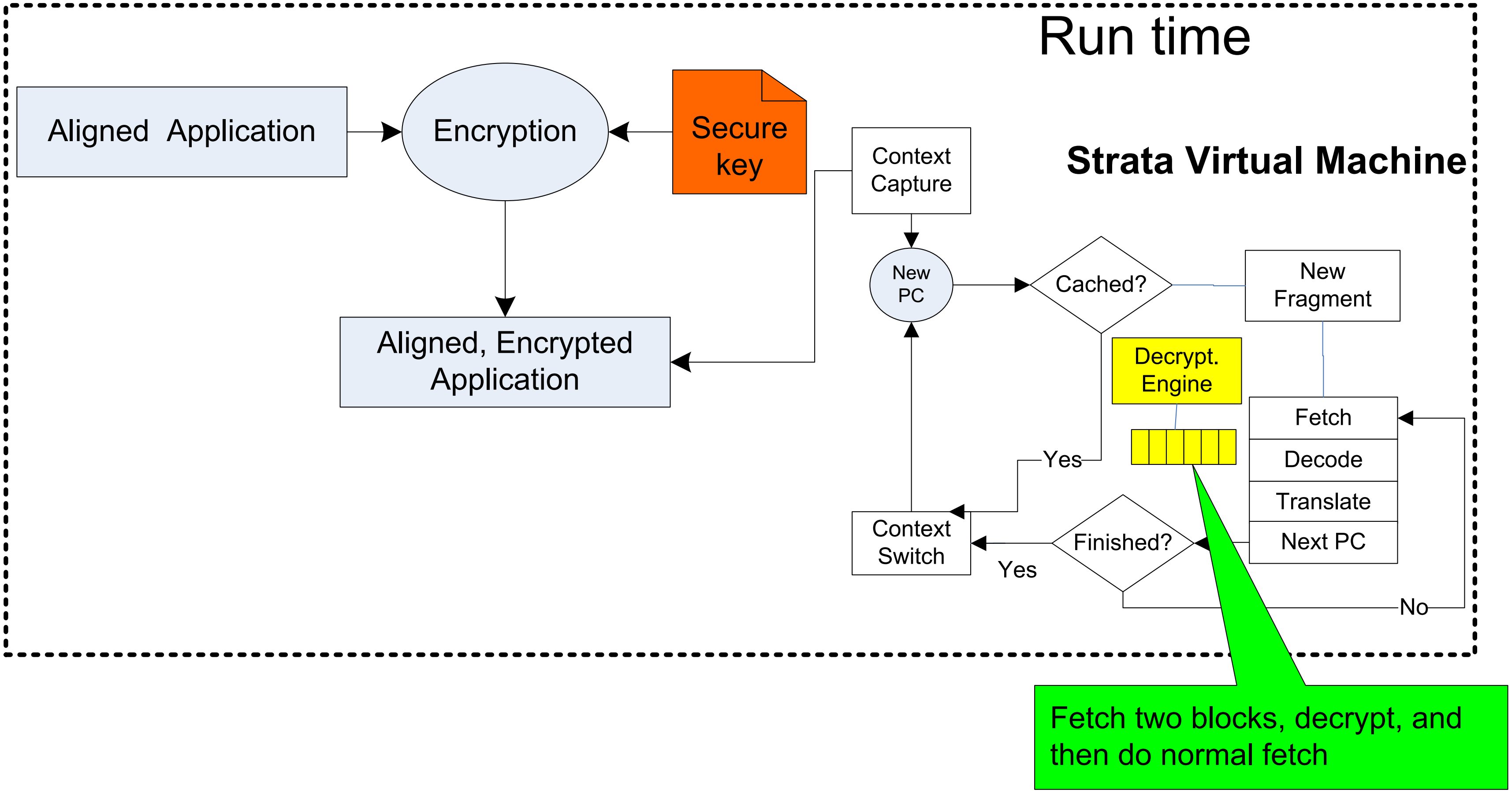
```
F:    inst1
      inst2
      inst3
      bne      L1
      inst5
L1:   inst6
      ...
```

Address		
0x40	3B	inst1
0x41	23	
0x42	F4	
0x43	71	
0x44	3A	inst2
0x45	21	
0x46	25	inst3
0x47	DB	
0x48	3C	
0x49	bne	inst4
0x4A	4E	
0x4B	49	inst5
0x4C	57	
0x4D	7E	
0x4E	33	inst6
0x4F	67	
0x50	...	

Address		
0x40	3B	inst1
0x41	23	
0x42	F4	
0x43	71	
0x44	3A	inst2
0x45	21	
0x46	25	inst3
0x47	DB	
0x48	3C	
0x49	bne	inst4
0x4A	50	
0x4B	49	inst5
0x4C	57	
0x4D	7E	
0x4E	nop	
0x4F	nop	
0x50	33	inst6
	67	
	...	

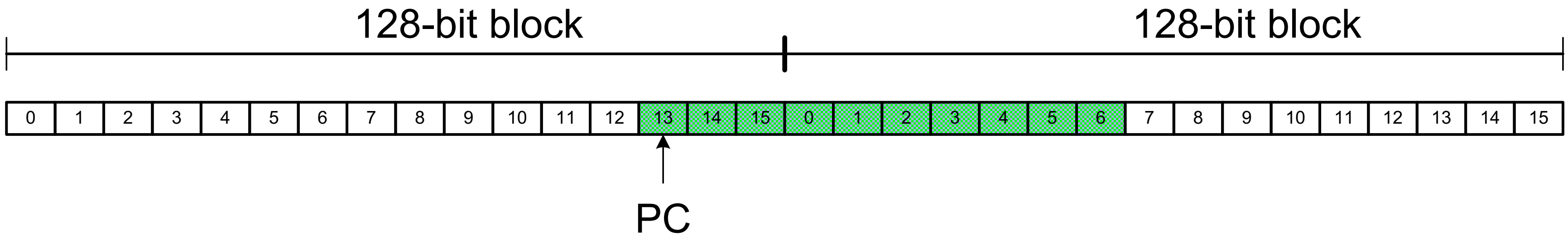
Address	
0x40	2A
0x41	37
0x42	FB
0x43	23
0x44	8C
0x45	67
0x46	27
0x47	54
0x48	BF
0x49	99
0x4A	12
0x4B	7E
0x4C	57
0x4D	3A
0x4E	5F
0x4F	91
0x50	33
	67
	...

Strata Modification



Fetch two blocks, decrypt, and then do normal fetch

Decoding Buffer Detail



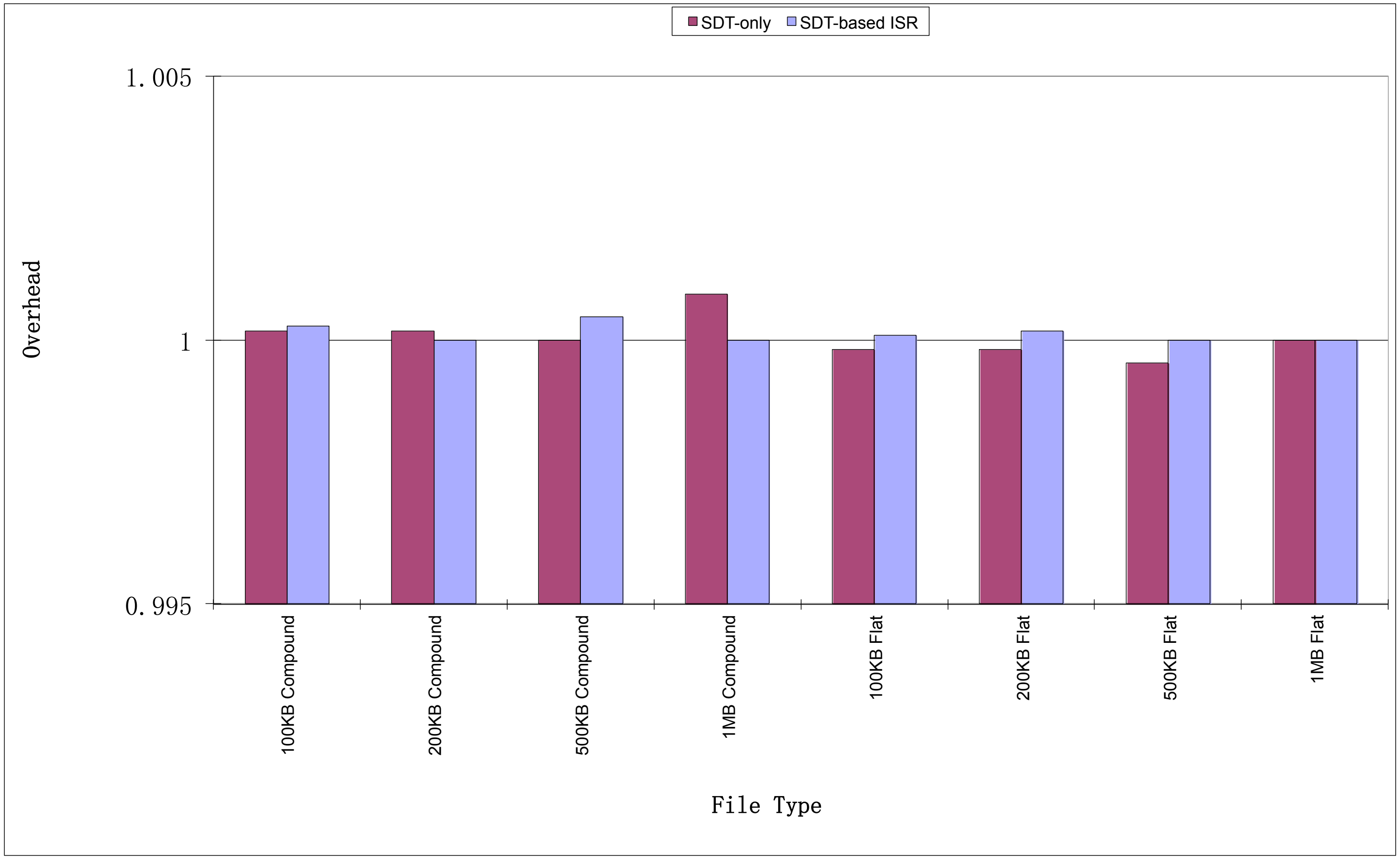
Creating Different Machines

- Software protection benefits
  - Encryption makes static analysis more difficult
  - Code must be executed to recover original code
- Software security benefits
  - Blocks ***all*** code injection attacks regardless of how the code was introduced
  - Low-overhead especially for server applications where translation overhead can be amortized over the lifetime of the server

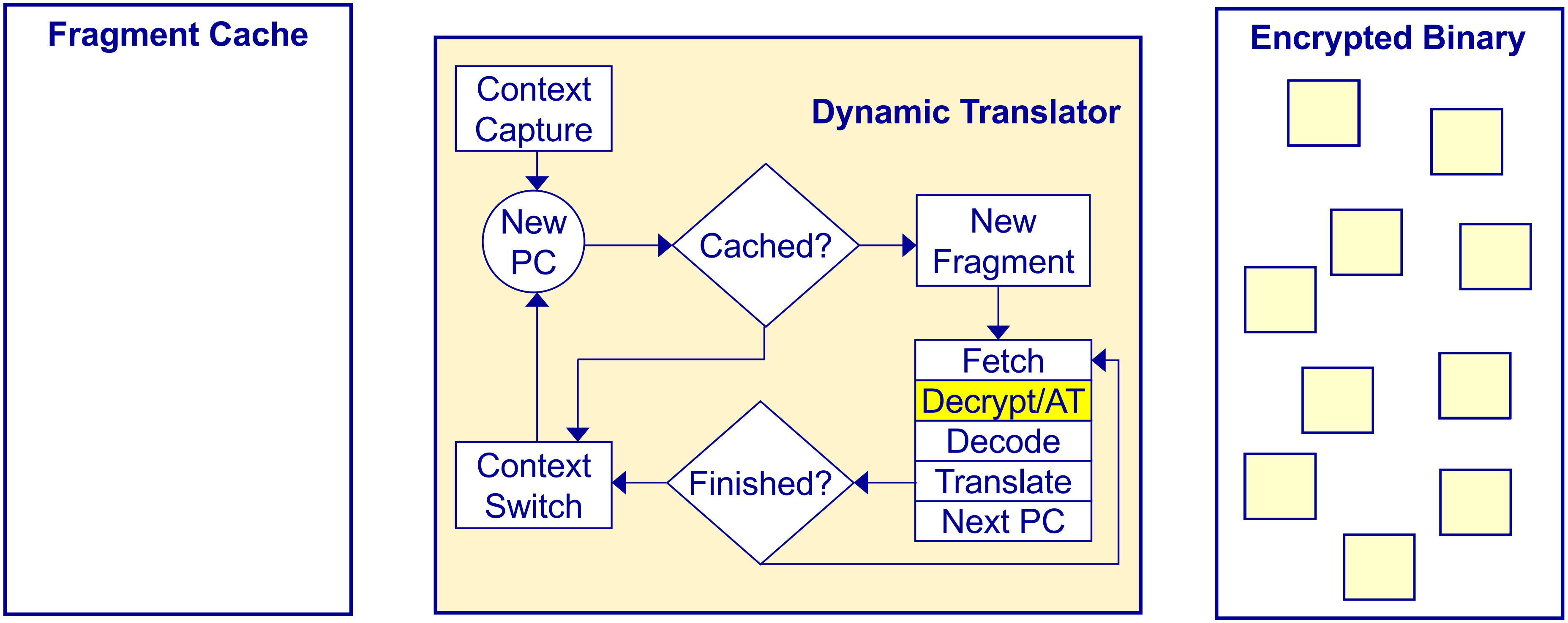
One last detail

- Not very satisfactory to just have code crash
- Diablo computes a message authentication code (MAC) over each block of 128 bits and inserts into instruction stream
- Strata checks and halts if MAC is not correct, otherwise it is removed before instructions are placed in the fragment cache

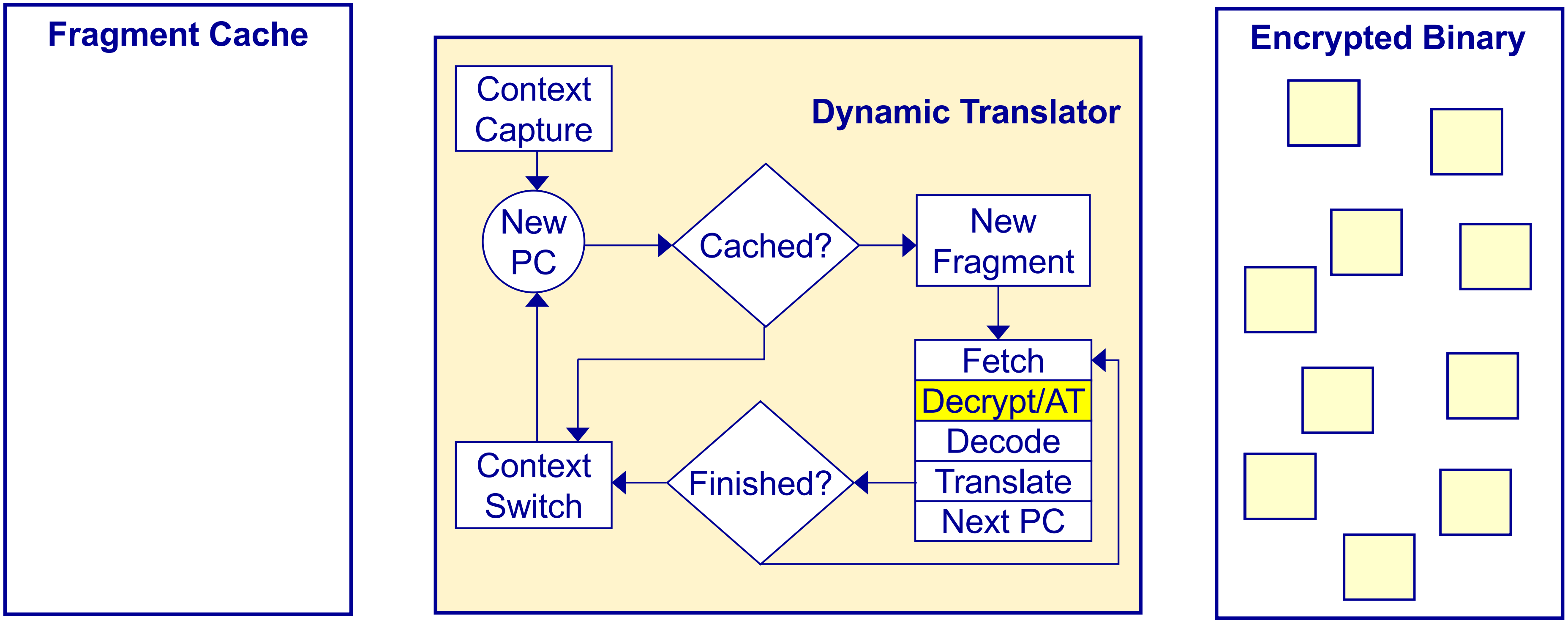
Performance of Apache



Dynamic Obfuscation



Dynamic Obfuscation



To delve further

W. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, J. W. Davidson: Zipr: Efficient Static Rewriting for Binaries. Proceedings of 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017), Denver, Colorado, June 2017.

W. Hawkins, J. D. Hiser, J. W. Davidson. Dynamic Canary Randomization for Improved Software Security. Proceedings of the 11th Annual Cyber and Information Security Research Conference, Oak Ridge, TN, April 2016, pp. 9:1–9:7.

D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong. Security through Diversity: Leveraging Virtual Machine Technology. *IEEE Security & Privacy*, 7(1), January/February 2009, pp. 26–33.

B. D. Rodes, A. Nguyen-Tuong, J. D. Hiser, J. C. Knight, M. Co, and J. W. Davidson. Defense Against Stack- Based Attacks Using Speculative Stack Layout Transformation. *Proceedings of International Conference on Run- time Verification*, Istanbul, Turkey, September 2012, pp. 308–313.

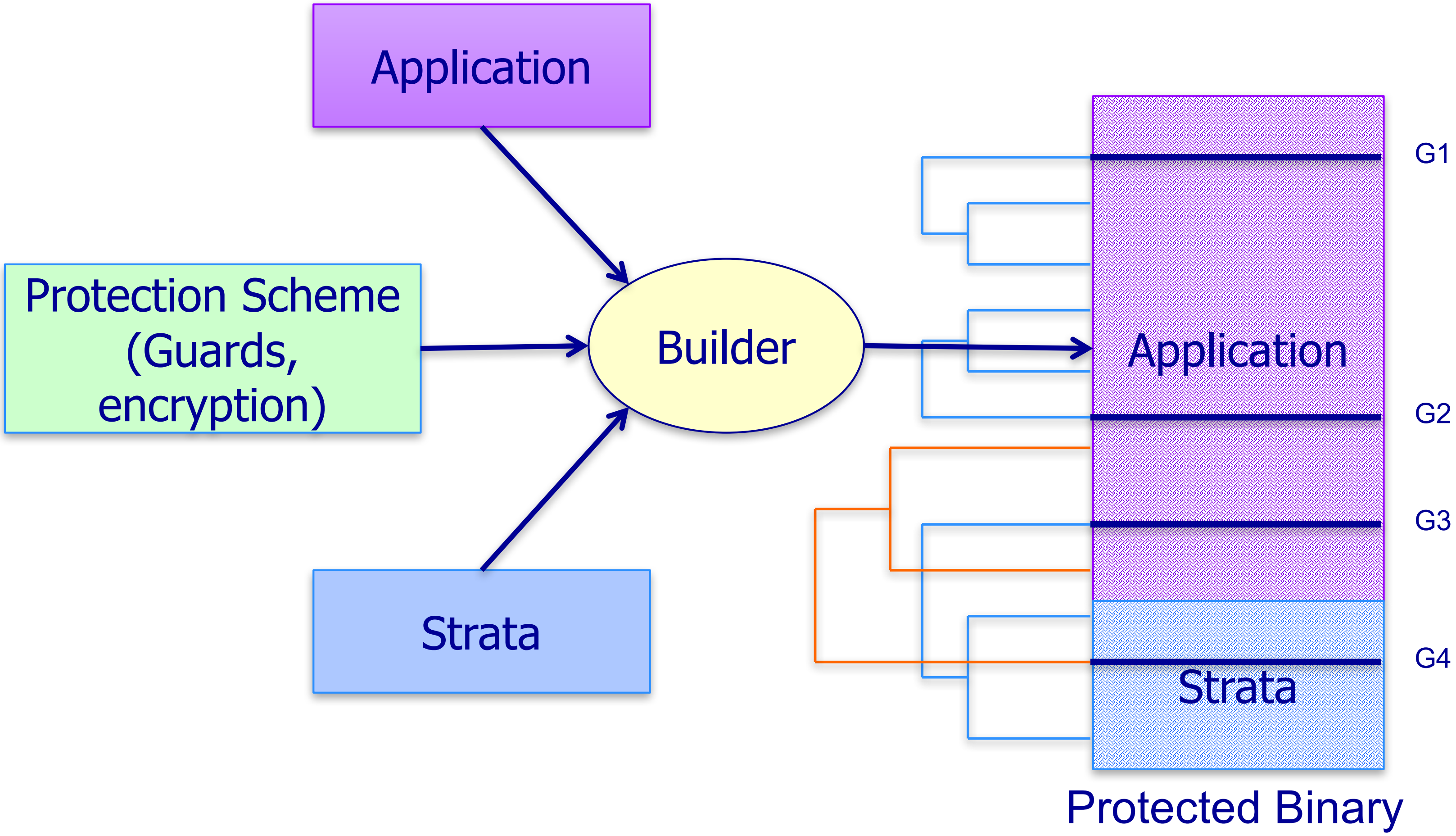
J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where did my gadgets go? *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, San Francsico, May 2012, pp. 571–585.

Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson. Security through Redundant Data Diversity, *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Anchorage, AK, June 2008, pp. 87–196.

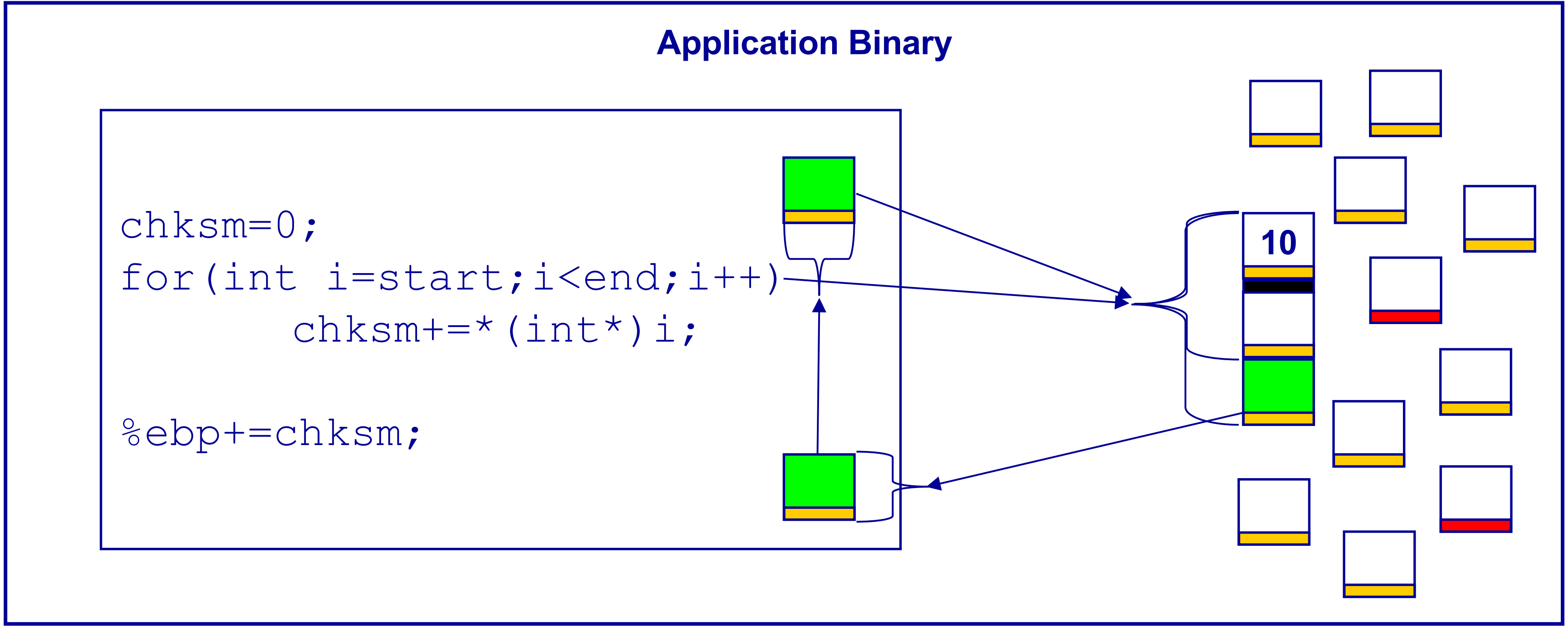
B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, and J. Knight, A. Nguyen-Tuong, J. D. Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. *Proceedings of the 15th USENIX Security Symposium*, Vancouver, B. C., Canada, August, 2006, pp. 105–120.

W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong and J. Hill. Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation. Proceedings of the 2006 International Conference on Virtual Execution Environments, Ottawa, CA, June 2006, pp. 2–12.

Protection Architecture



Guards Example



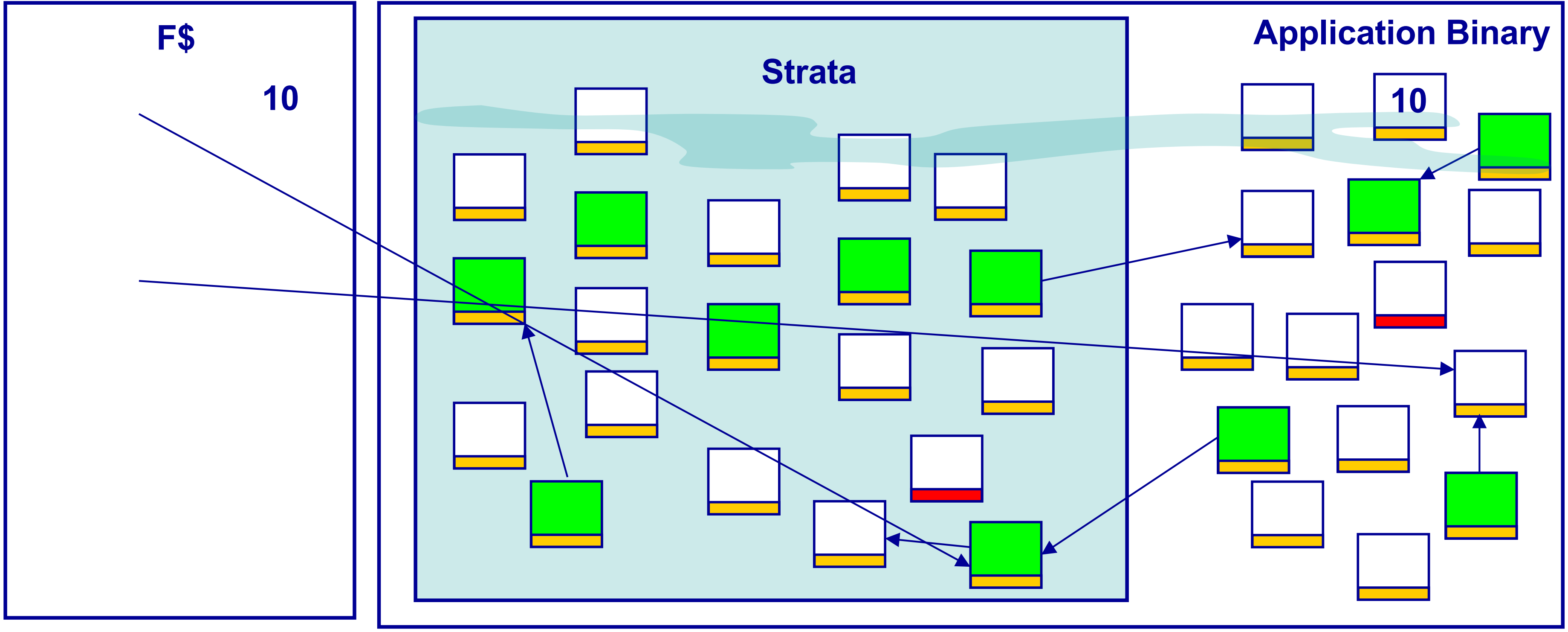
```
chksm=0;
for(int i=start;i<end;i++)
        chksm+=*(int*)i;

%ebp+=chksm;
```

▬ EXPECTED_CHECKSUM

- Advantages
  - Provides circular protection
  - Reasonable overhead

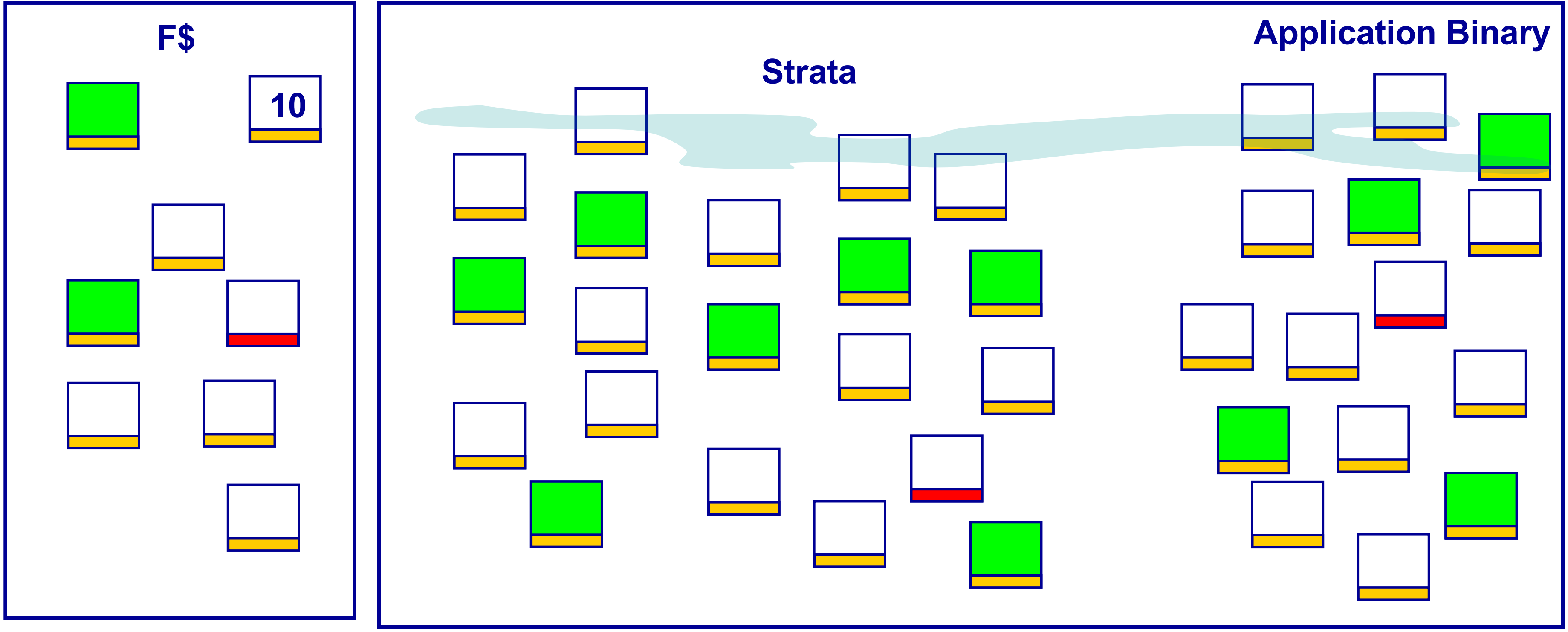
- Disadvantages
  - Applied once at link time
  - Execution of guard may reveal its location

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.

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

Resistance to Reverse Engineering

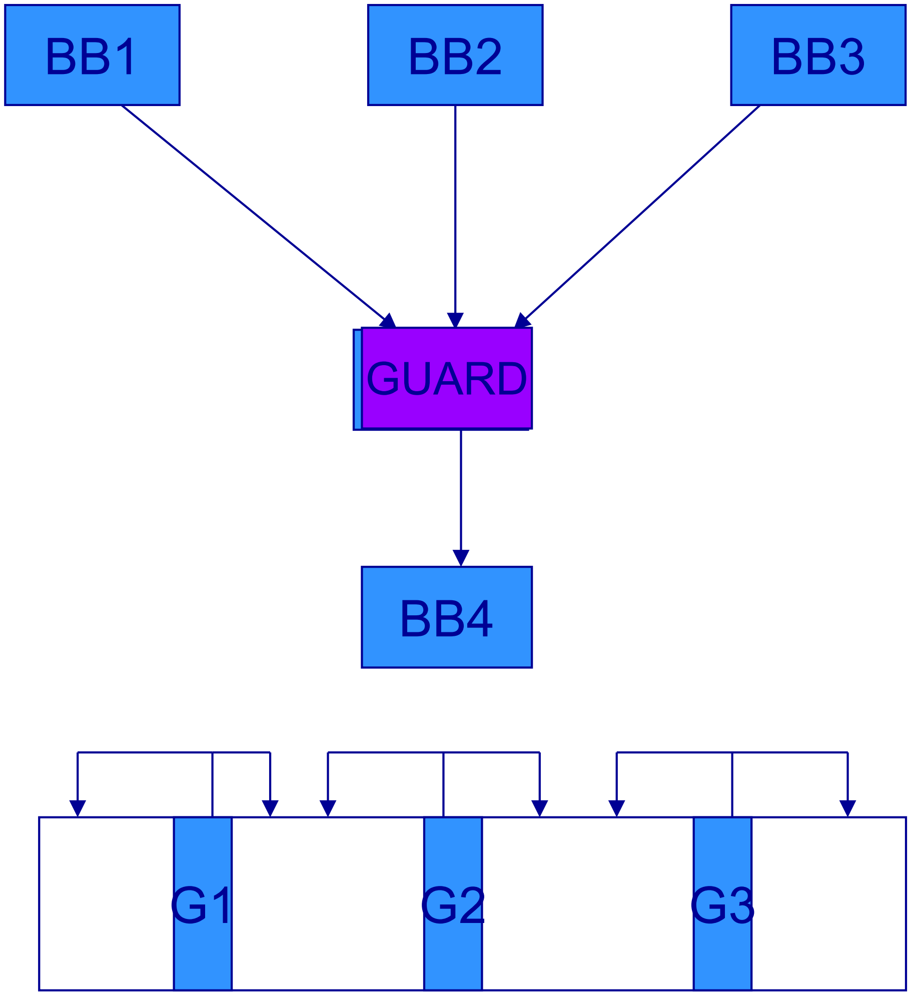
- Problem: Code buildup in software cache
  - The adversary could obtain code from the code cache.
- Solution: Flush code cache at periodic intervals (Temporal polymorphism)
  - Prevents adversary from obtaining sizable chunks of application code
  - Allied with randomized block sizes, ensures constant shifting of application code, making it hard for the adversary to locate and analyze

Triggering Flush

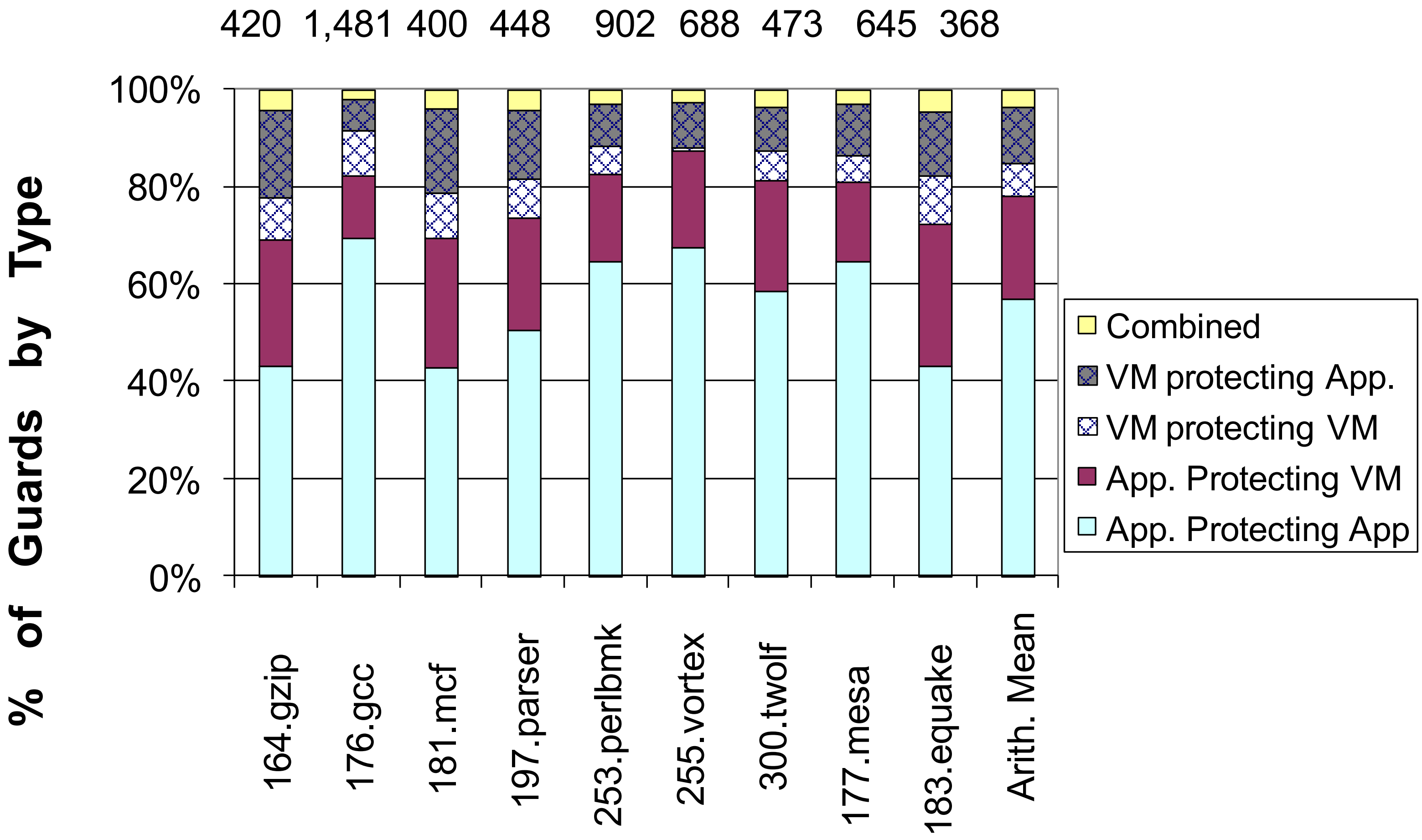
- A technique is required to flush the cache that is fairly periodic but stealthy.
  - Use signal triggered by OS.
    - In the threat model, the OS is assumed to be vulnerable to compromise.
  - Use a property inherent to the application itself (e.g., instruction counting)
    - Profile application and count particular instruction.
    - During production run, flush code cache based on some function of this count.

Guard Characteristics

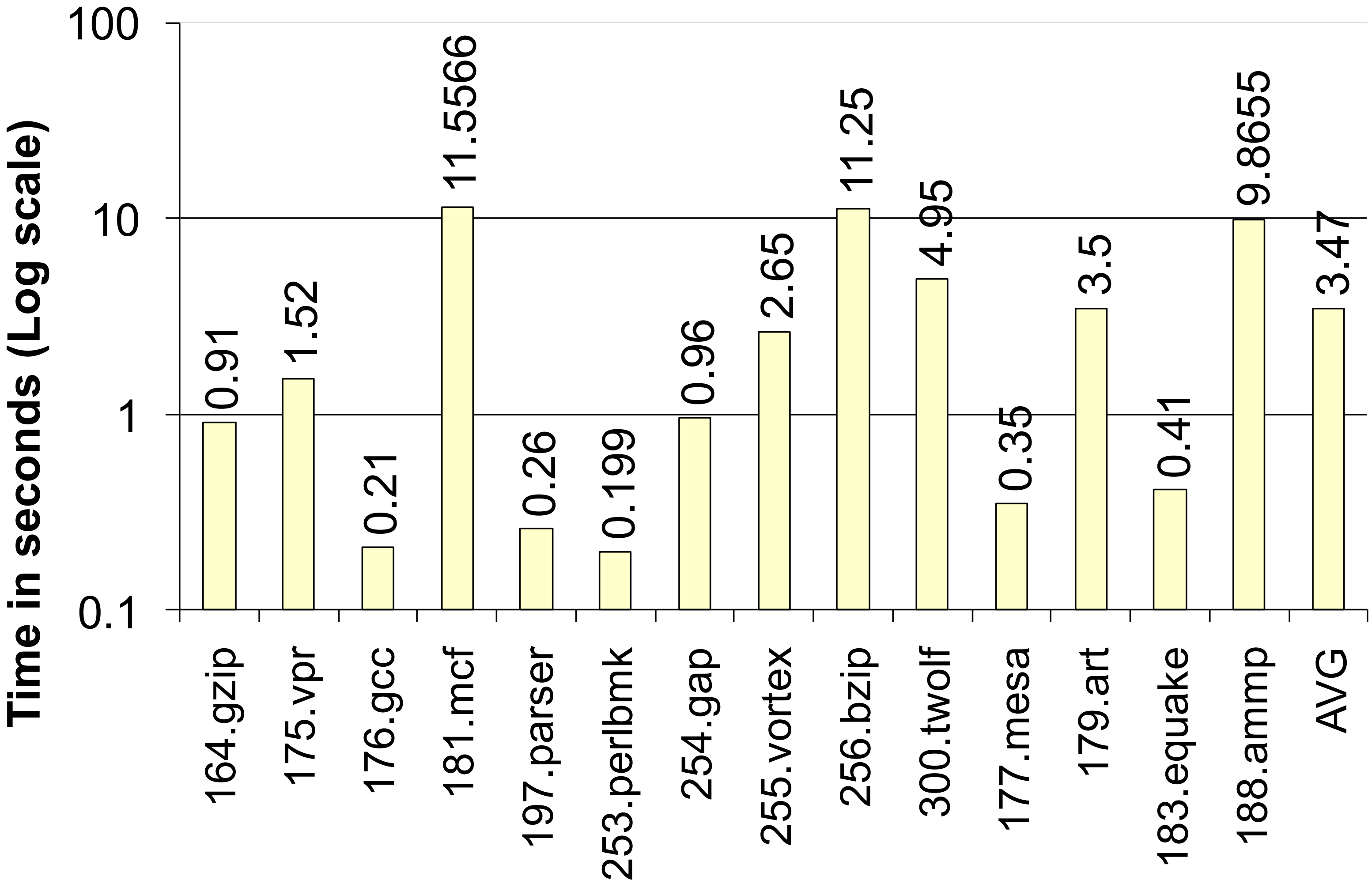
- Profiled approach
  - Run application on training inputs and get profiled count
  - Target BB chosen based on some function of profiled count.
- Range Selection
  - Guard range is selected in a cyclic manner, with the actual width being randomized



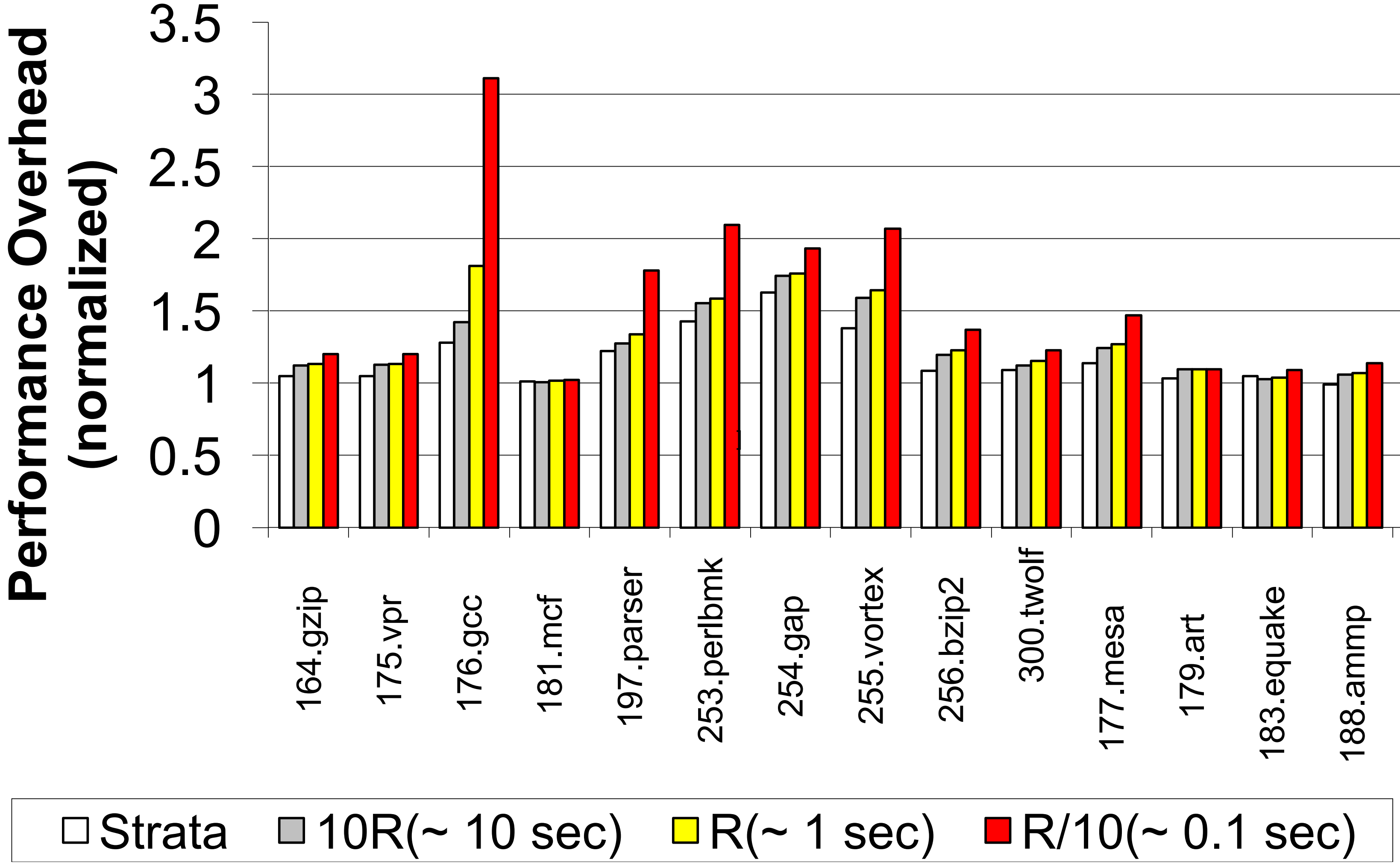
Distribution of Guards



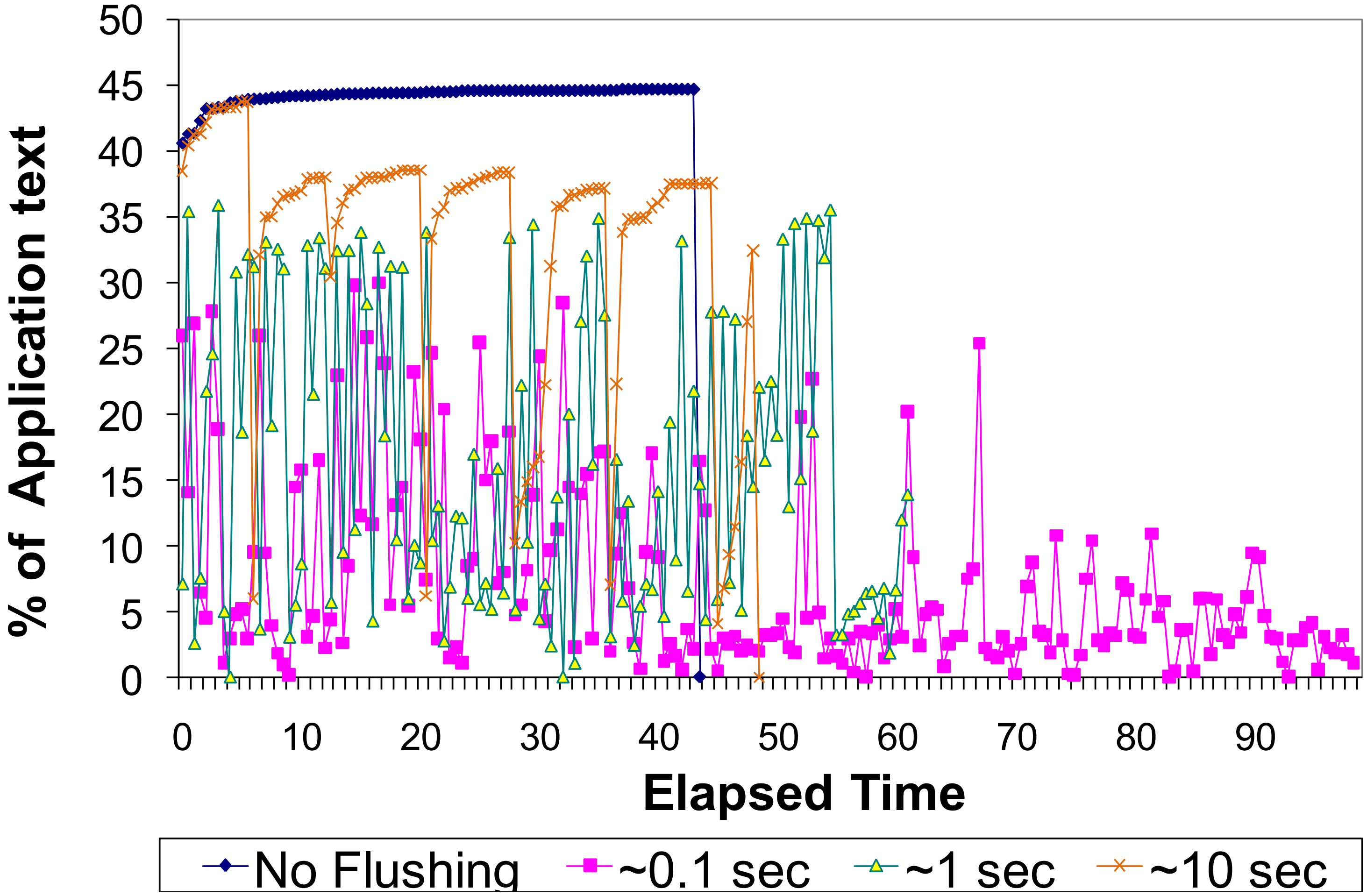
Effectiveness



Flushing Performance

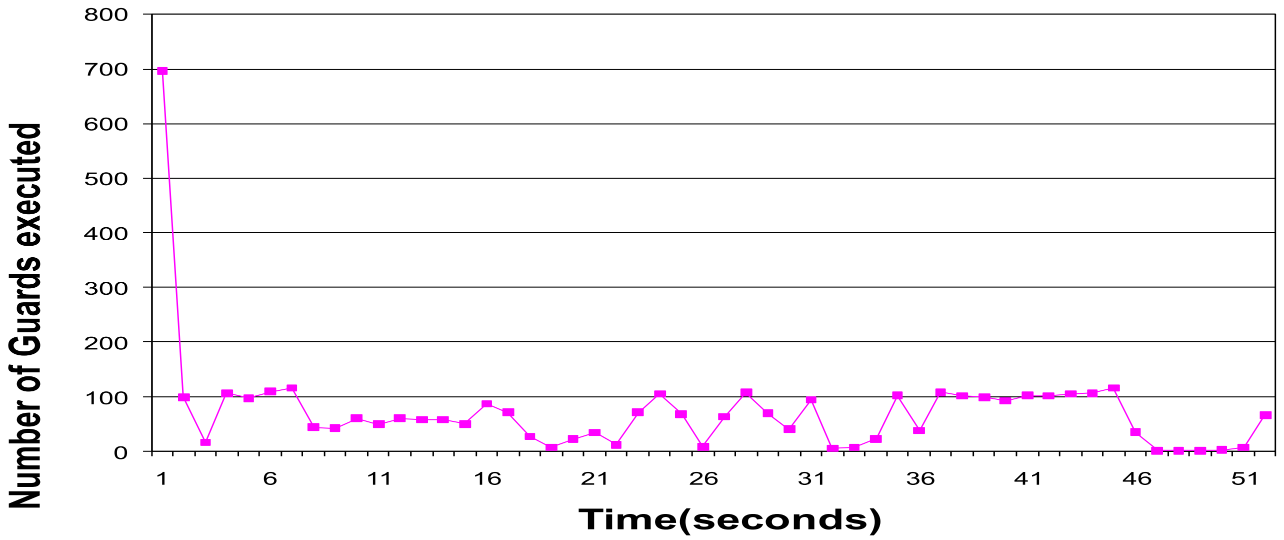


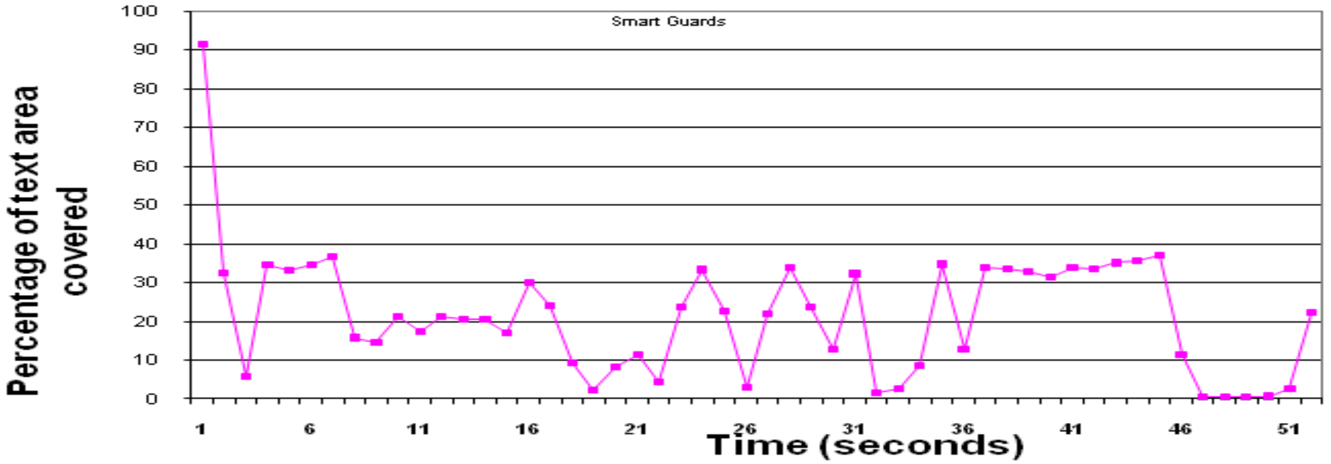
Flushing



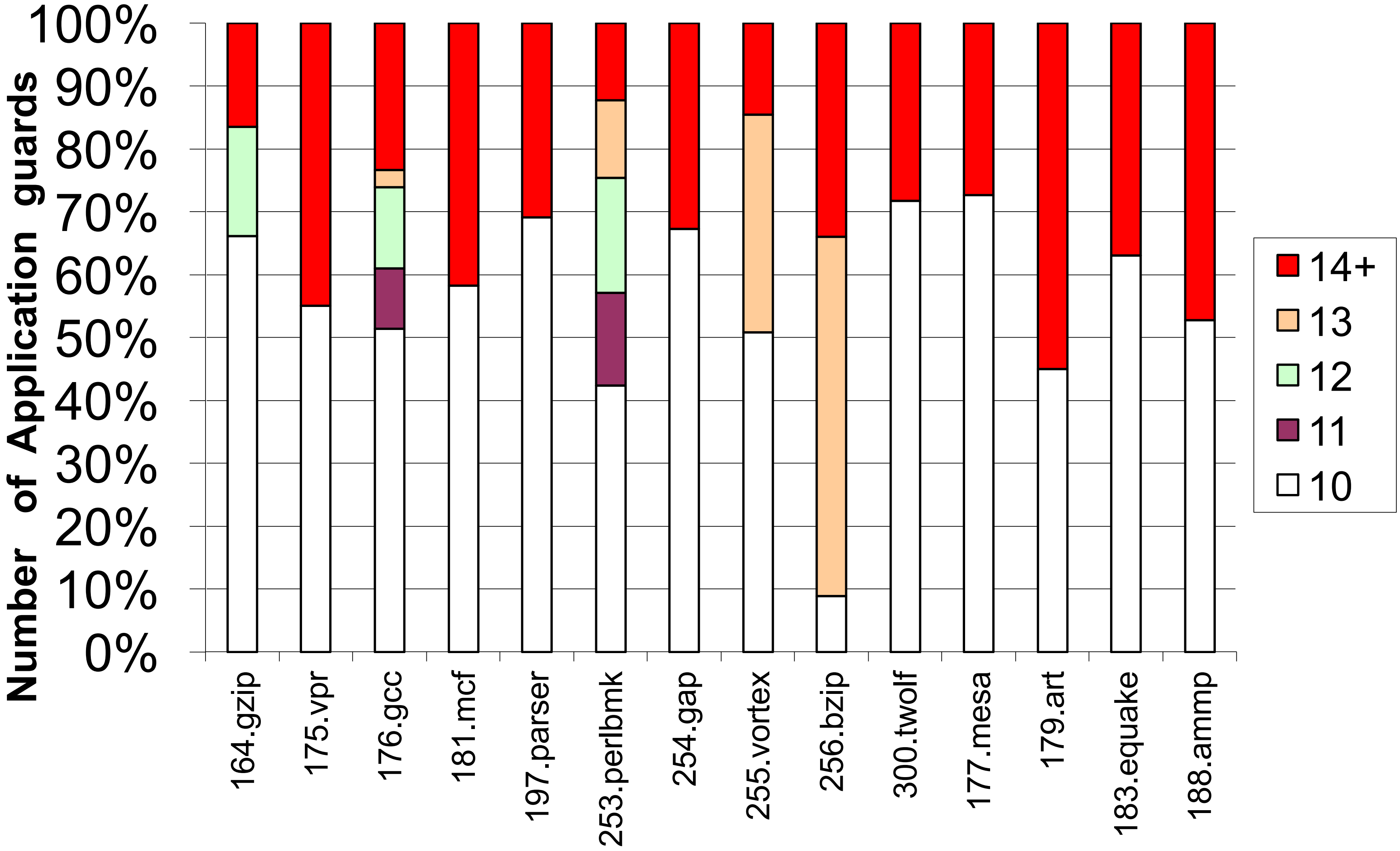
% of Application text
50
45
40
35
30
25
20
15
10
5
0
0
10
20
30
40
50
60
70
80
90
Elapsed Time
No Flushing
~0.1 sec
~1 sec
~10 sec

Runtime Coverage

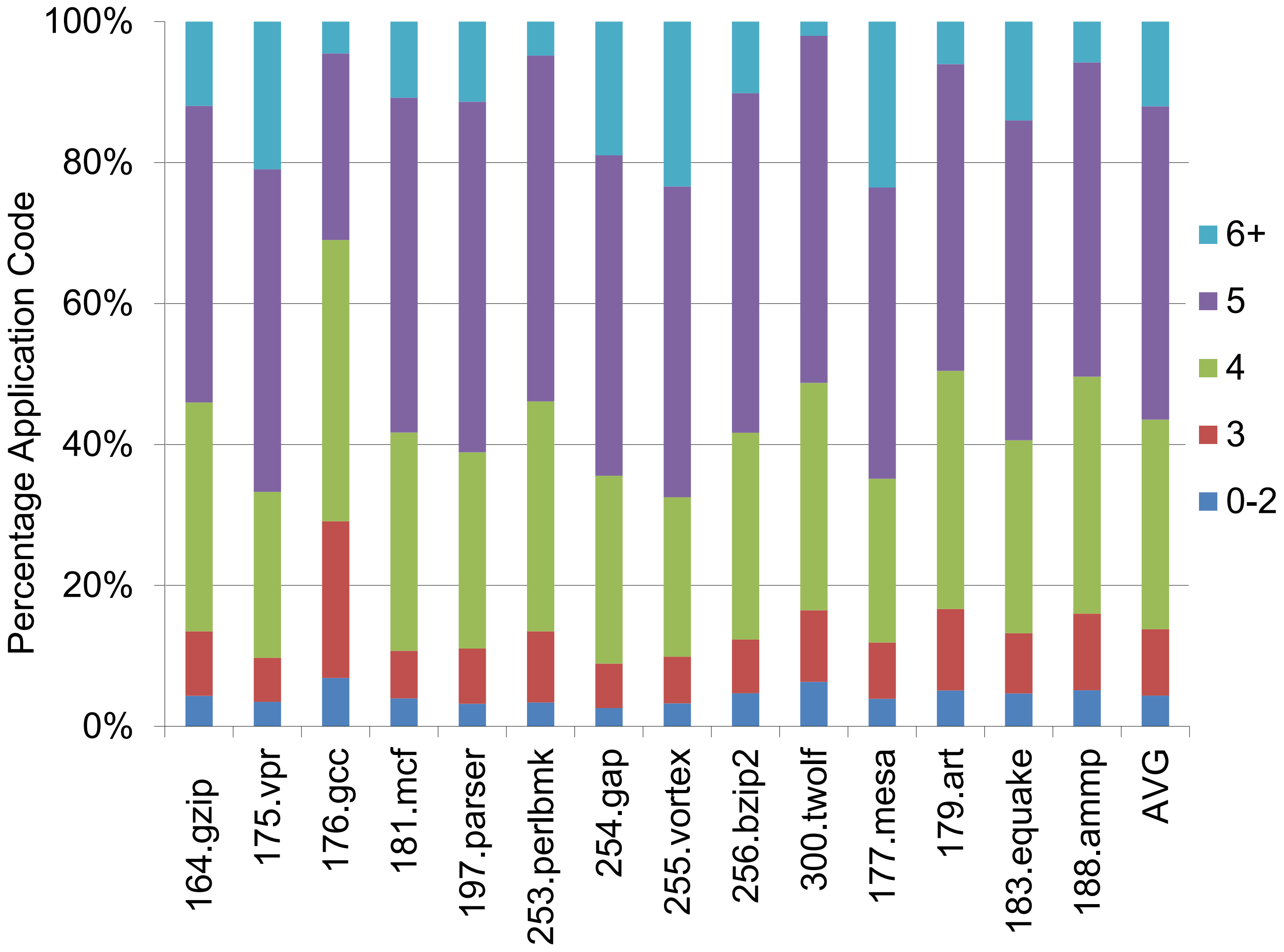




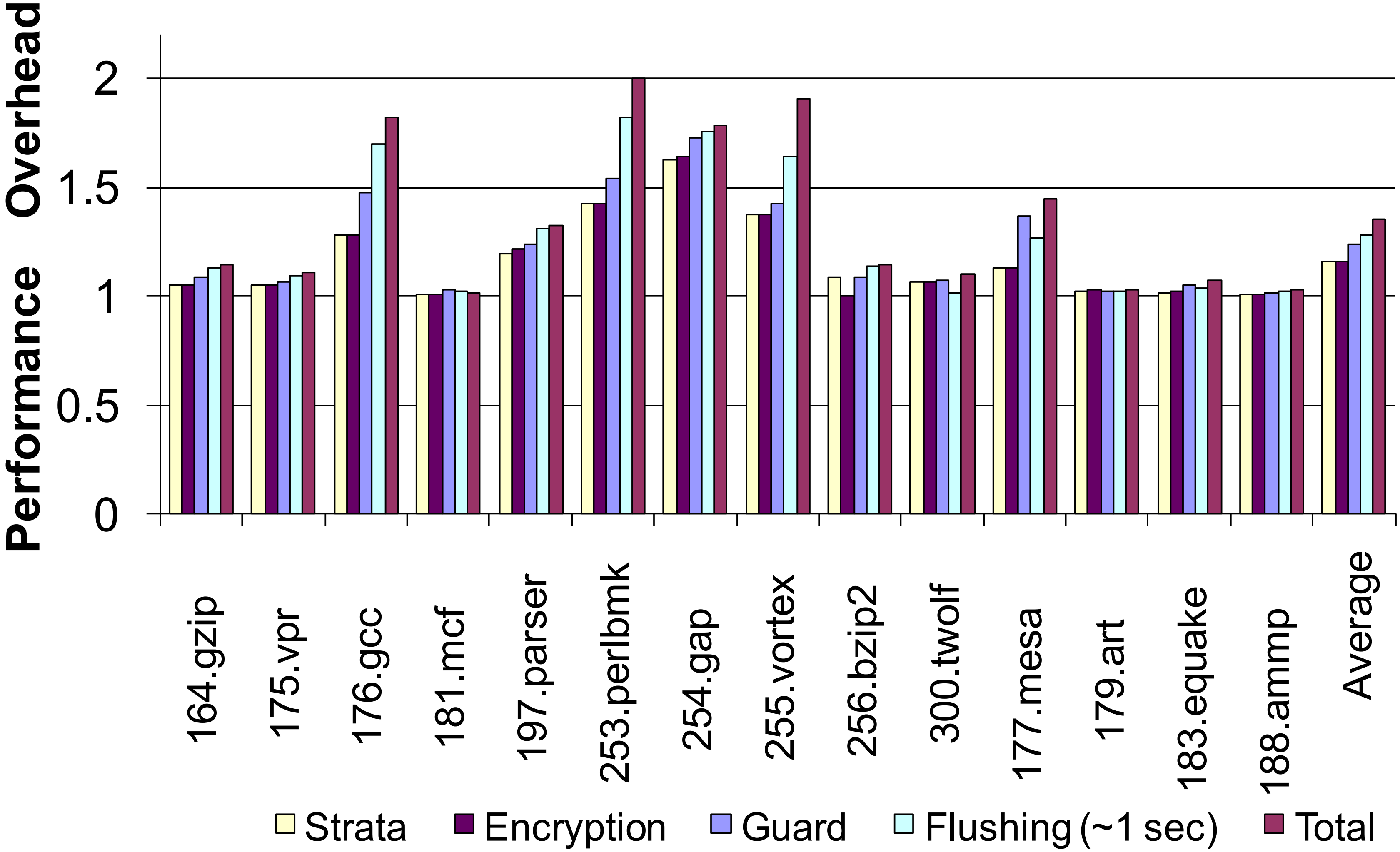
Code Shifting



Connectivity



Performance



Security Discussion

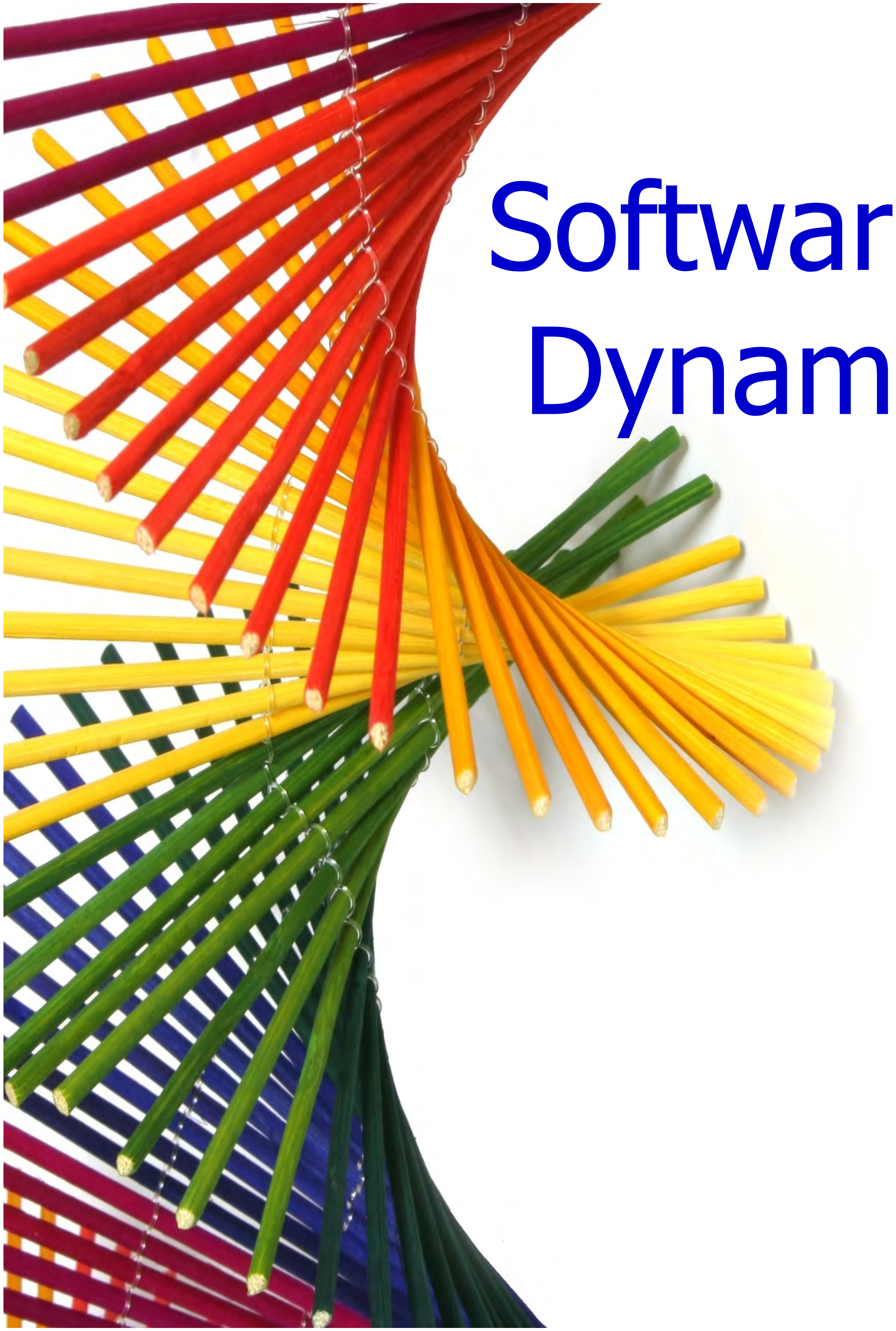
- Effectiveness against static and dynamic analysis
  - Encryption prevents static analysis
    - Knowledge of VM code irrelevant
  - Dynamic code relocation hampers dynamic analysis
    - Flushing
    - Randomized basic-block size

Related Work

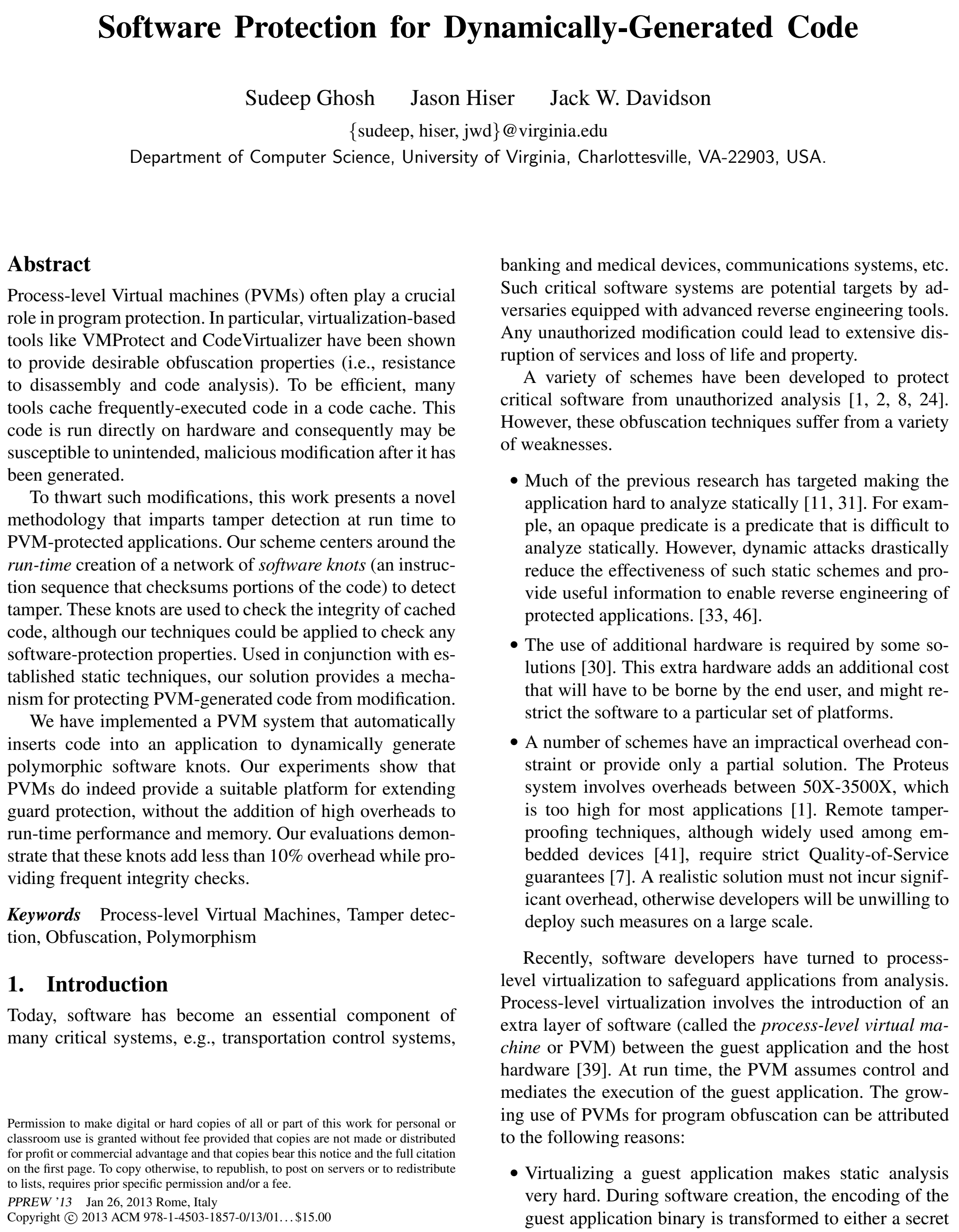
- Software Tamper Resistance
  - Aucsmith, D.: Tamper resistant software: An implementation.
  - Chang, H., Atallah, M.: Protecting software code by guards.
- Code Encryption
  - Cappaert, J., Preneel, B., Anckaert, B., Madou, M., Bosschere, K.D.: Towards tamper resistant code encryption: Practice and experience
- Remote Tamper-proofing
  - Collberg, C., Nagra, J., Snavely, W.: bianlian: Remote tamper-resistance with continuous replacement.
- VM-based approaches
  - Anckaert, B., Jakubowski, M., Venkatesan, R.: Proteus: virtualization for diversified tamper-resistance.

Summary

- Process-level virtualization provides a efficient platform for software protection. virtualization.
- Provides
  - Dynamic code obfuscation.
  - Granularity of decryption is much finer than previous work
  - Increased resistance against OS-based attacks on guards
  - Periodic flushing ensures any successful modification is temporary.



Software Protection For Dynamically-generated Code



Software Protection for Dynamically-Generated Code

Sudeep Ghosh Jason Hiser Jack W. Davidson
{sudeep, hiser, jwd}@virginia.edu
Department of Computer Science, University of Virginia, Charlottesville, VA-22903, USA.

Abstract

Process-level Virtual machines (PVMs) often play a crucial role in program protection. In particular, virtualization-based tools like VMProtect and CodeVirtualizer have been shown to provide desirable obfuscation properties (i.e., resistance to disassembly and code analysis). To be efficient, many tools cache frequently-executed code in a code cache. This code is run directly on hardware and consequently may be susceptible to unintended, malicious modification after it has been generated.

To thwart such modifications, this work presents a novel methodology that imparts tamper detection at run time to PVM-protected applications. Our scheme centers around the *run-time* creation of a network of *software knots* (an instruction sequence that checksums portions of the code) to detect tamper. These knots are used to check the integrity of cached code, although our techniques could be applied to check any software-protection properties. Used in conjunction with established static techniques, our solution provides a mechanism for protecting PVM-generated code from modification.

We have implemented a PVM system that automatically inserts code into an application to dynamically generate polymorphic software knots. Our experiments show that PVMs do indeed provide a suitable platform for extending guard protection, without the addition of high overheads to run-time performance and memory. Our evaluations demonstrate that these knots add less than 10% overhead while providing frequent integrity checks.

Keywords Process-level Virtual Machines, Tamper detection, Obfuscation, Polymorphism

1. Introduction

Today, software has become an essential component of many critical systems, e.g., transportation control systems, banking and medical devices, communications systems, etc. Such critical software systems are potential targets by adversaries equipped with advanced reverse engineering tools. Any unauthorized modification could lead to extensive disruption of services and loss of life and property.

A variety of schemes have been developed to protect critical software from unauthorized analysis [1, 2, 8, 24]. However, these obfuscation techniques suffer from a variety of weaknesses.

- Much of the previous research has targeted making the application hard to analyze statically [11, 31]. For example, an opaque predicate is a predicate that is difficult to analyze statically. However, dynamic attacks drastically reduce the effectiveness of such static schemes and provide useful information to enable reverse engineering of protected applications. [33, 46].
- The use of additional hardware is required by some solutions [30]. This extra hardware adds an additional cost that will have to be borne by the end user, and might restrict the software to a particular set of platforms.
- A number of schemes have an impractical overhead constraint or provide only a partial solution. The Proteus system involves overheads between 50X-3500X, which is too high for most applications [1]. Remote tamper-proofing techniques, although widely used among embedded devices [41], require strict Quality-of-Service guarantees [7]. A realistic solution must not incur significant overhead, otherwise developers will be unwilling to deploy such measures on a large scale.

Recently, software developers have turned to process-level virtualization to safeguard applications from analysis. Process-level virtualization involves the introduction of an extra layer of software (called the *process-level virtual machine* or PVM) between the guest application and the host hardware [39]. At run time, the PVM assumes control and mediates the execution of the guest application. The growing use of PVMs for program obfuscation can be attributed to the following reasons:

- Virtualizing a guest application makes static analysis very hard. During software creation, the encoding of the guest application binary is transformed to either a secret

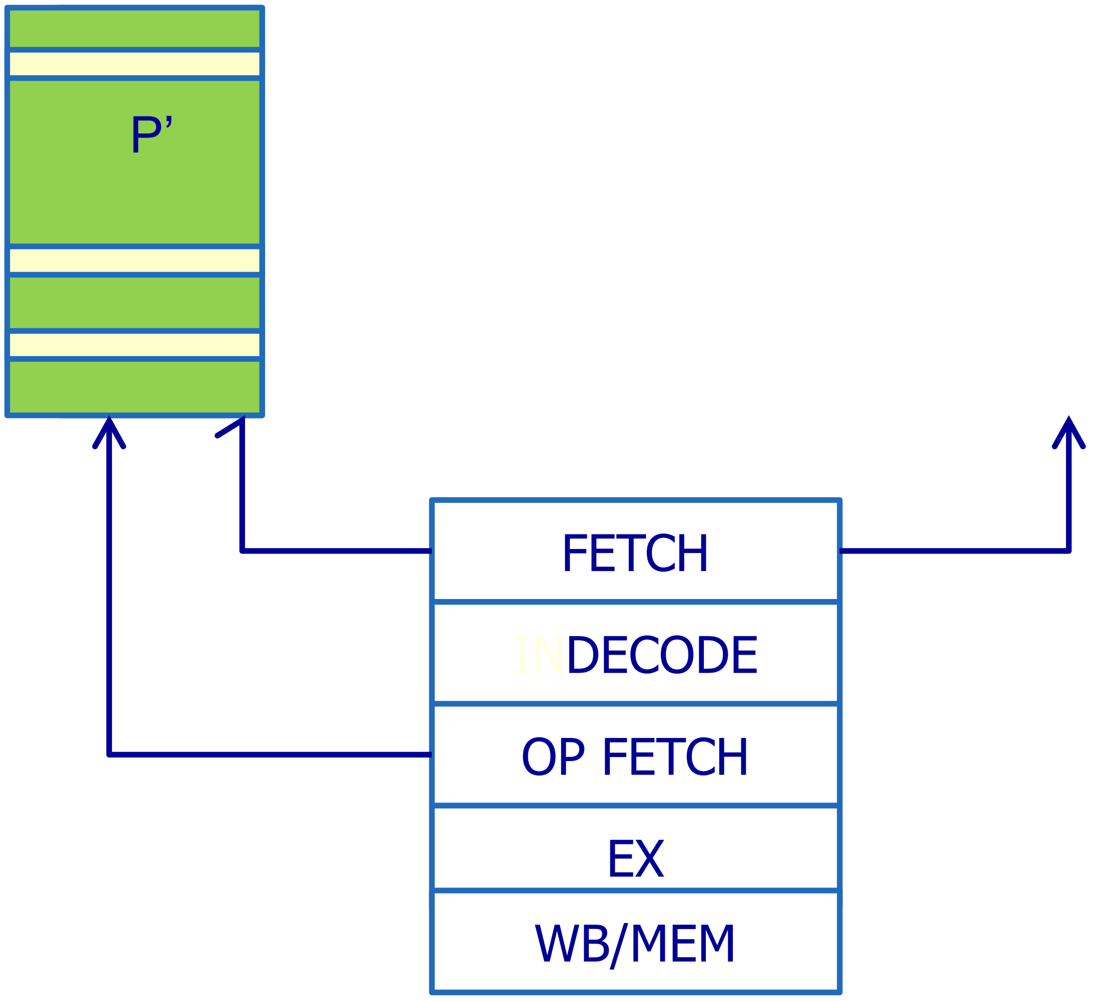
PPREW '13 Jan 26, 2013 Rome, Italy
Copyright © 2013 ACM 978-1-4503-1857-0/13/01…$15.00

Strengths and Weaknesses

- Strengths
  - Software-only solution
  - Thwarts static analysis
  - Relatively easy to integrate with applications
  - Low run-time overhead
  - Requires no resource guarantees, beyond application requirements
- Weaknesses
  - Dynamically generated code vulnerable to tamper
  - Subject to split memory attacks
  - Application and PVM not tightly bound

Split Memory Attack on Guards

ORIGINAL
APP



P’

FETCH

DECODE

OP FETCH

EX

WB/MEM

MODIFIED
BINARY

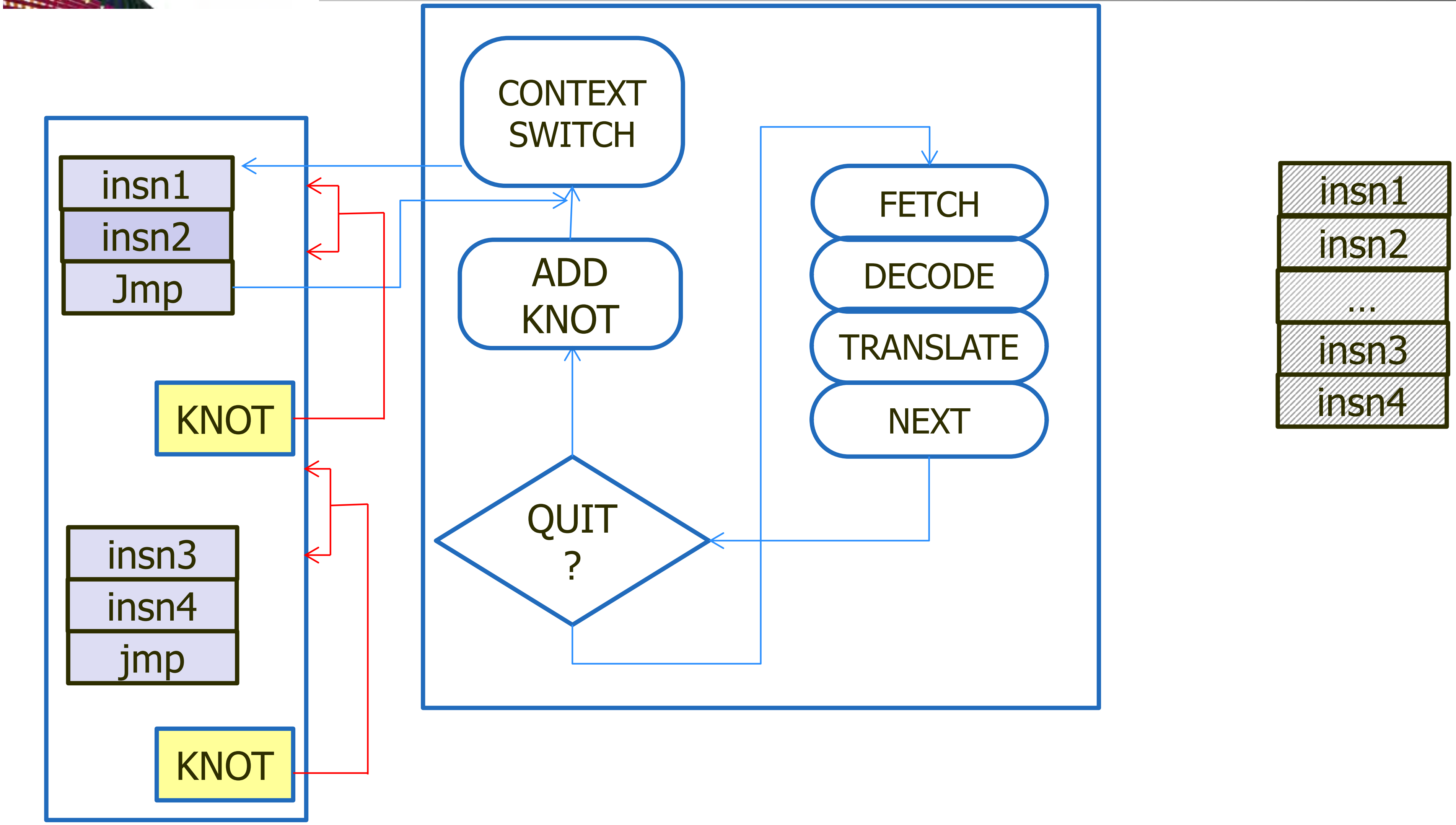
INSTRUCTION
EXECUTION

CHKSUM
CALCULATION

Knots

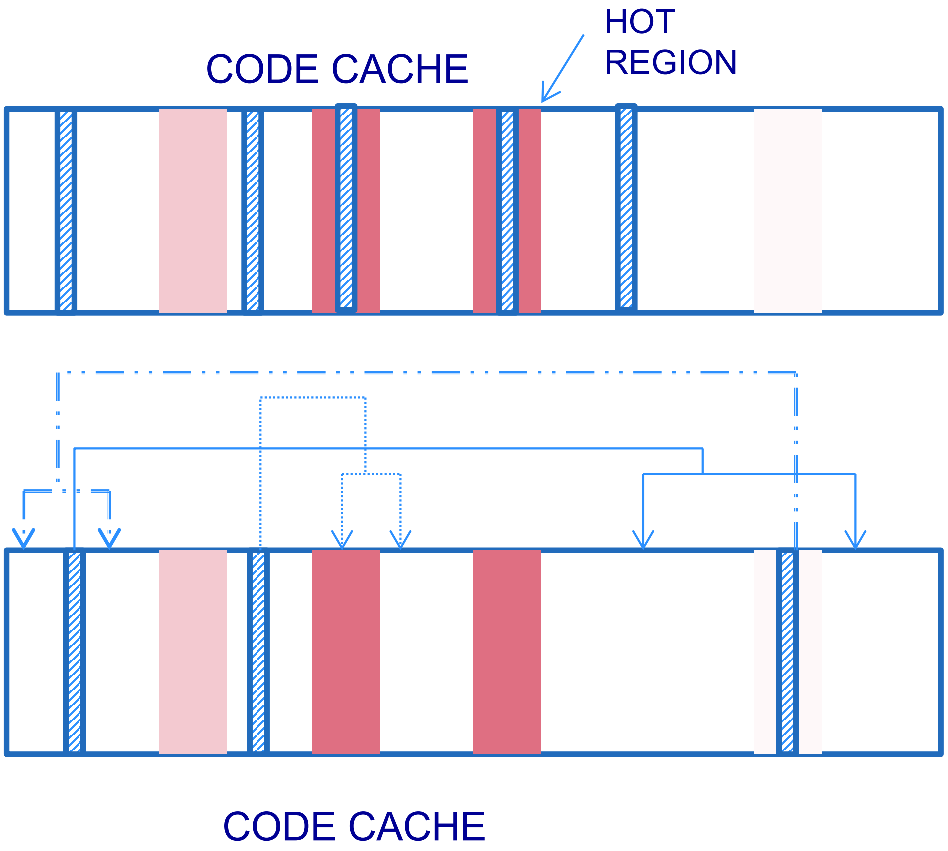
- Definition:
  - Dynamically-generated code that checksums a range of memory addresses
  - Based on static guards
    - H.Chang, and M.Atallah. *Protecting Software Code by Guards.* In the Proceedings of the ACM Workshop on Security and Privacy in DRM, 2000.
- Network of polymorphic guards protect code that is generated by the PVM at run time

Strata PVM With Polymorphic Knots



Knot Parameters

- Location
  - Random
  - Probabilistic
  - Predicated
- Network
  - Connectivity, K
  - Range Size



Instantiation Polymorphism

- Preamble
- Loop
- Checker
- Response

```
preamble :
  mov edx , -checksum
  mov eax , range_start


loop :
  cmp ebx , range_end
  jg checker
  add edx , dword [ ebx ]
  add eax , 4
  jmp loop
....
checker :
  cmp edx , 0
  je app_code
  jmp tamper_response
....

tamper_response :
```

```
preamble :
  push checksum
  pop ecx
  mov eax , range_start

loop :
  cmp eax , range_end
  jg checker
  xor ecx , dword [ eax ]
  lea eax , [ eax + 4]
  jmp loop
....
checker :
  jecxz app_code
  sub esp , 4
  mov [esp] , tamper_response
  ret
....
tamper_response :
...
```

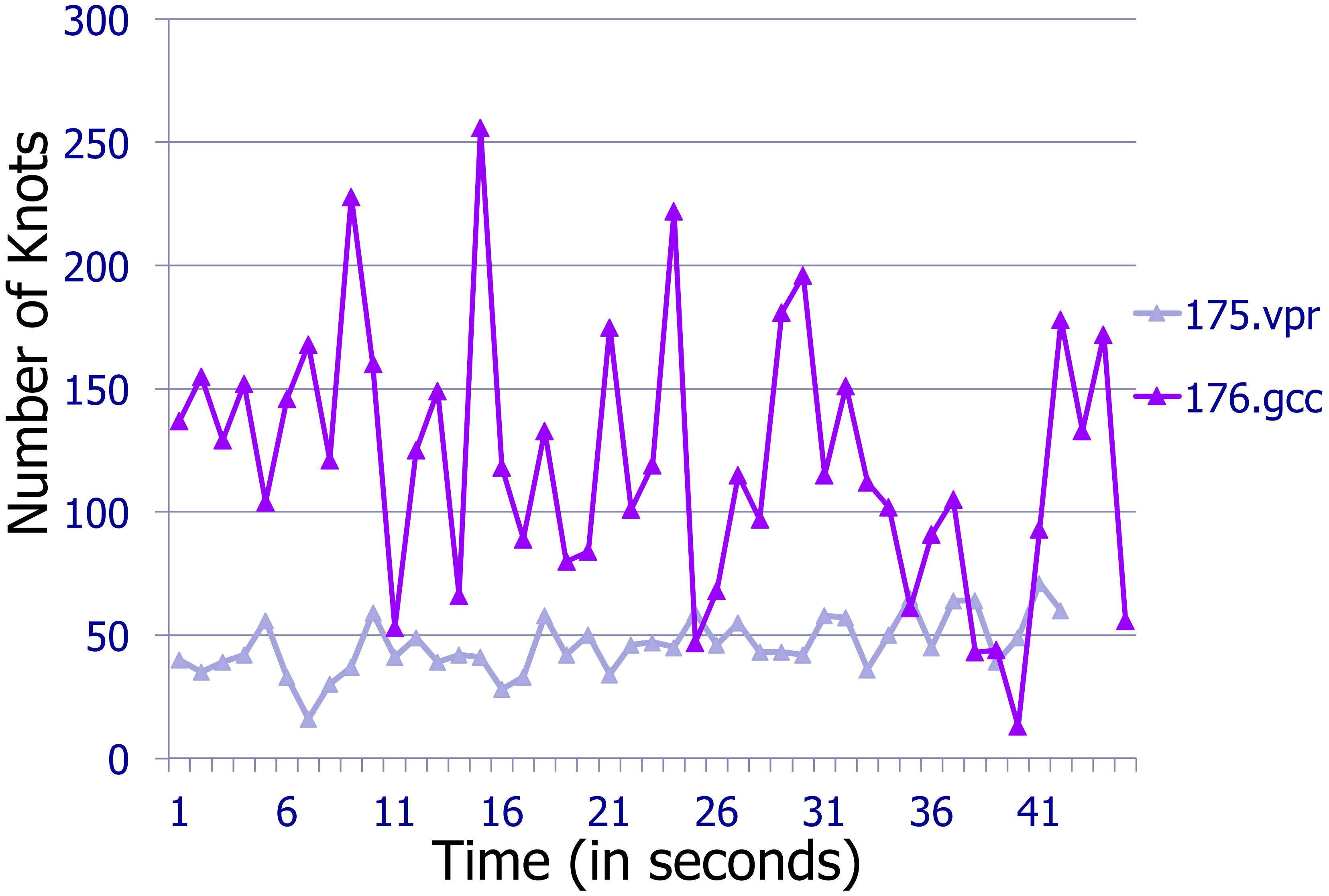
Temporal Polymorphism

- Periodic code cache flushing creates a dynamic execution environment
  - S. Ghosh, J. Hiser and J. Davidson*. A Secure and Robust Approach to Software Tamper Resistance*. In the Proceedings of the 12th International Conference on Information Hiding, 2010 (previously presented)
- Periodically create a new network of knots that cyclically reinforce each other

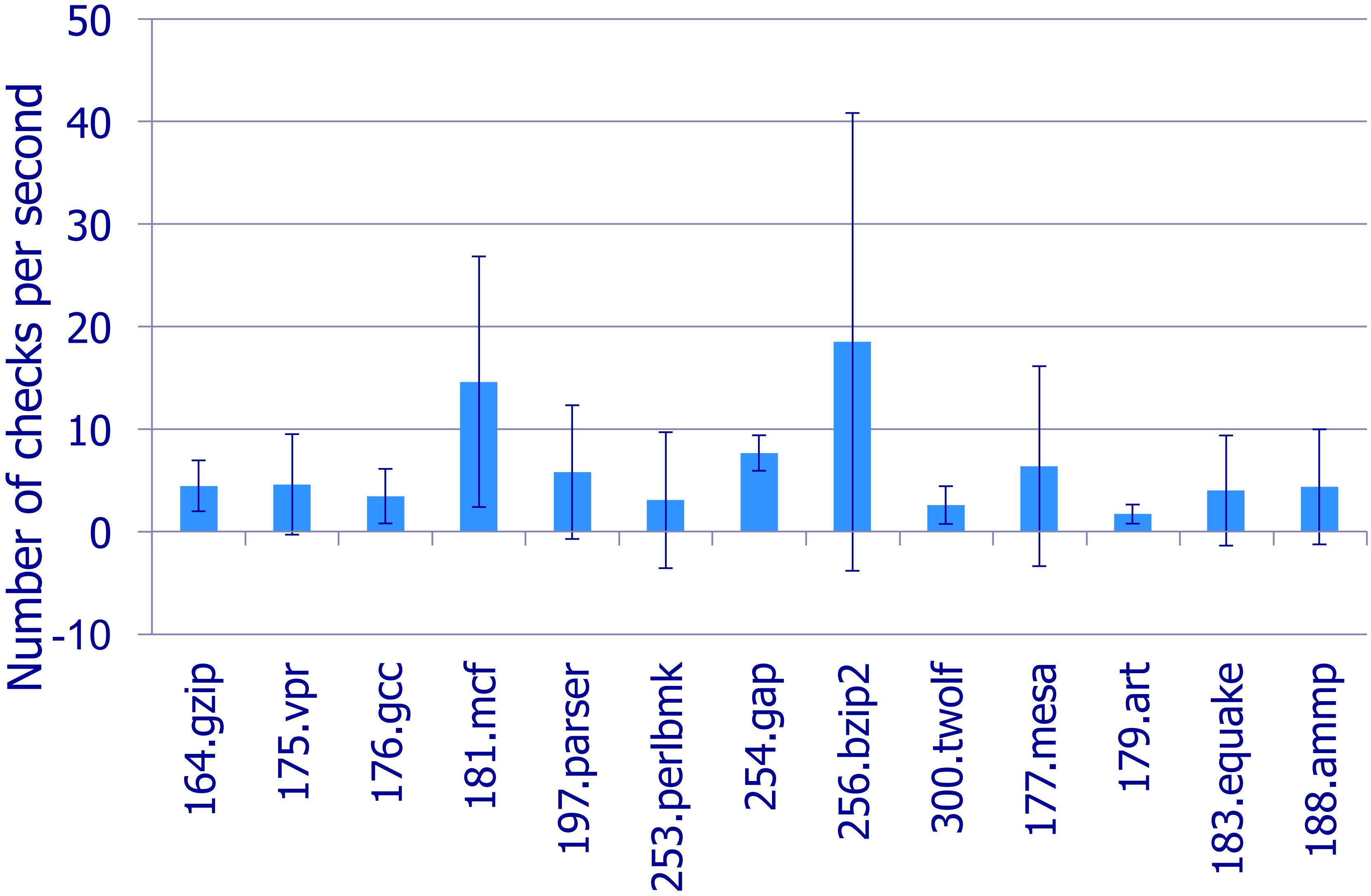
Evaluation

- 32-bit Intel x86 platform
- Ubuntu 10.04
- SPEC2000 C language benchmarks
- Knot insertion rate: 2%
- Connectivity *K*: 7

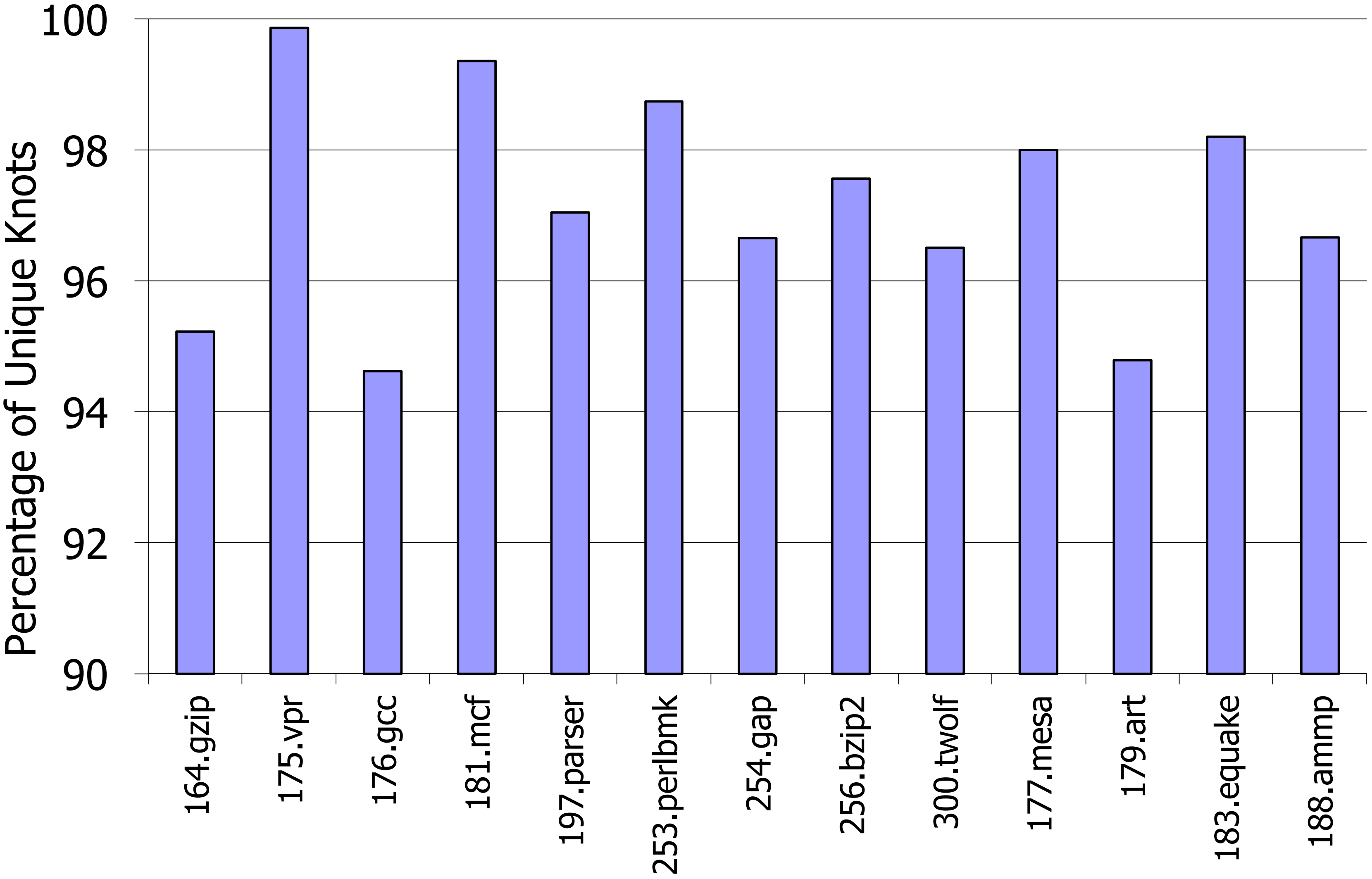
Knots invoked



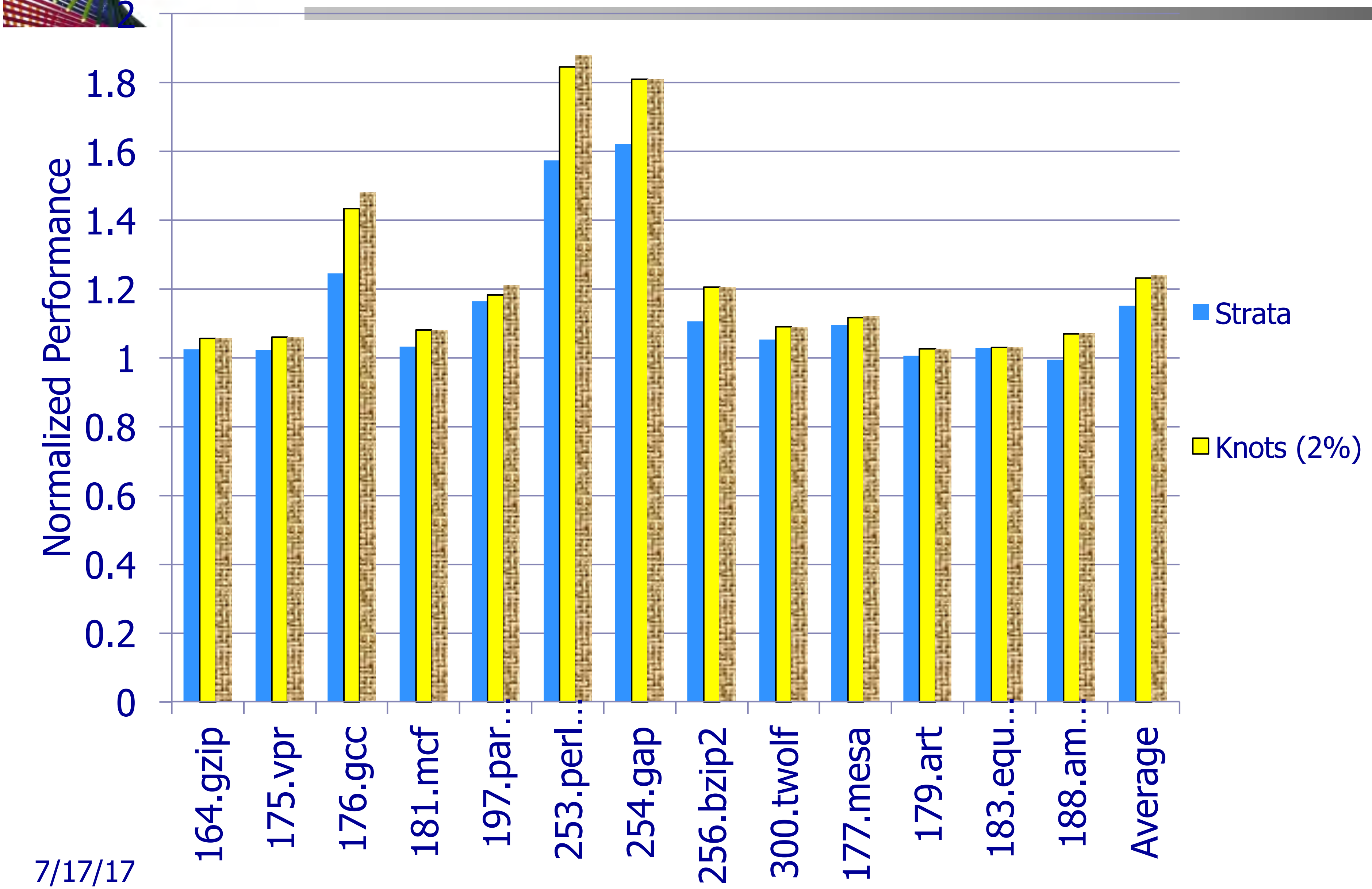
Average checks/second per byte



Unique Knots



Performance



Security Discussion

- Static in-place guards (described by Chang and Atallah) are susceptible to split-memory attacks, which is not the case for knots
  - G. Wurster, P.C. Van Oorschot, and A. Somayaji. *A Generic Attack on Checksumming-based Software Tamper Resistance.* In the Proceedings of the 2005 IEEE Symposium on Security and Privacy (Washington D.C., U.S.A, 2005)
- Knots are susceptible to replacement attacks
  - S. Ghosh, J.Hiser, and J. Davidson, *Replacement attacks against VM-protected Applications.* In the Proceedings of the 2012 ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments.
- Knots, like generic protection techniques, can be susceptible to information leakage
  - N. Lawson. *Side-channel attacks on Cryptographic Software.* IEEE Security and Privacy Volume 6, Nov. 2009.

Related Work

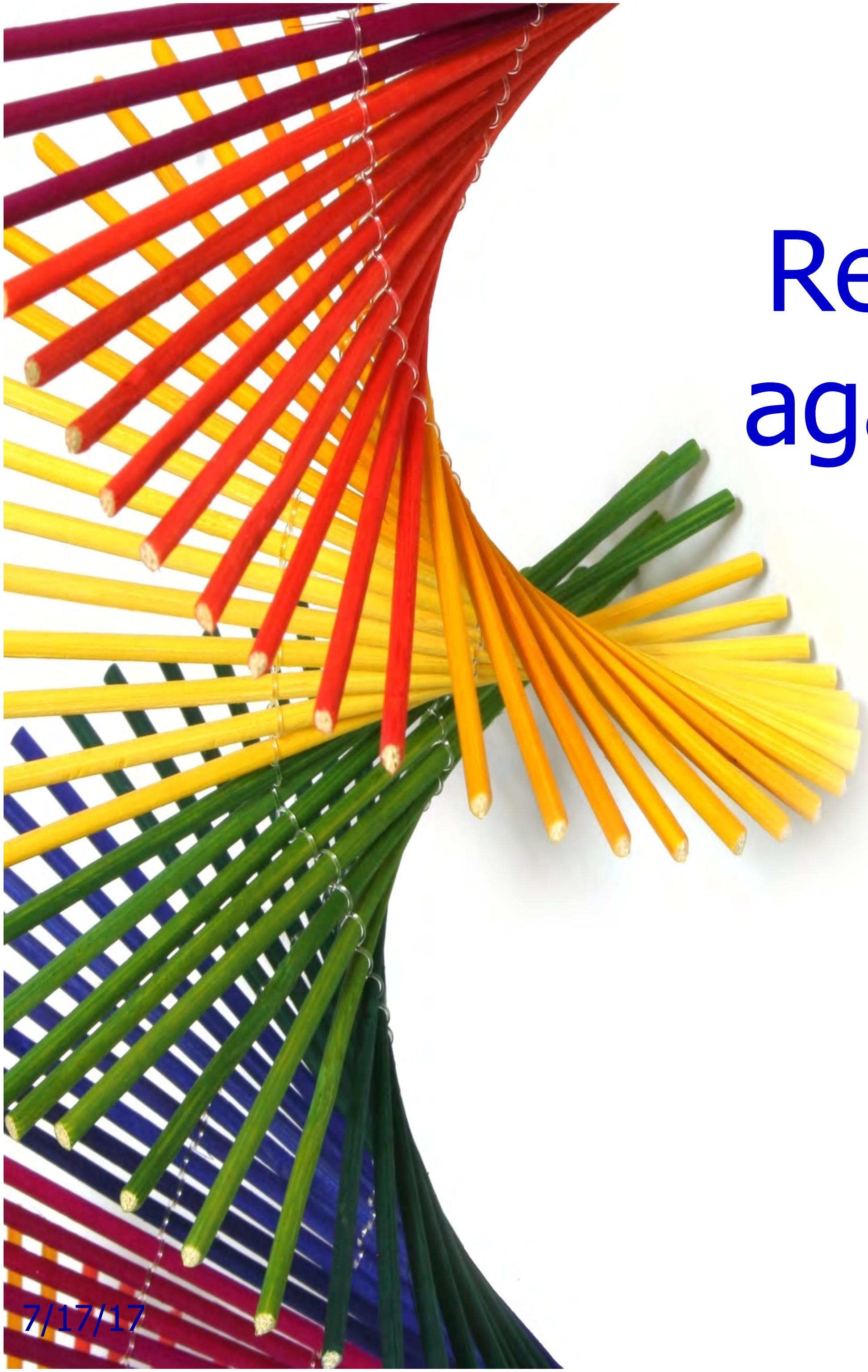
- David Aucsmith. Tamper-resistant software: An implementation. *Proceedings of the 1st International Workshop on Information Hiding*, 1996.
- C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient and stealthy opaque constructs. *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1998.
- B. Horne et al. Dynamic self-checking techniques for improved tamper resistance. *Proceedings of the Digital Rights Management Workshop*, 2001.
- C. Linn, and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. *Proceedings of the ACM Conference on Computer and Communications Security*, 2003.
- K. Coogan, G. Lu, and S. Debray, Deobfuscation of virtualization-obfuscated software: a semantics-based approach. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.
- J. Kinder. Towards static analysis of virtualization-obfuscated binaries. *Proceedings of the IEEE Working Conference on Reverse Engineering,* 2012.

Summary

- PVMs being increasingly used to provide software protection
- Knots are a novel scheme to protect dynamically-generated code in PVMs
- The approach addresses a major weakness in PVMs
- Working prototype created on the Linux platform

Outline

- Part I: Brief Biography
- Part II: Introduction to SDT
- Part III: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance
- Part III: SDT Code security applications
  - Obfuscating Virtualization
  - Replacement Attacks
  - PointISAs
  - Matryoshka



Replacement Attacks against VM-protected Applications

In Proceedings of the 8th International Conference on
Virtual Execution Environments (VEE '12), London UK

Replacement Attacks against VM-protected Applications

Sudeep Ghosh Jason Hiser Jack W. Davidson

Department of Computer Science, University of Virginia, Charlottesville, VA-22903, USA.
{sudeep,hiser,jwd}@virginia.edu

Abstract

Process-level virtualization is increasingly being used to enhance the security of software applications from reverse engineering and unauthorized modification (called *software protection*). Process-level virtual machines (PVMs) can safeguard the application code at run time and hamper the adversary's ability to launch dynamic attacks on the application. This dynamic protection, combined with its flexibility, ease in handling legacy systems and low performance overhead, has made process-level virtualization a popular approach for providing software protection. While there has been much research on using process-level virtualization to provide such protection, there has been less research on attacks against PVM-protected software. In this paper, we describe an attack on applications protected using process-level virtualization, called a replacement attack. In a replacement attack, the adversary replaces the protecting PVM with an attack VM thereby rendering the application vulnerable to analysis and modification. We present a general description of the replacement attack methodology and two attack implementations against a protected application using freely available tools. The generality and simplicity of replacement attacks demonstrates that there is a strong need to develop techniques that meld applications more tightly to the protecting PVM to prevent such attacks.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors – Run-time Environments; D.3.6 [*Operating Systems*]: Security and Protection

General Terms Security, Protection

Keywords Software protection, Process-level virtualization, Dynamic binary translation, Code obfuscation, Reverse engineering, Software tamper resistance.

1. Introduction

Process-level virtualization is a versatile and powerful technique that addresses a wide range of system challenges. It has been increasingly used to deliver solutions in the area of software protection (i.e., mechanisms that protect software applications from reverse-engineering attacks and tamper) [1, 20]. A number of commercial products have been designed to provide software protection via process-level virtualization such as VMProtect [51], Code Virtualizer [37], Themida [36]. A number of computer gaming software applications employ the StarForce virtualization system for copy protection and anti-reverse engineering [48]. Recently, malicious agents have used this protection technique to design state-of-the-art malware that can evade current detection systems [41]. There are several reasons why PVMs are popular amongst security researchers for enhancing software protection.

- Process-level virtualization provides a platform for enhanced run-time security. Attackers are increasingly using dynamic techniques to attack software (e.g., running applications under a debugger or a simulator) [5]. PVMs allow run-time monitoring and checking of the code being executed, making them an excellent tool for devising dynamic protection schemes [38]. PVMs can also mutate the application code as it is running (e.g., changing code and data locations, replacing instructions with semantically equivalent instructions, etc.), hampering iterative attacks [20].
- It is advantageous to have the protection techniques closely integrated with the application, yet keep the implementations separate. This modular approach enables easier testing and debugging of the system, and it allows legacy systems to be retrofitted with new protections without the need for modification and recompilation. PVMs can be used to provide such a flexible capability.
- Static protection schemes can be strengthened when the application is run under a PVM. For example, encryption is a useful technique that hampers static analysis of programs. Because the encrypted code cannot be run directly on commodity processors, the software decryption of the application code becomes a point of vulnerability. For example, schemes which decrypt the application in bulk are susceptible to dynamic analysis techniques [5], whereas decryption at a lower granularity (e.g., functions) can suffer from high overhead [9, 29]. In contrast, executing encrypted applications under the control of a PVM has been shown to have a better performance-security tradeoff [27]. The PVM incurs low information leakage, with the addition of a small performance overhead [20]. Another example is the improvement of the robustness of integrity checks that are located in the application. When run under a PVM, these integrity checks never execute from their original location, instead, they can be invoked from randomized locations in memory [20]. This randomization makes it harder for the attacker to locate and disable the checks.

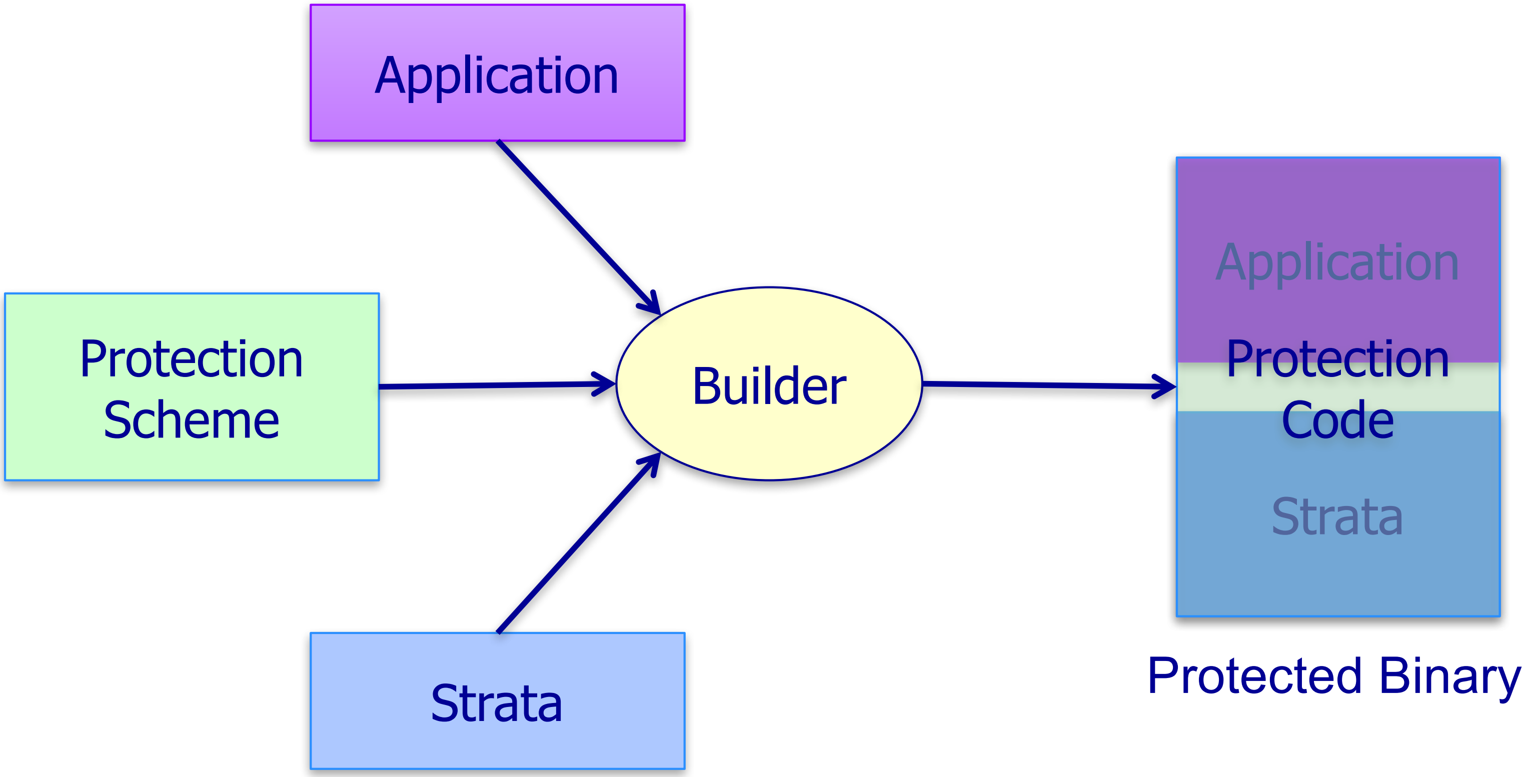
Considering their increasing use in program protection, it is important to assess the security of process-level virtualization as a whole. The existence of any weaknesses in the PVM or its interaction with the application can seriously undermine any protection mechanism that relies on process-level virtualization.

VEE'12, March 3–4, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-1175-5/12/03. . . $10.00

Replacement Attack

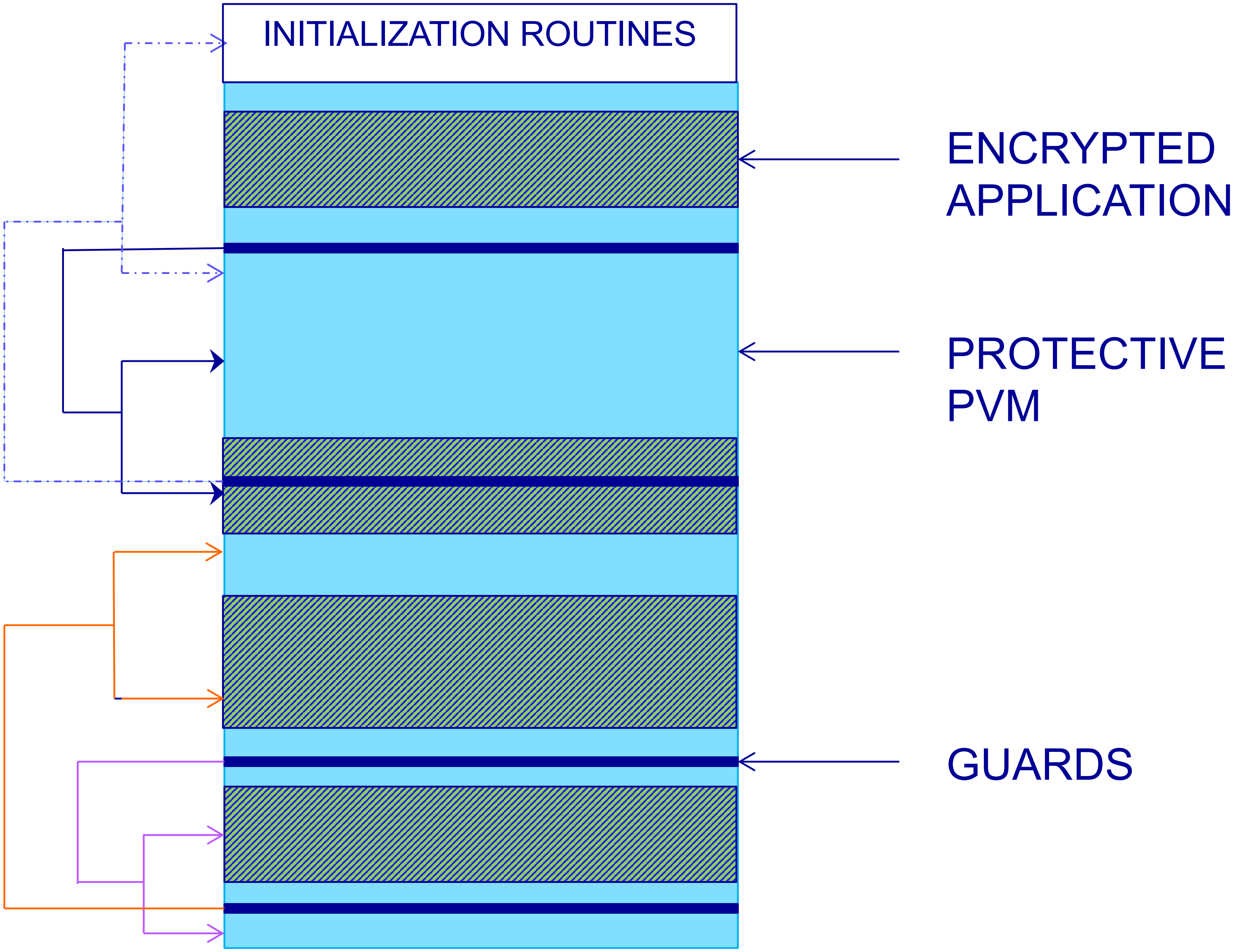
- Dynamically replace protective PVM with an attack PVM.
  - Protective PVM weakly bound.
  - Disables any run-time protection scheme based on PVM.
- Contributions:
  - Novel attack methodology that targets virtualization-protected application.
  - Extensive case study that deals with PVM-based program protections, and the implications of the replacement attack.
  - Demonstrates the need for stronger coupling between application and protective PVM.

Protection Architecture

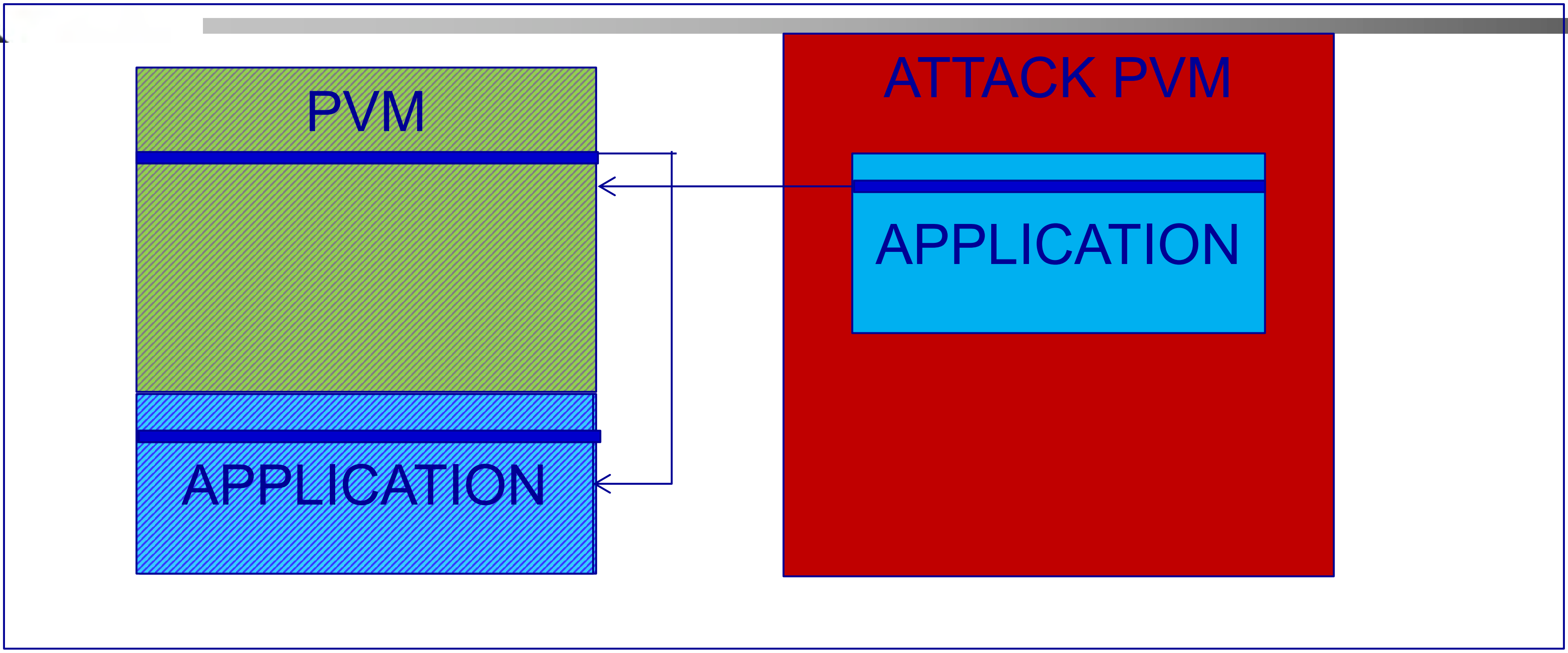


Protected Binary

Protected Application



Replacement Attack



- The protective PVM is substituted at run time.
- The static protections fail to protect against this class of attacks.

Discussion

- The attacker has to determine PVM entry function.
  - The attacker can search for specific instructions (e.g. on Intel IA32 platforms, context switch is preceded by (*pusha, pushf*) sequence.
  - Information flow tracing is a powerful tool that can aid in locating the entry function.
- The attacker has to determine the guest ISA.
  - Often, guest ISAs are simple RISC-like, lacking complex features.
    - Rolles, R. Unpacking virtualization obfuscators. In the *Proceedings of the 3rd USENIX Conference on Offensive Technologies,* 2009.
  - Researchers have designed techniques that facilitate automatic identification and extraction of cryptographic routines.

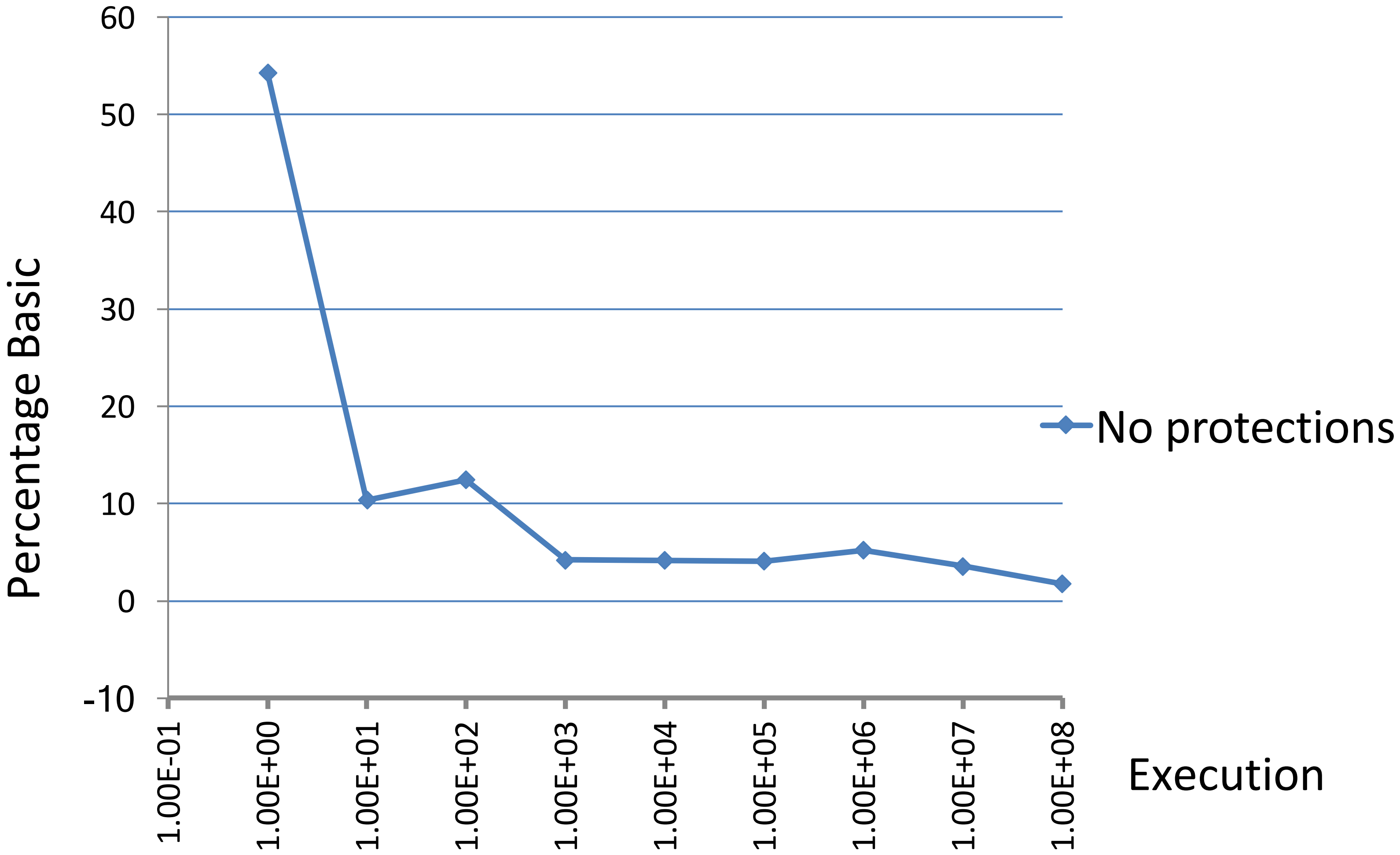
Attack Implementations

- Tools
  - Introspection framework: Pin
    - Luk, C. et al. Pin: building customized program analysis tools with dynamic instrumentation. *In Proceedings of the ACM Conference on Programming Language Design and Implementation, 2005*.
  - Attack VM: HDTrans
- Steps
  - Determine guest language (i.e., encryption algorithm and key) and configure attack PVM.
    - Grobert, F., Willems, C., and Holz, T. Automatic identification of cryptographic primitives in binary programs. In *The Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, 2011
  - Locate call to entry function.
  - Load HDTrans and virtualize the application.

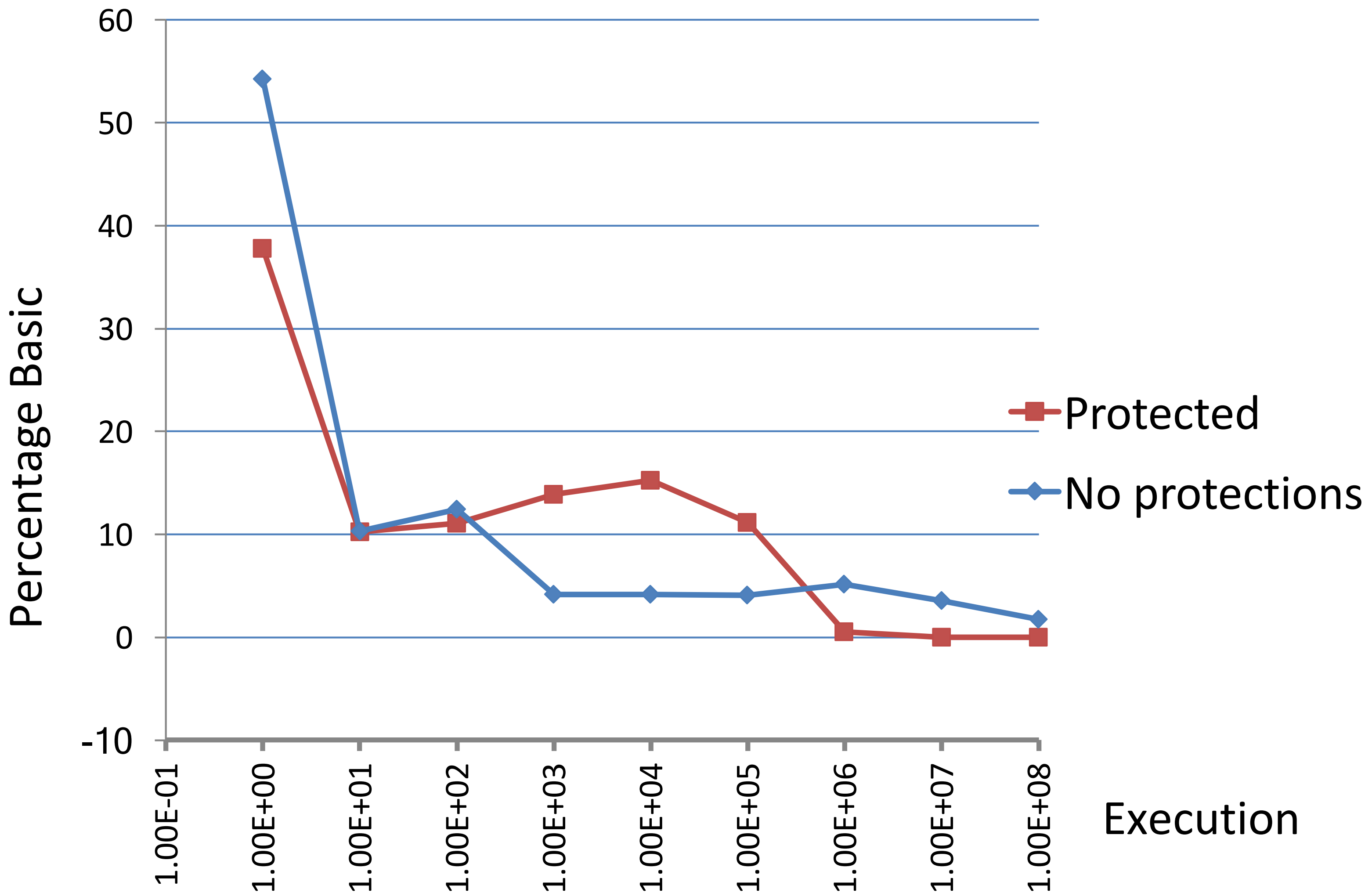
Implications

- Replacement attack disables PVM-based protections.
- How does it benefit the attacker?
- Answer: It makes program analysis easier.
    - Madou, M., Anckaert, B., De Sutter, B., and De Bosschere, K. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM Workshop on Digital Rights Management,* 2005.
    - Udupa, S., Debray, S., and Madou, M. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the International Working Conference on Reverse Engineering,* 2005.

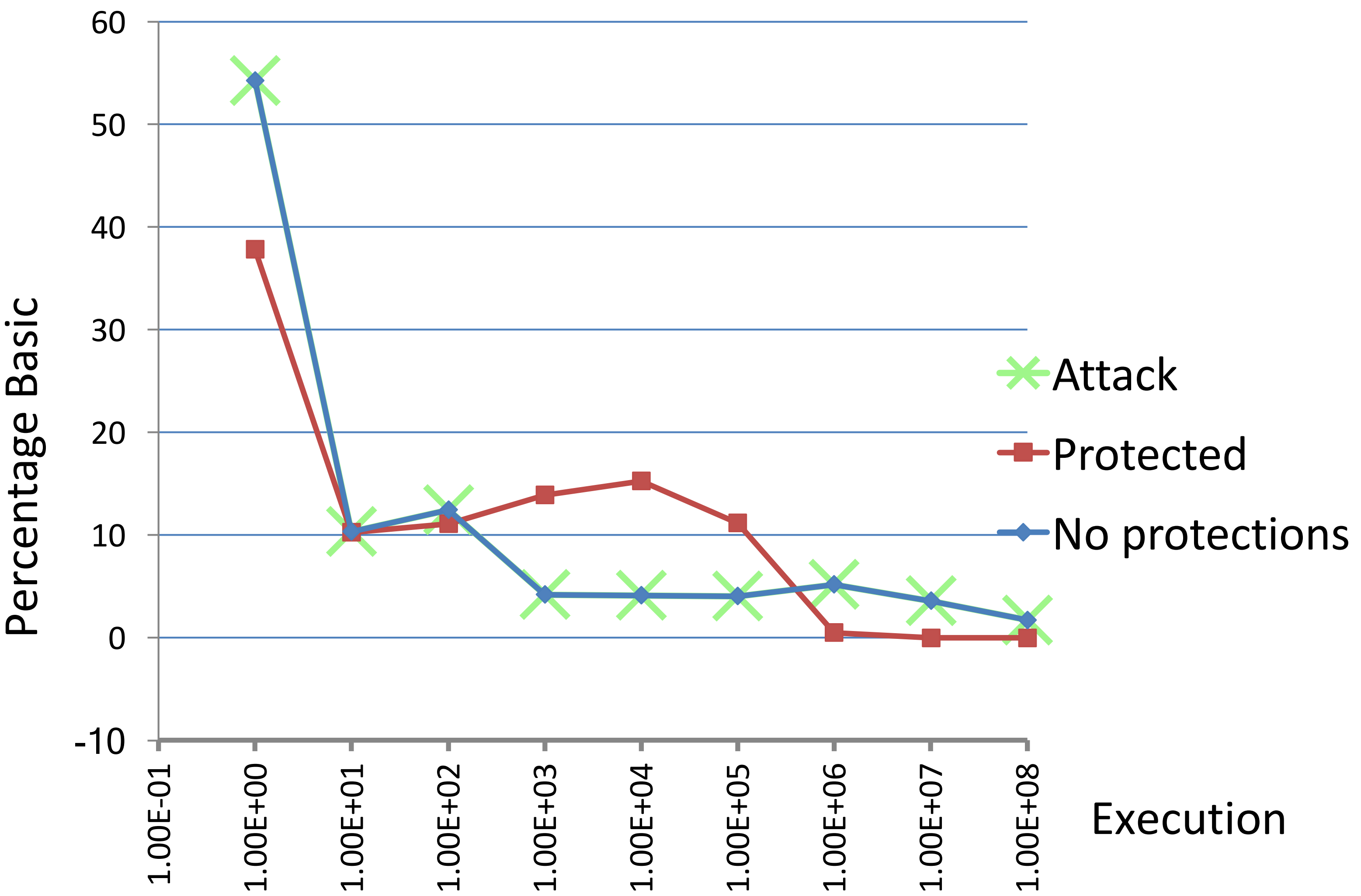
Basic Block Execution Frequencies



Basic Block Execution Frequencies



Basic Block Execution Frequencies



Effect of Flushing

Application Address	Rank (No protections)	Rank (Protected)	Rank (Attack)
0x8048830	1	121	1
0x804ac3c	2	45	2
0x804ae1b	3	13	3
0x80507d4	4	9	4
0x80507d9	5	173	5
0x80507c0	6	18	6
0x80507fa	7	29	7
0x805082c	8	351	8
0x8050810	9	139	9
0x804a750	10	779	10

Effect of Flushing

Application Address	Rank (No protections)	Rank (Protected)	Rank (Attack)
0x804bec0	162	1	162
0x804ae21	17	2	17
0x804ac74	36	3	36
0x804bed3	21	4	21
0x804a7a0	42	5	42
0x804abaf	164	6	164
0x804a81a	88	7	88
0x804a99b	126	8	126
0x80507d4	4	9	4
0x804a7ca	63	10	63

References

- Wurster G., van Oorschot, P.C., et al. A generic attack on checksumming-based software tamper resistance. *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.
- Sharif M., Lee W., et al. Automatic Reverse Engineering of Malware Emulators. *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.
- Coogan K., Debray S., et al. Deobfuscating Virtualization-Obfuscated Software: A Semantics-Based Approach. *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

Conclusion

- Replacement attack threatens PVM-protected applications.
  - Extensive case study shows that such attacks leave the application vulnerable to reverse-engineering.
  - Mounting such an attack only requires modest amount of information.

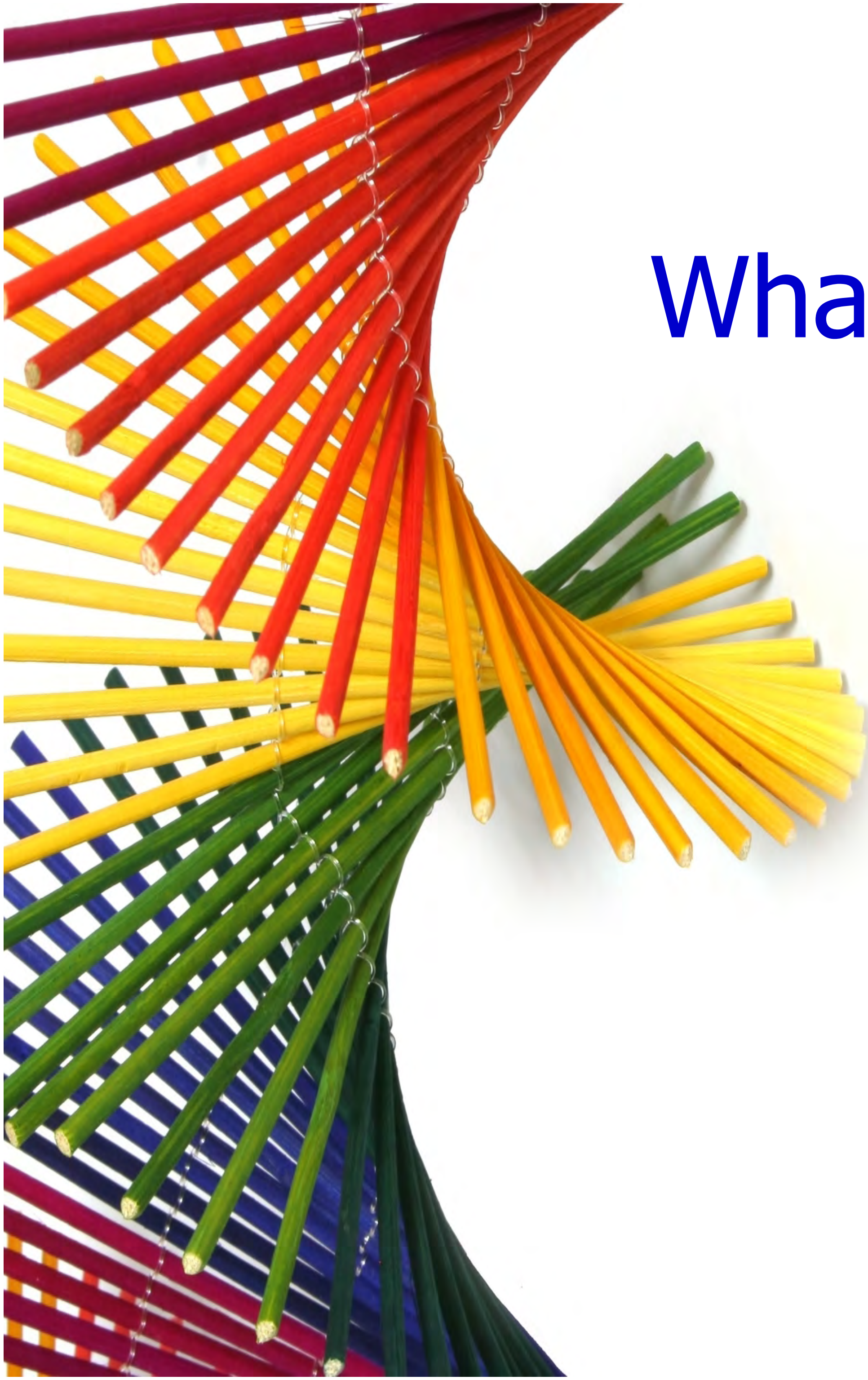
Outline

- Part I: Brief Biography

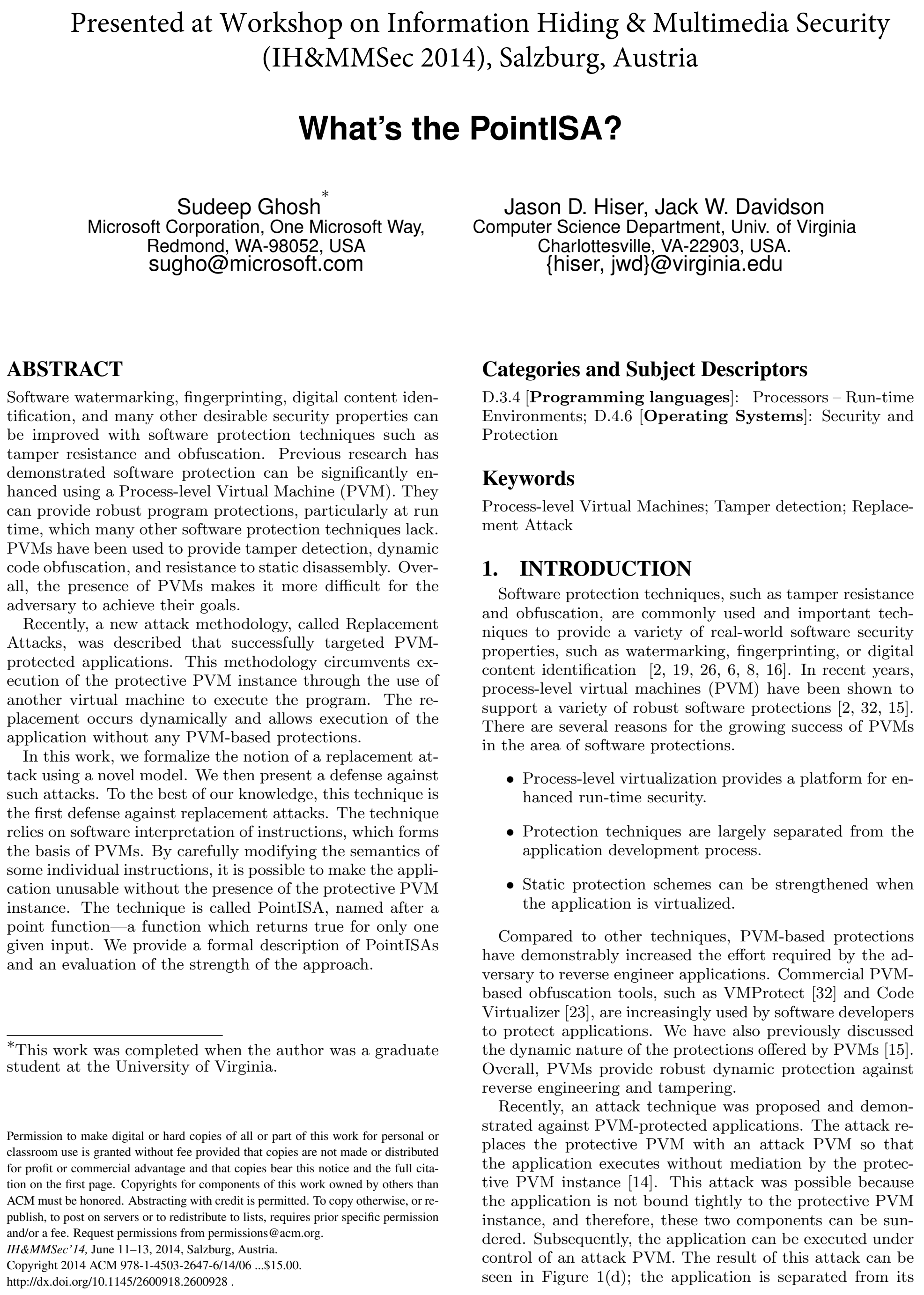
- Part II: Introduction to SDT

- Part III: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance

- Part III: SDT Code security applications
  - Obfuscating Virtualization
  - Replacement Attacks
  - PointISAs
  - Matryoshka



What’s the PointISA?



Presented at Workshop on Information Hiding & Multimedia Security
(IH&MMSec 2014), Salzburg, Austria

What's the PointISA?

Sudeep Ghosh[*]
Microsoft Corporation, One Microsoft Way,
Redmond, WA-98052, USA
sugho@microsoft.com

Jason D. Hiser, Jack W. Davidson
Computer Science Department, Univ. of Virginia
Charlottesville, VA-22903, USA.
{hiser, jwd}@virginia.edu

ABSTRACT

Software watermarking, fingerprinting, digital content identification, and many other desirable security properties can be improved with software protection techniques such as tamper resistance and obfuscation. Previous research has demonstrated software protection can be significantly enhanced using a Process-level Virtual Machine (PVM). They can provide robust program protections, particularly at run time, which many other software protection techniques lack. PVMs have been used to provide tamper detection, dynamic code obfuscation, and resistance to static disassembly. Overall, the presence of PVMs makes it more difficult for the adversary to achieve their goals.

Recently, a new attack methodology, called Replacement Attacks, was described that successfully targeted PVM-protected applications. This methodology circumvents execution of the protective PVM instance through the use of another virtual machine to execute the program. The replacement occurs dynamically and allows execution of the application without any PVM-based protections.

In this work, we formalize the notion of a replacement attack using a novel model. We then present a defense against such attacks. To the best of our knowledge, this technique is the first defense against replacement attacks. The technique relies on software interpretation of instructions, which forms the basis of PVMs. By carefully modifying the semantics of some individual instructions, it is possible to make the application unusable without the presence of the protective PVM instance. The technique is called PointISA, named after a point function—a function which returns true for only one given input. We provide a formal description of PointISAs and an evaluation of the strength of the approach.

[*]This work was completed when the author was a graduate student at the University of Virginia.

IH&MMSec'14, June 11–13, 2014, Salzburg, Austria.
Copyright 2014 ACM 978-1-4503-2647-6/14/06 ...$15.00.
http://dx.doi.org/10.1145/2600918.2600928 .

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors – Run-time Environments; D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Process-level Virtual Machines; Tamper detection; Replacement Attack

1. INTRODUCTION

Software protection techniques, such as tamper resistance and obfuscation, are commonly used and important techniques to provide a variety of real-world software security properties, such as watermarking, fingerprinting, or digital content identification [2, 19, 26, 6, 8, 16]. In recent years, process-level virtual machines (PVM) have been shown to support a variety of robust software protections [2, 32, 15]. There are several reasons for the growing success of PVMs in the area of software protections.

- Process-level virtualization provides a platform for enhanced run-time security.
- Protection techniques are largely separated from the application development process.
- Static protection schemes can be strengthened when the application is virtualized.

Compared to other techniques, PVM-based protections have demonstrably increased the effort required by the adversary to reverse engineer applications. Commercial PVM-based obfuscation tools, such as VMProtect [32] and Code Virtualizer [23], are increasingly used by software developers to protect applications. We have also previously discussed the dynamic nature of the protections offered by PVMs [15]. Overall, PVMs provide robust dynamic protection against reverse engineering and tampering.

Recently, an attack technique was proposed and demonstrated against PVM-protected applications. The attack replaces the protective PVM with an attack PVM so that the application executes without mediation by the protective PVM instance [14]. This attack was possible because the application is not bound tightly to the protective PVM instance, and therefore, these two components can be sundered. Subsequently, the application can be executed under control of an attack PVM. The result of this attack can be seen in Figure 1(d); the application is separated from its

Solution: PointISA

- Semantically bind the application to the PVM
  - Select an instruction not found in the program
    “add esp, -28“, not just “add“
  - Give that byte pattern a new meaning
    “add esp, -28“ means “push 0x1”
  - Insert protections into the program

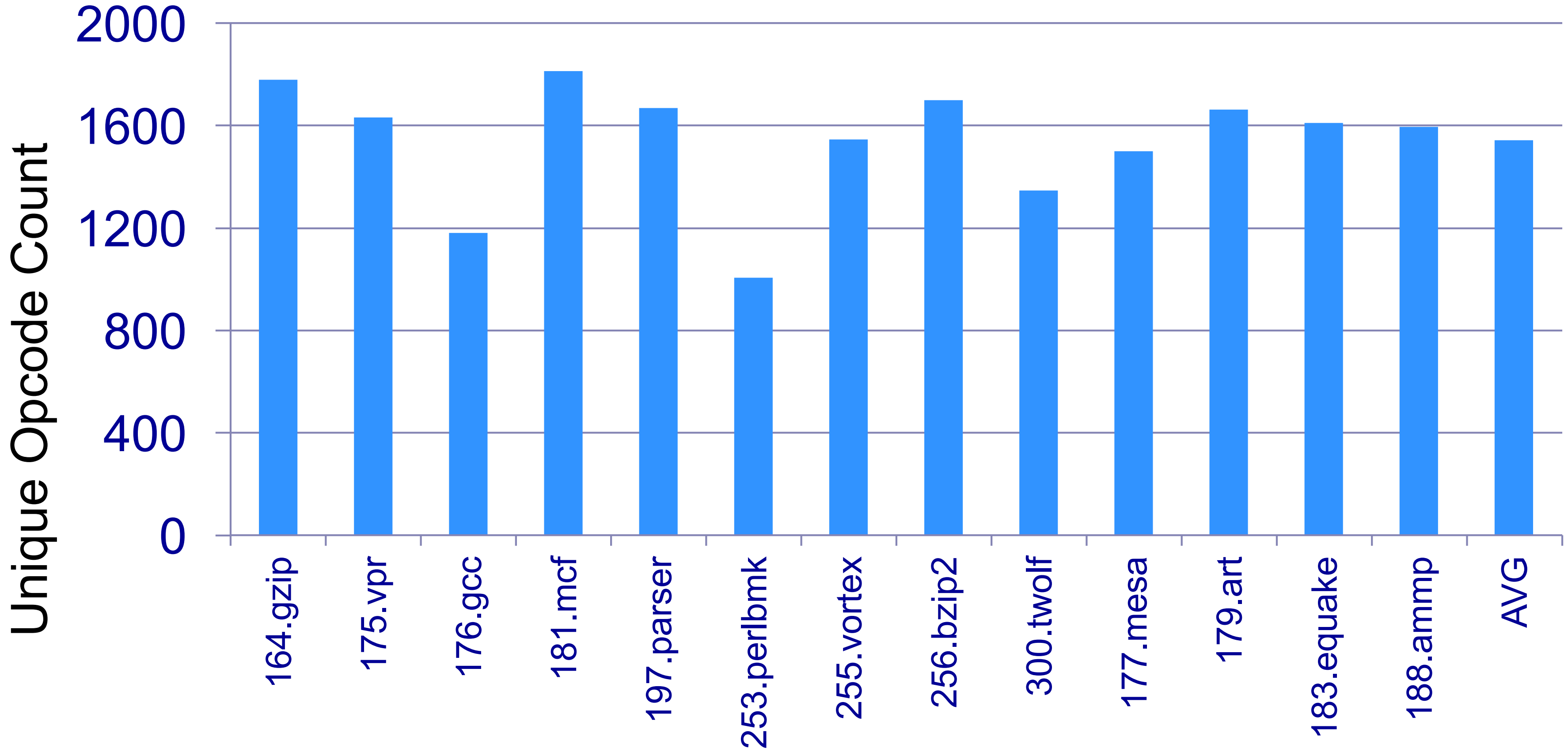
    ```
    add esp, -28
    if ( pop() != 0x1) goto
      tamper_response;
    ```

  - Teach the PVM about it
  - Repeat until “secure enough”
  - Release/Run the program

Selecting an Instruction

- Compile a bunch of programs
- Compile database of instructions that the compiler used
- Select an instruction from DB that is not included in program to protect

Selection of Homographic Instructions



- i^{HG} not part of I_P
- Create DB of known opcodes using static analysis
- Calculate set difference for each benchmark

Responding to Attacks

- Auditor Mechanism (shown previously)
  - Advantage: Guaranteed detection
  - Disadvantage: Adversary might locate and disable check
- Value Modification Mechanism –
  - Change a register/memory location in the program randomly, let it keep running

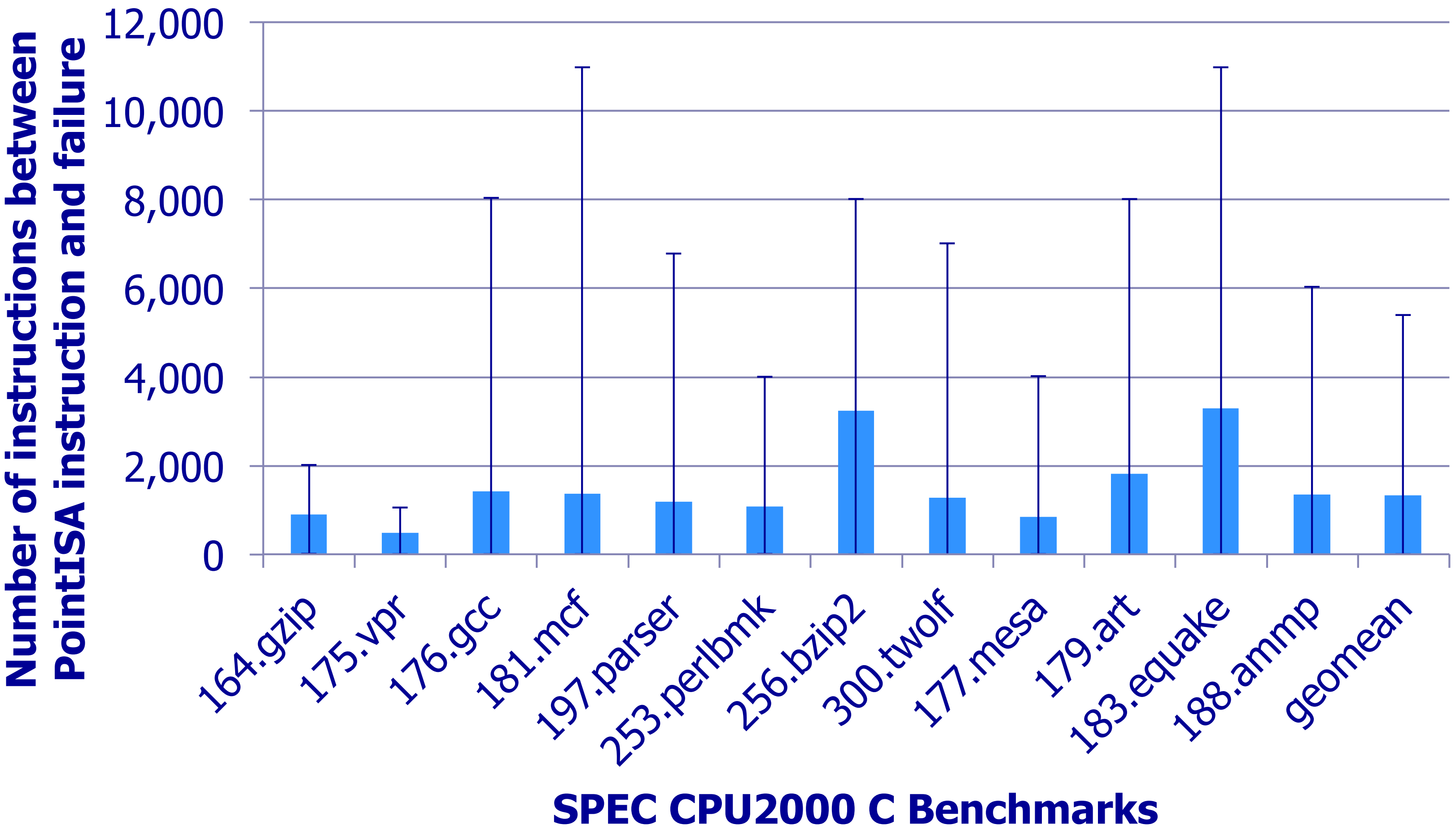
```
add eax, -28  ; really push esp
pop esp
```

  - Advantage: Stealthy
  - Disadvantage: The application could run normally

Response Evaluation

- SPEC CPU2006 C Benchmarks
- Auditors always fail, verified experimentally.
- Value Modification mechanism
  - 120 trials (12 programs, 10 tries each)
  - Insert 1 protection, placed randomly
  - Protection randomly modifies one register to constant value
    - e.g. `mov eax, 7`
- Subject program to replacement attack
- Typically program faults
- Four times it ran to completion (3.3%)
  - Correct result thrice
  - Bad output once

Value Modification (Instructions Executed before Failure)



PointISA: Stealthiness Notes

- Avoid odd instructions
  - `pushf`
- Fail silently
  - Corrupt state, modify control flow, etc.
- Attacker has the VM binary, obfuscate it too

Related Work

- Tamper detection/resistance
  - Aucsmith, Tamper-resistant software: An implementation
  - Other: Biondi, Horne, Jacob, Linn, Wurster, Collberg, …
    - Problem: requires hardware, high overhead, unreasonable models, etc.
- PVM-based solutions
  - VMProtect, Themida, Ghosh
    - Problem: Subject to replacement attacks
- System VM-based solutions
  - Terra, Overshadow
    - Problem: Deployment challenging, hardware required

Discussion: What if Attacker knows the PointISA Meanings?

- Replacement attacks are possible if he/she knows ALL the PointISA meanings
- Probably too hard to find all of them manually
- Can it be done automatically?
  - Not currently.

Discussion: What if Attacker Reverse Engineers PVM?

- Protections (obfuscations, etc.) also applied to PVM
- Consequently, not easy to do automatically
  - Attacker will spend lots of time reverse engineering PVM, instead of reverse engineering the program!

I call that a win.

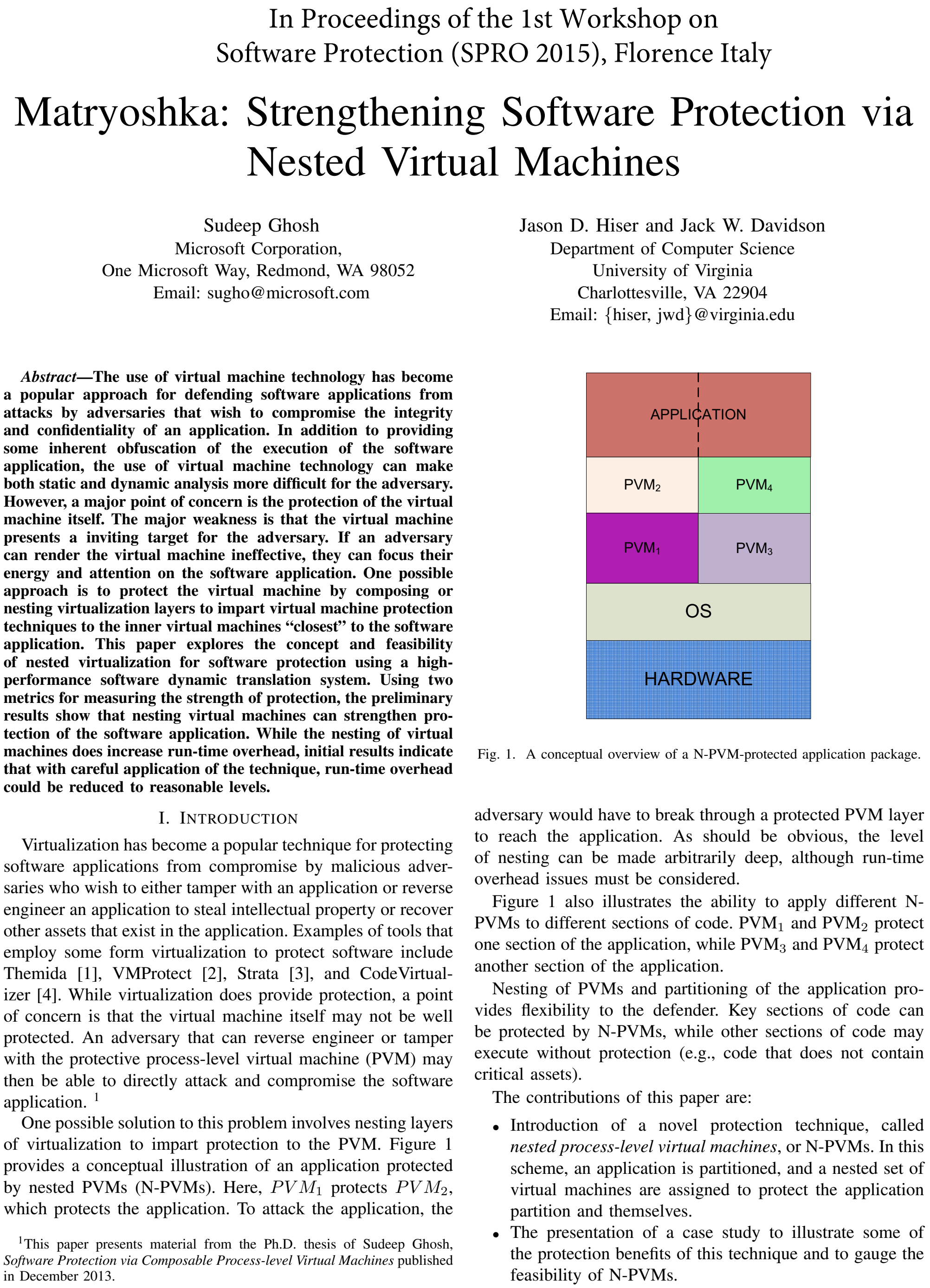
Just use a Completely Custom ISA

- VMProtect, Thermida do this
  - But, designing a full, completely-custom ISA is hard
  - So, they use very simple ISAs
  - And, have lots of slowdown for executing the custom ISA
  - So, the custom ISA is only used for critical bits, not the whole program, which makes it easier to attack
  - And, the simple ISA can be semi-automatically reverse engineered

Outline

- Part I: Brief Biography
- Part II: Introduction to SDT
- Part III: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance
- Part III: SDT Code security applications
  - Obfuscating Virtualization
  - Replacement Attacks
  - PointISAs
  - Matryoshka

Matryoshka: Strengthening Software Protection via Nested Virtual Machines



In Proceedings of the 1st Workshop on
Software Protection (SPRO 2015), Florence Italy

Matryoshka: Strengthening Software Protection via Nested Virtual Machines

Sudeep Ghosh
Microsoft Corporation,
One Microsoft Way, Redmond, WA 98052
Email: sugho@microsoft.com

Jason D. Hiser and Jack W. Davidson
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
Email: {hiser, jwd}@virginia.edu

***Abstract*—The use of virtual machine technology has become a popular approach for defending software applications from attacks by adversaries that wish to compromise the integrity and confidentiality of an application. In addition to providing some inherent obfuscation of the execution of the software application, the use of virtual machine technology can make both static and dynamic analysis more difficult for the adversary. However, a major point of concern is the protection of the virtual machine itself. The major weakness is that the virtual machine presents a inviting target for the adversary. If an adversary can render the virtual machine ineffective, they can focus their energy and attention on the software application. One possible approach is to protect the virtual machine by composing or nesting virtualization layers to impart virtual machine protection techniques to the inner virtual machines "closest" to the software application. This paper explores the concept and feasibility of nested virtualization for software protection using a high-performance software dynamic translation system. Using two metrics for measuring the strength of protection, the preliminary results show that nesting virtual machines can strengthen protection of the software application. While the nesting of virtual machines does increase run-time overhead, initial results indicate that with careful application of the technique, run-time overhead could be reduced to reasonable levels.**

I. Introduction

Virtualization has become a popular technique for protecting software applications from compromise by malicious adversaries who wish to either tamper with an application or reverse engineer an application to steal intellectual property or recover other assets that exist in the application. Examples of tools that employ some form virtualization to protect software include Themida [1], VMProtect [2], Strata [3], and CodeVirtualizer [4]. While virtualization does provide protection, a point of concern is that the virtual machine itself may not be well protected. An adversary that can reverse engineer or tamper with the protective process-level virtual machine (PVM) may then be able to directly attack and compromise the software application. [1]

Fig. 1. A conceptual overview of a N-PVM-protected application package.

One possible solution to this problem involves nesting layers of virtualization to impart protection to the PVM. Figure 1 provides a conceptual illustration of an application protected by nested PVMs (N-PVMs). Here, PVM_1 protects PVM_2, which protects the application. To attack the application, the adversary would have to break through a protected PVM layer to reach the application. As should be obvious, the level of nesting can be made arbitrarily deep, although run-time overhead issues must be considered.

Figure 1 also illustrates the ability to apply different N-PVMs to different sections of code. PVM$_1$ and PVM$_2$ protect one section of the application, while PVM$_3$ and PVM$_4$ protect another section of the application.

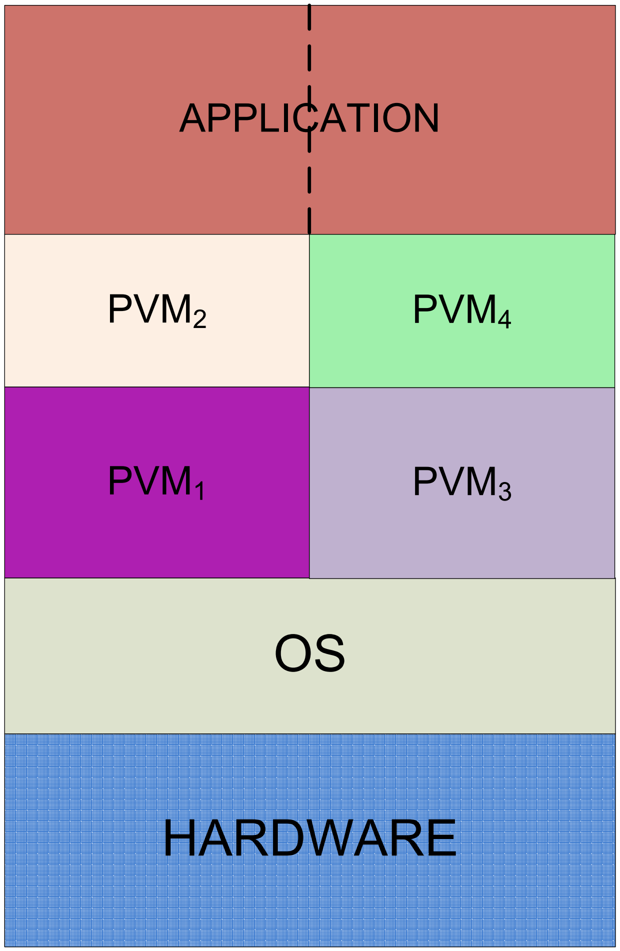
Nesting of PVMs and partitioning of the application provides flexibility to the defender. Key sections of code can be protected by N-PVMs, while other sections of code may execute without protection (e.g., code that does not contain critical assets).

The contributions of this paper are:

- Introduction of a novel protection technique, called *nested process-level virtual machines*, or N-PVMs. In this scheme, an application is partitioned, and a nested set of virtual machines are assigned to protect the application partition and themselves.
- The presentation of a case study to illustrate some of the protection benefits of this technique and to gauge the feasibility of N-PVMs.

[1] This paper presents material from the Ph.D. thesis of Sudeep Ghosh, *Software Protection via Composable Process-level Virtual Machines* published in December 2013.

Matryoshka: Nested PVMs



APPLICATION

PVM_2

PVM_4

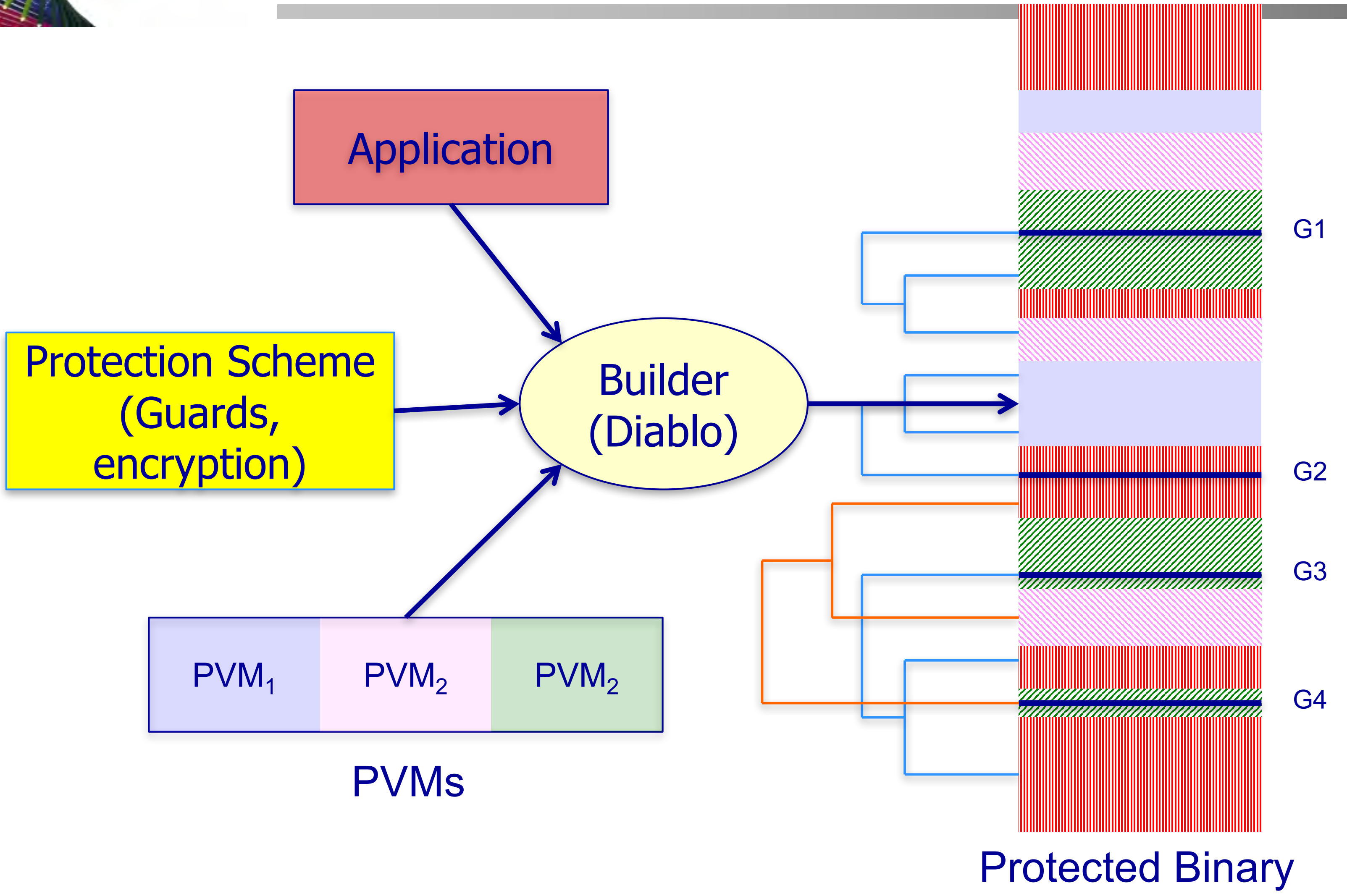
PVM_1

PVM_3

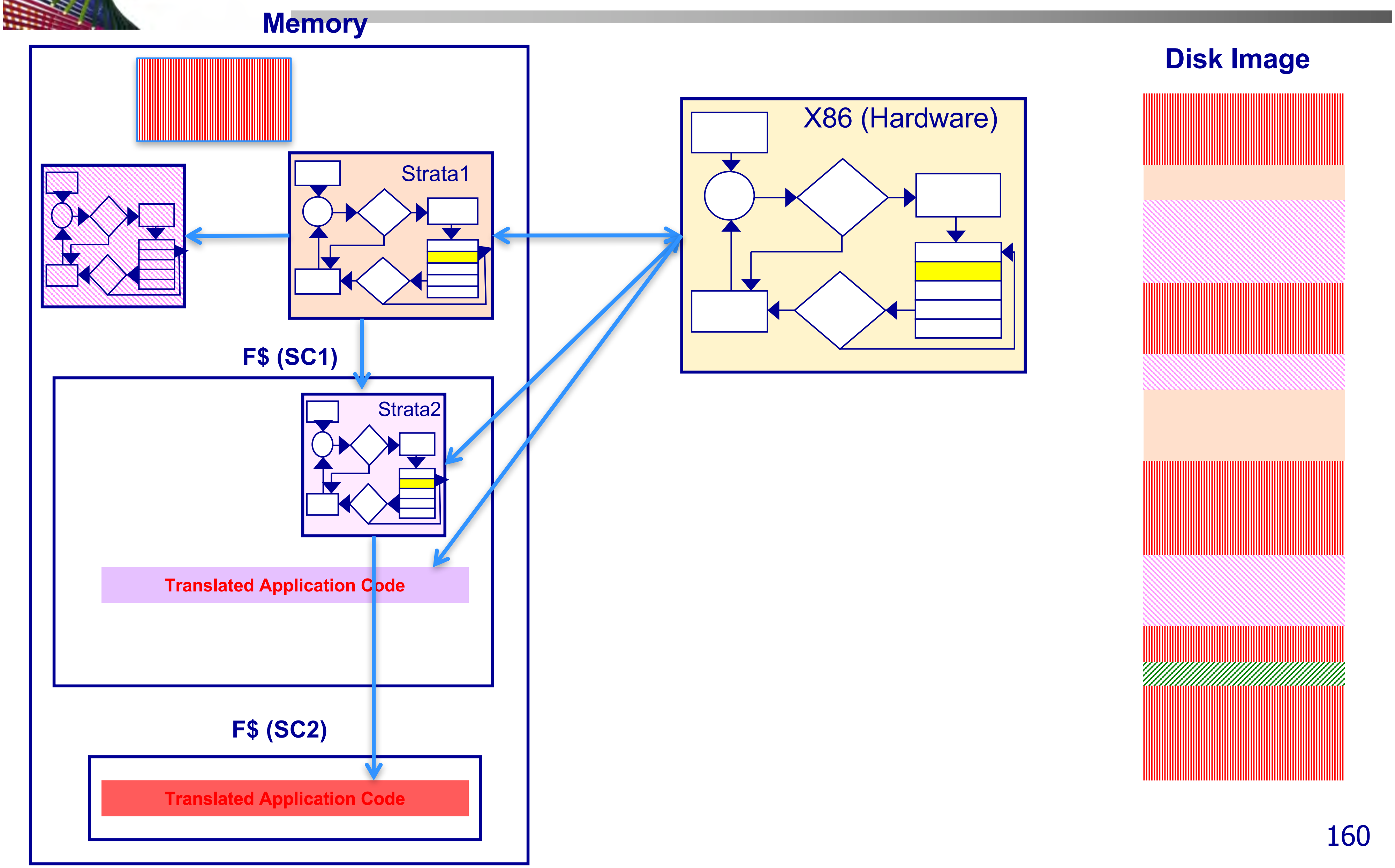
OS

HARDWARE

Software Protection via Virtualization



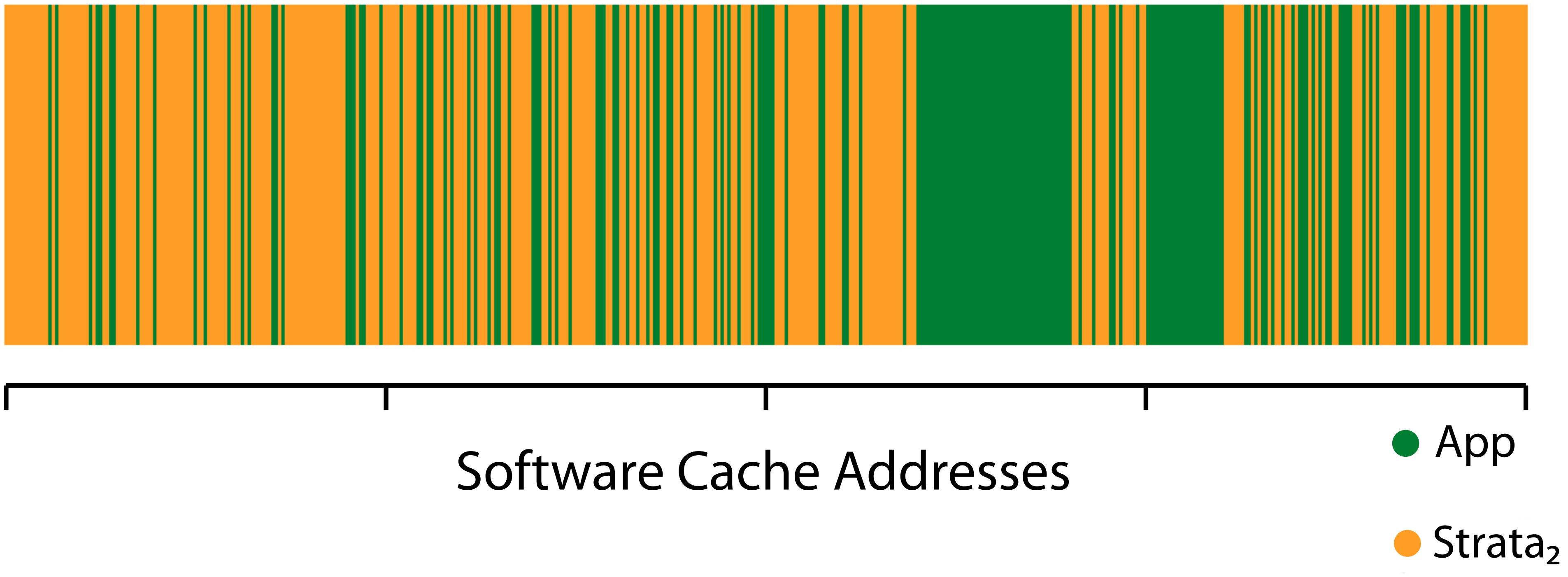
Nested PVMs



Memory
Strata1
F$ (SC1)
Strata2
Translated Application Code
F$ (SC2)
Translated Application Code
X86 (Hardware)
Disk Image

Evaluation

F$ Diversity



- Use compression ratio as a proxy for diversity and obfuscation.
- Single PVM: 149; N-PVM: 15.63

Evaluation

Cyclomatic Complexity

- Developed by McCabe in 1976 as a measure of software complexity (TSE Vol. 2, No. 4)
  - $M = E - N + 2P$

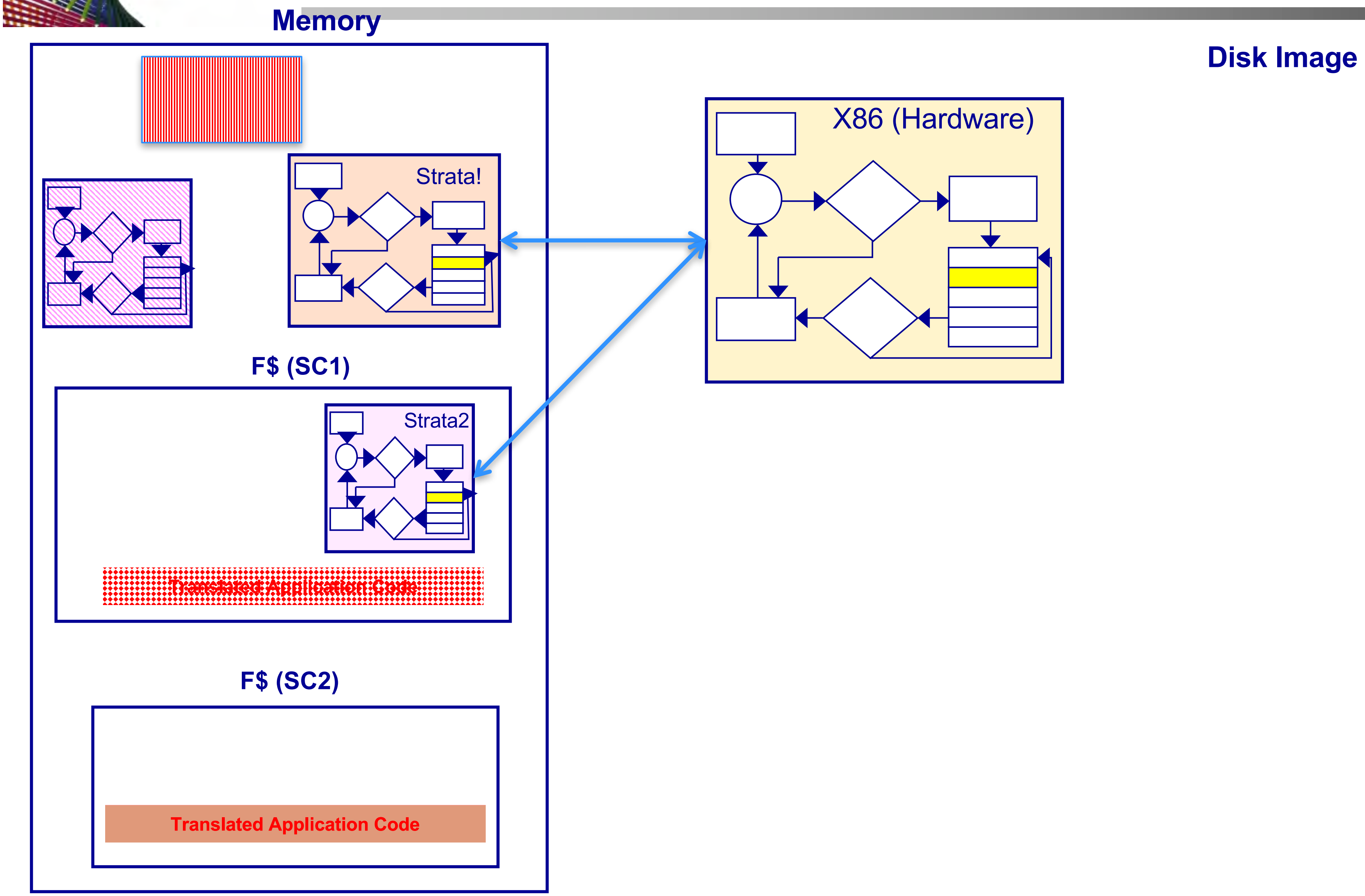
Benchmark	CC for PVM	CC for N-PVM	Increase
176.gcc	1,604	80,109	49X
181.mcf	351	9828	27X
256.perlbmk	803	32,903	40X
179.Art	181	5,130	27X

Evaluation

Run-time Overhead

- With a nesting level of two, the base run-time overhead was 35X.
- The problem is trampoline patching (i.e., self-modifying code), which causes excessive F$ flushes.

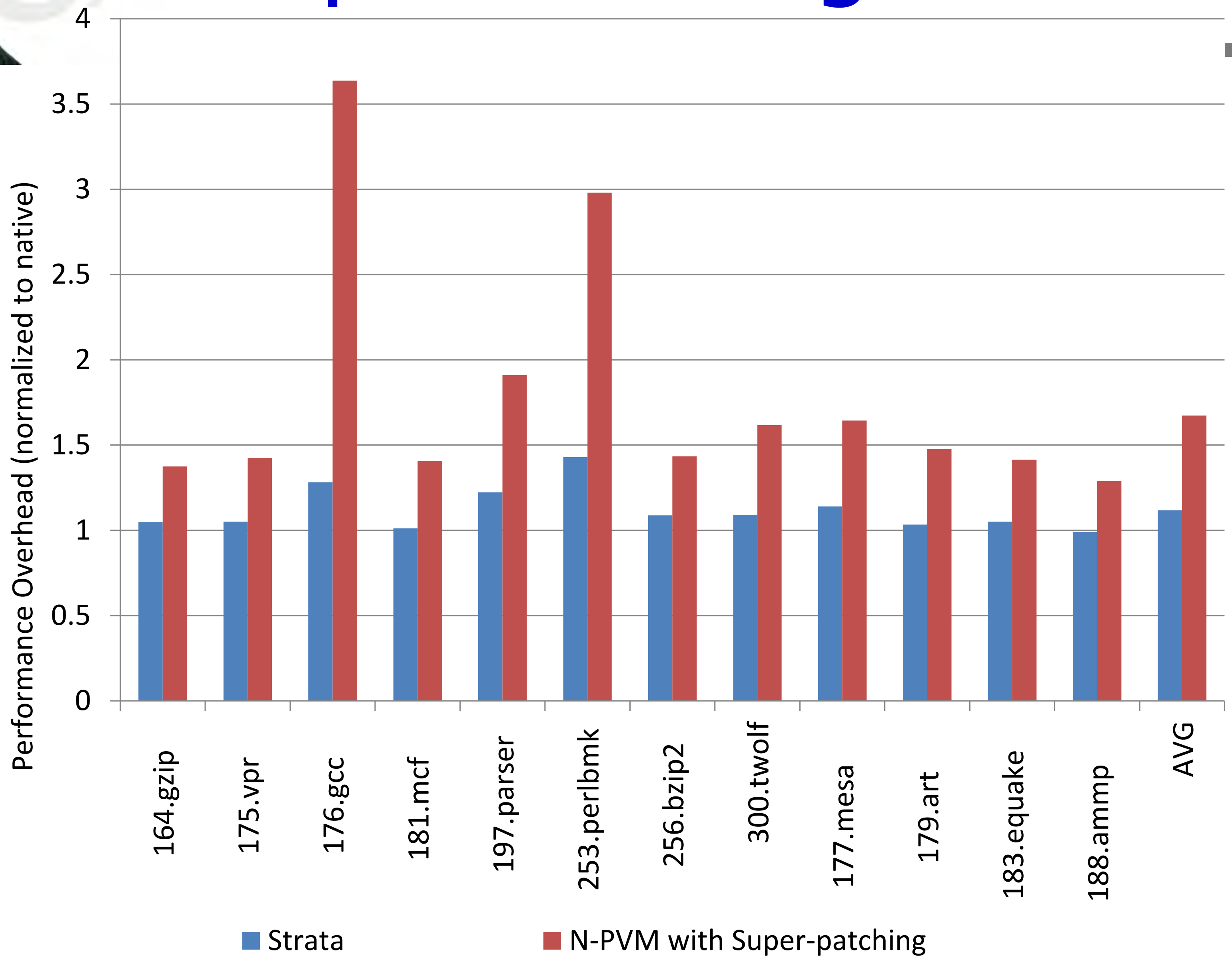
Nested PVMs


Memory
Disk Image
X86 (Hardware)
Strata!
F$ (SC1)
Strata2
Translated Application Code
F$ (SC2)
Translated Application Code

Super Patching

- When Strata$_n$ patches a trampoline, the patch information is sent to Strata$_{n-1}$
- When a patched (in F\$$_2$) target block is translated to F\$$_{n-1}$ by Strata$_{n-1,}$ Strata$_{n-1}$ patches its F\$ (F\$$_{n-1}$), thereby avoiding the F\$ flush

Super Patching Overhead



Related Work

- Collberg and Nagra [Pearson 2006] provide an excellent overview of the area
- Anckaert et al [DRM 2006] showed the promise of virtualization for software protection
- Themida [1] and VMProtect [2] use interpreted virtual machines for software protection. Nested VMs apply to them as well.
- Anckaert et al [QoP 2007] discuss metrics for metrics for software protection

Summary

- Nested PVMs can significantly increase the complexity of software that is the target of crackers
- More research is needed to determine when and how to apply nested VMs to software to balance run-time performance and the strength of the protection provided