# Monitoring Hypervisor Integrity at Runtime

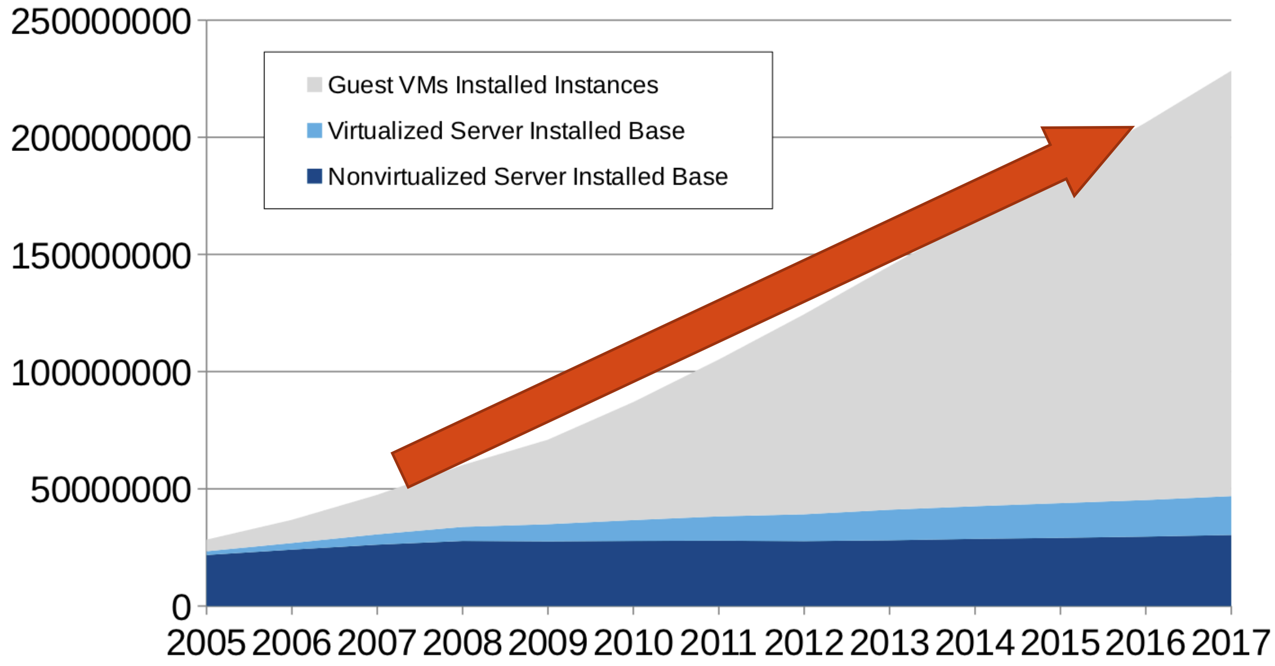
Student: Cuong Pham
PIs: Prof. Zbigniew Kalbarczyk, Prof. Ravi K. Iyer
ACC Meeting, Oct 2015

# Motivation - Server Virtualization Trend

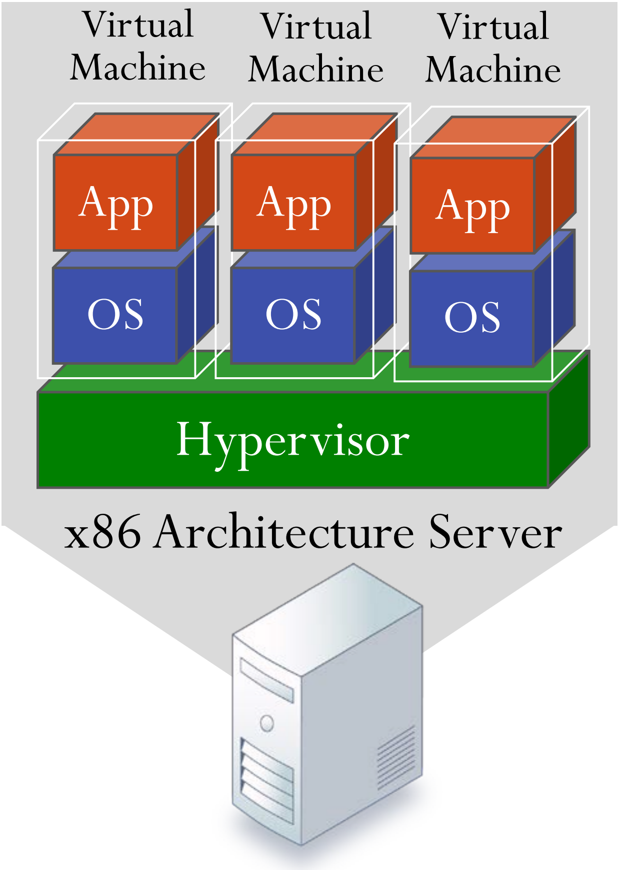
**51%**

x86 servers were **virtualized** in 2012

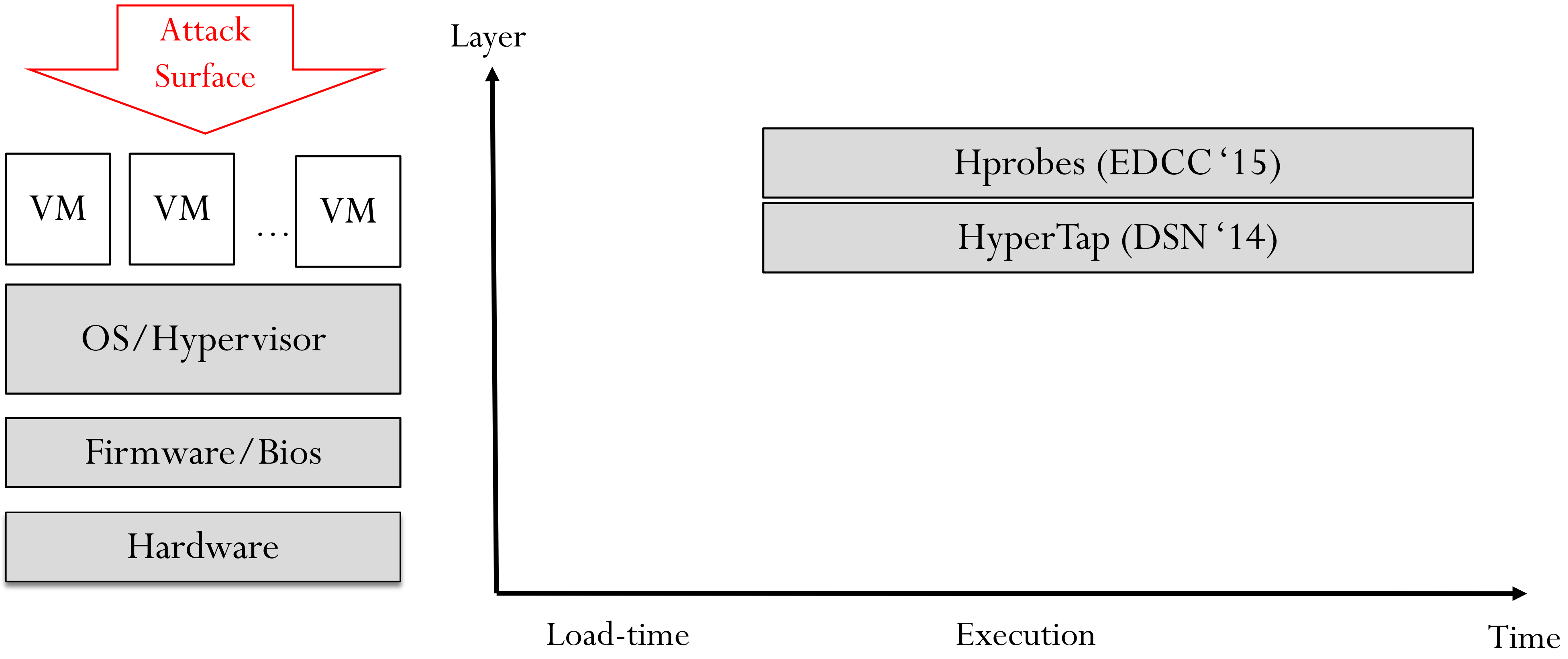
Source: 451 Research's TheInfoPro service reports



Source: Derivative analysis based on Worldwide Virtual Machine 2013–2017 Forecast: Virtualization Buildout Continues Strong IDC #242762 / Aug 2013

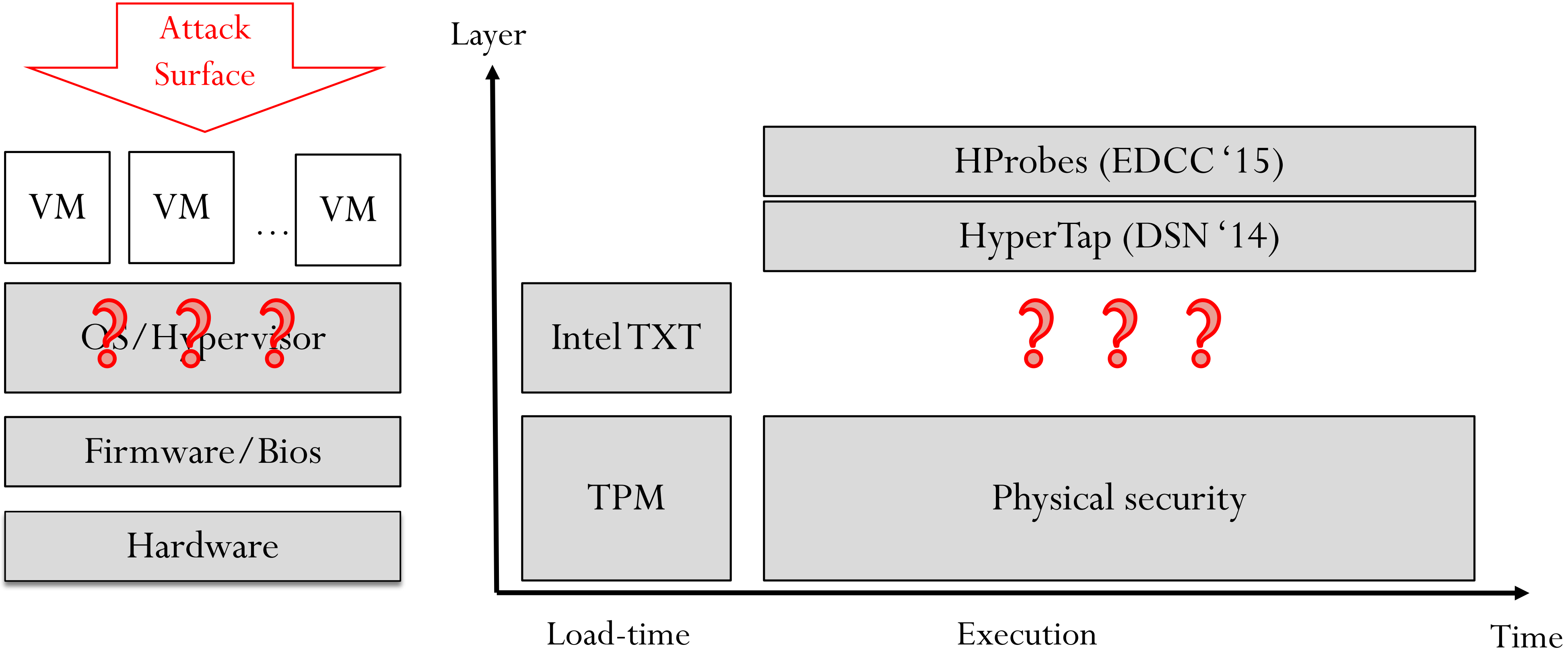


# Building Secure & Reliable VMs



- Chain of trust must be built from *bottom up*
- … and *continuously through time*

# Building Secure & Reliable VMs



Attack Surface

VM

VM

…

VM

OS/Hypervisor

? ? ?

Firmware/Bios

Hardware

Layer

HProbes (EDCC '15)

HyperTap (DSN '14)

Intel TXT

? ? ?

TPM

Physical security

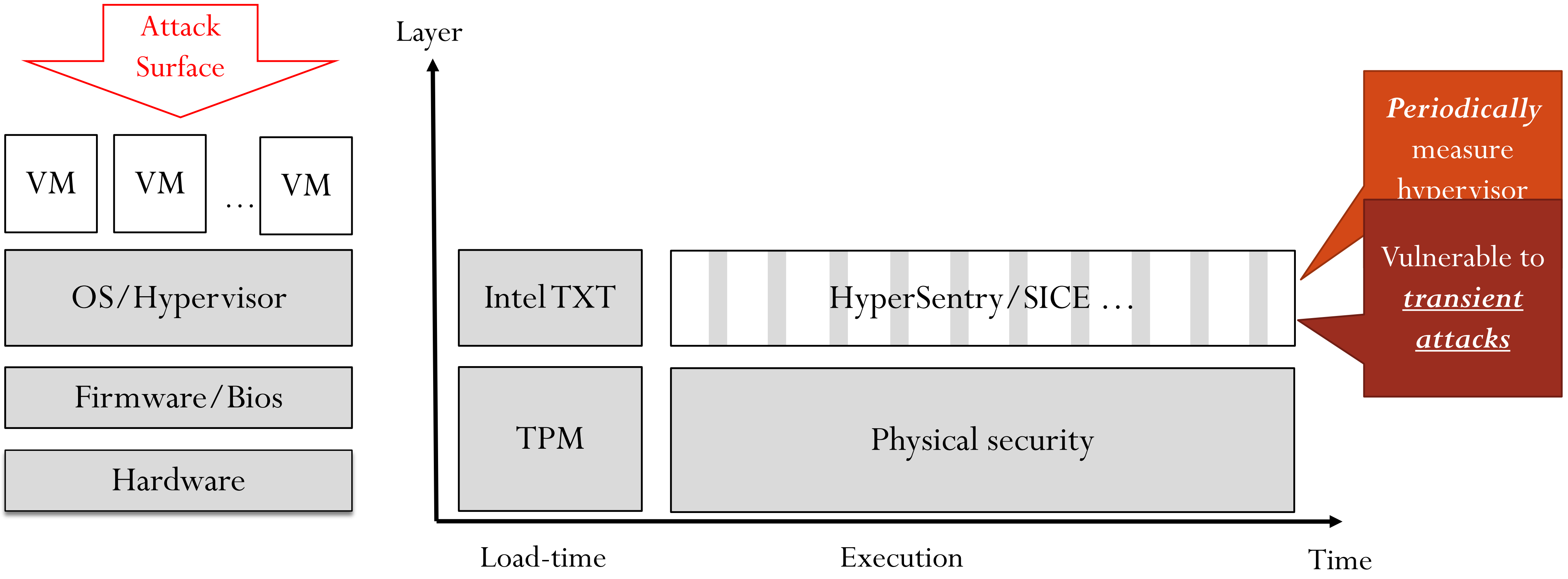
Load-time

Execution

Time

- Chain of trust must be built from *bottom up*
- … and *continuously through time*

# Protect Hypervisors: Existing approaches



Attack Surface

VM VM … VM

OS/Hypervisor

Firmware/Bios

Hardware

Layer

Intel TXT

HyperSentry/SICE …

TPM

Physical security

Load-time

Execution

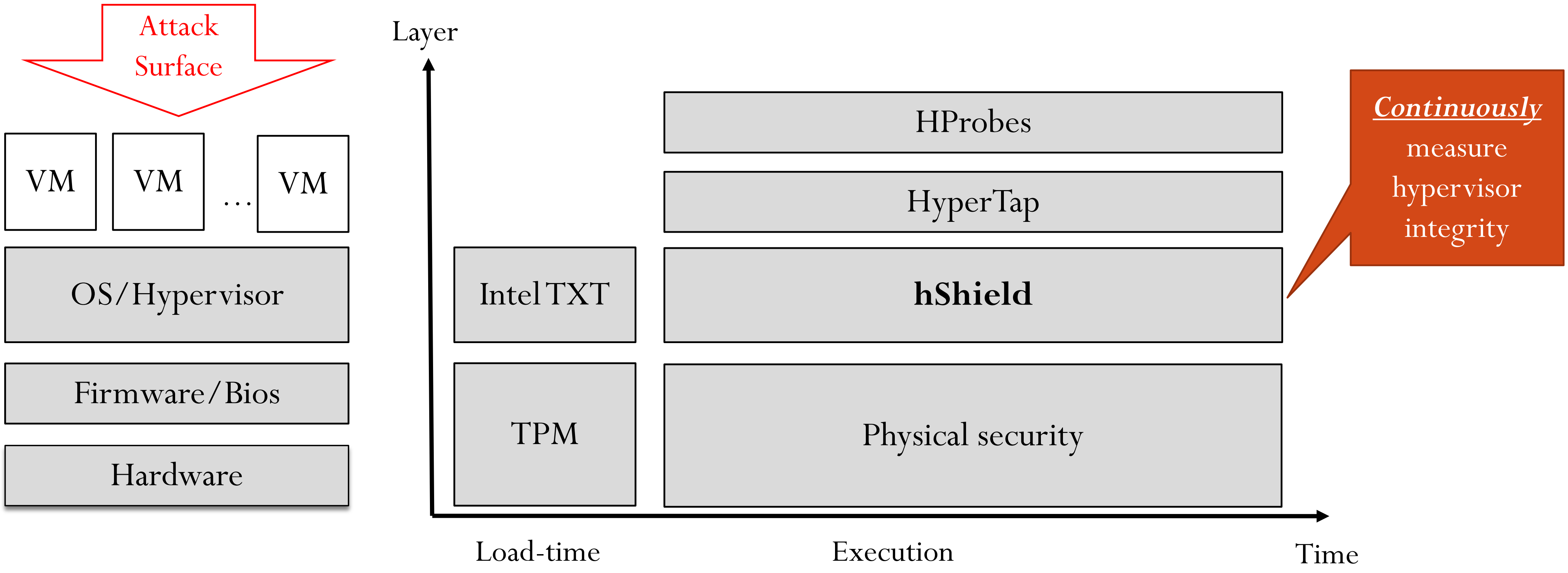
Time

***Periodically*** measure hypervisor

Vulnerable to ***transient attacks***

- Did I say "*continuously through time*"?

# Introducing hShield



Attack Surface

VM

VM

…

VM

OS/Hypervisor

Firmware/Bios

Hardware

Layer

HProbes

HyperTap

Intel TXT

**hShield**

TPM

Physical security

Load-time

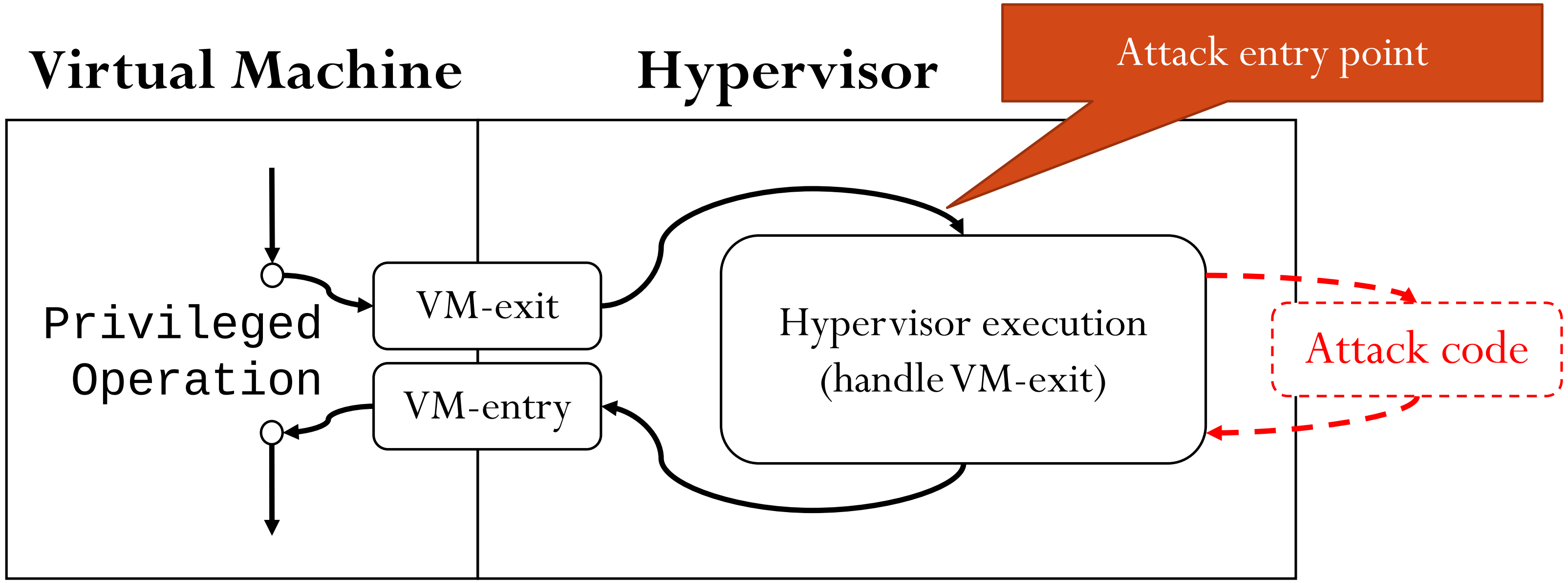
Execution

Time

***Continuously*** measure hypervisor integrity

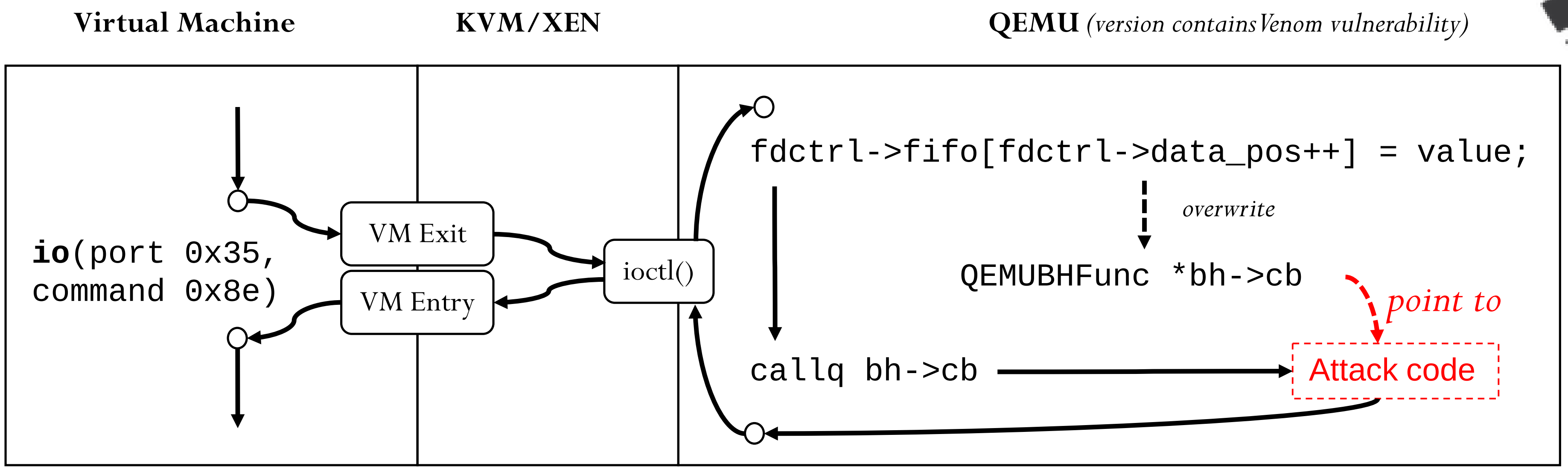
- Assumption: Hardware is trusted
  - TPM, Intel TXT are enabled
  - Physical security

# Threat Model: VM Escape Attacks



- Hardware Assisted Virtualization
- Attackers have full control of guest OS
- Violate hypervisor Control Flow Integrity (CFI)

# Example: Venom VM Attack (CVE-2015-3456)


**Virtual Machine**

**KVM/XEN**

**QEMU** *(version contains Venom vulnerability)*

**io**(port 0x35, command 0x8e)

VM Exit

VM Entry

ioctl()

`fdctrl->fifo[fdctrl->data_pos++] = value;`

*overwrite*

`QEMUBHFunc *bh->cb`

*point to*

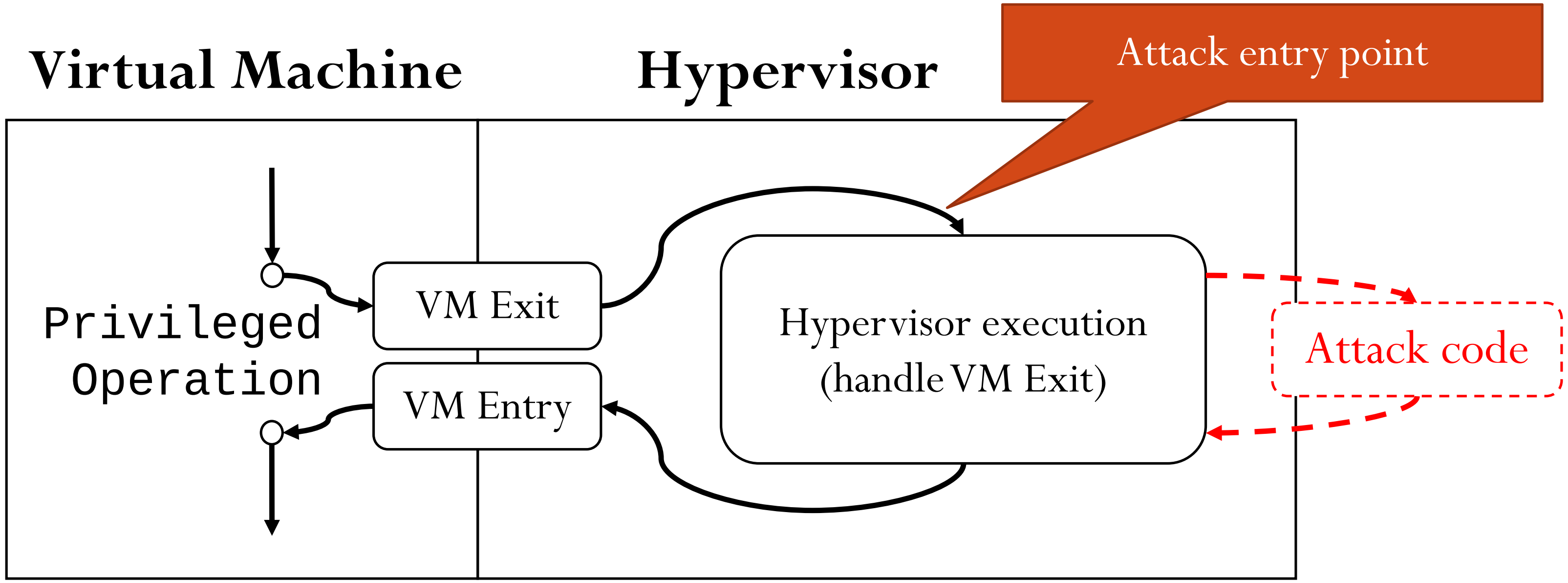
`callq bh->cb`

Attack code

**Description**
Though the VENOM vulnerability is also agnostic of the guest operating system, an attacker (or an attacker's malware) would need to have administrative or root privileges in the guest operating system in order to exploit VENOM
**Impact Type:** Allows unauthorized disclosure of information; Allows unauthorized modification; Allows disruption of service

Source: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3456

# Threat Model: VM Escape Attacks



- **Attack code**
  - DoS host
  - Access other co-located VMs (e.g., sniffing network traffic, stealing images)
  - Install backdoors, access secrets in host…

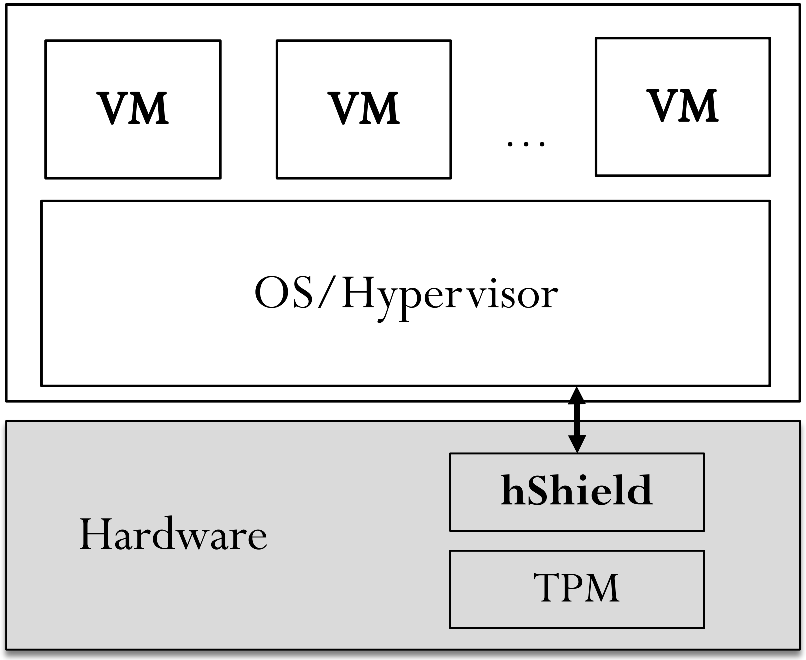
# VM Escape-enabling CVEs…

- CVE-2007-1744 – Directory traversal vulnerability in shared folders feature
- CVE-2008-0923 – Path traversal vulnerability in VMware's shared folders implementation
- CVE-2009-1244 – Cloudburst (VMware virtual video adapter vulnerability)
- CVE-2012-0217 – 64-bit PV guest privilege escalation vulnerability
- CVE-2014-0983 – Oracle VirtualBox 3D acceleration multiple memory corruption vulnerabilities
- CVE-2015-(2336-2340) – Escaping VMware Workstation through COM1 (5 CVE!!!)
- CVE-2015-3456 – QEMU heap overflow flaw in floppy disk driver
- CVE-2015-5154 – QEMU heap overflow flaw while processing certain ATAPI commands.

# hShield Design Goals

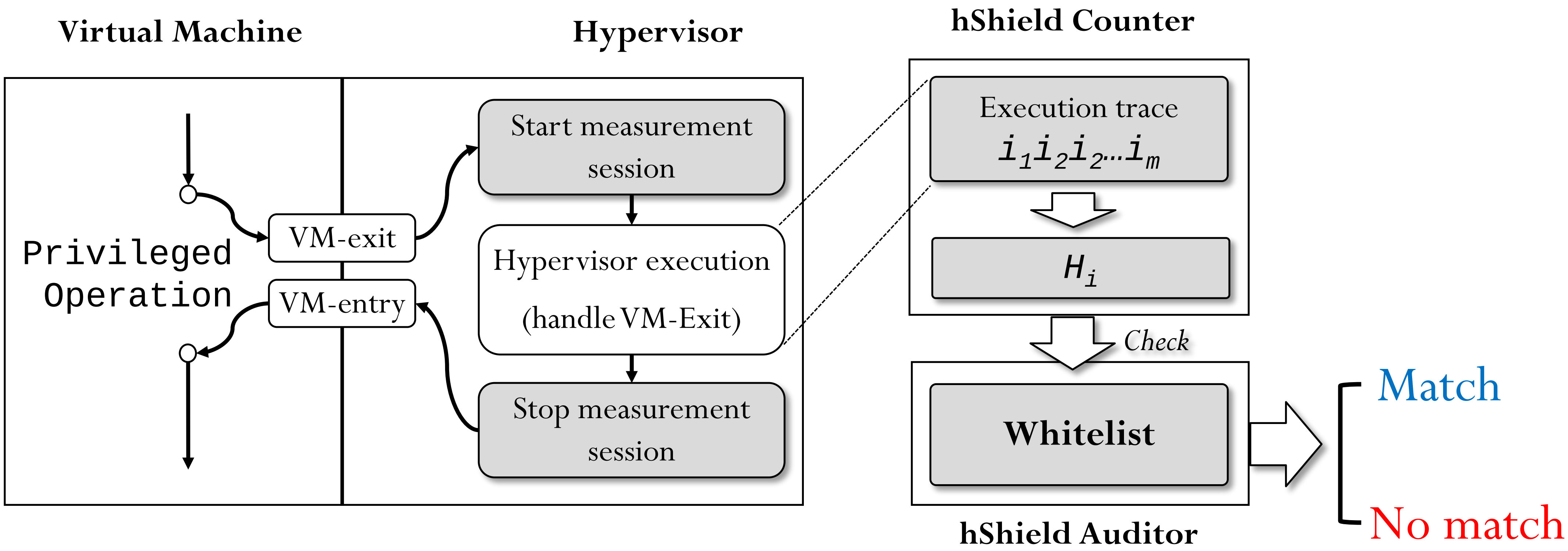
1. Resistance to zero-day VM escape attacks
2. Detect both transient and persistent attacks
3. Small performance overhead in attack-free executions
4. Support target randomization

# hShield Approach



- **Continuous monitoring**
  - Detect both persistent and transient attacks
- **White-list monitoring**
  - Detect unknown attacks
- **Hardware extension**
  - Hardware isolation and performance

# hShield Overview



**Virtual Machine**

**Hypervisor**

**hShield Counter**

Privileged Operation

VM-exit

VM-entry

Start measurement session

Hypervisor execution (handle VM-Exit)

Stop measurement session

Execution trace $i_1 i_2 i_2 \ldots i_m$

$H_i$

*Check*

**Whitelist**

**hShield Auditor**

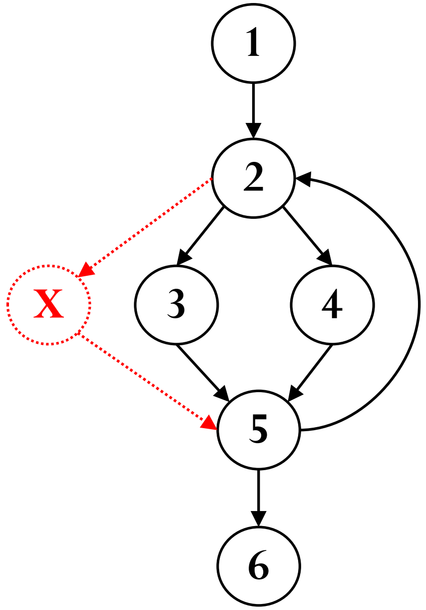
Match

No match

Detect a VM-escape attack right at the end of the exploited VM-exit: defeat transient attacks.

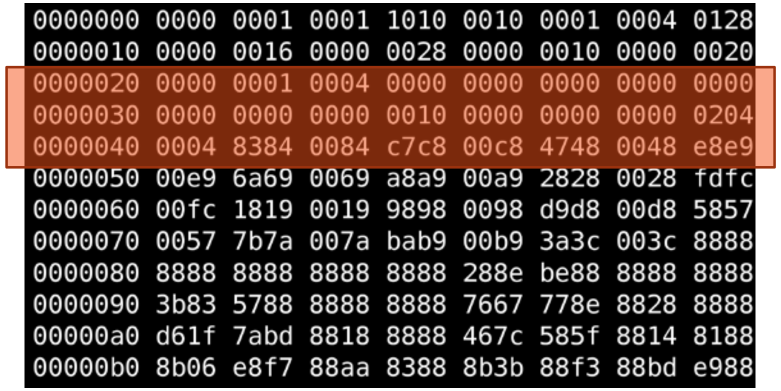
# White-list vs. Black-list

- **White-list**
  - ***No one*** can access except the white-listed
  - Prevents <u>unknown attacks</u>
  - E.g., Control Flow Integrity (CFI) techniques



**Control Flow Graph (CFG)**

- **Black-list**
  - ***Everyone*** can access except the black-listed
  - Prevents known (black-listed) attacks
  - E.g., Signature-based malware detection

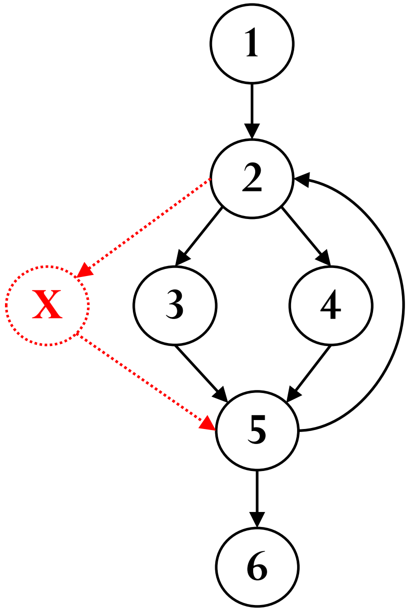


```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
```

Signature based detection

# Current CFI Techniques

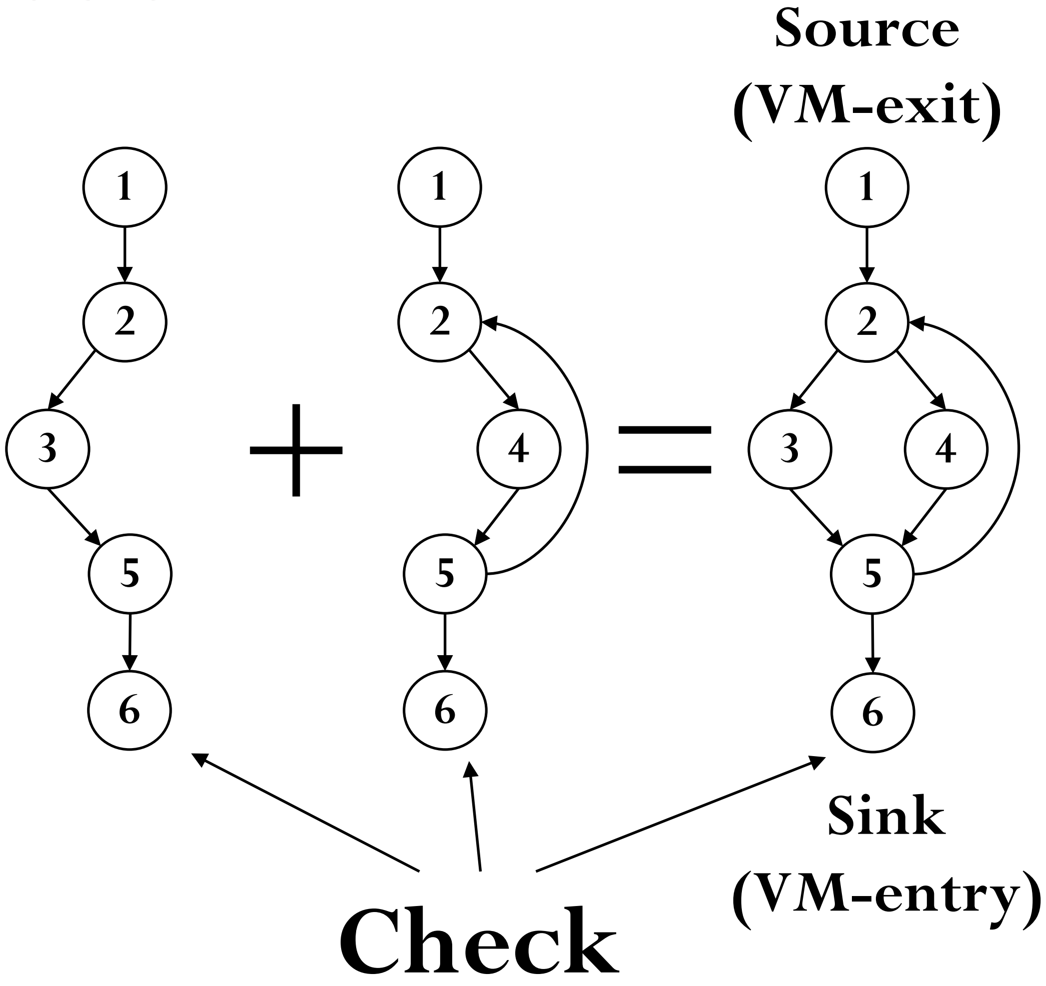
- **Heavily relies on static analysis to construct CFG**
  - No interactions with dynamic libraries, OS
  - Heuristic (e.g., pointer analysis is imperfect)
  - Scalability issues (e.g., large binaries)
- **High runtime overhead**
  - Check at every branch
- **Compatibility issues against**
  - Address Space Layout Randomization (ASLR)
  - Programs relying on dynamic binary re-writing (e.g., Linux kernels)



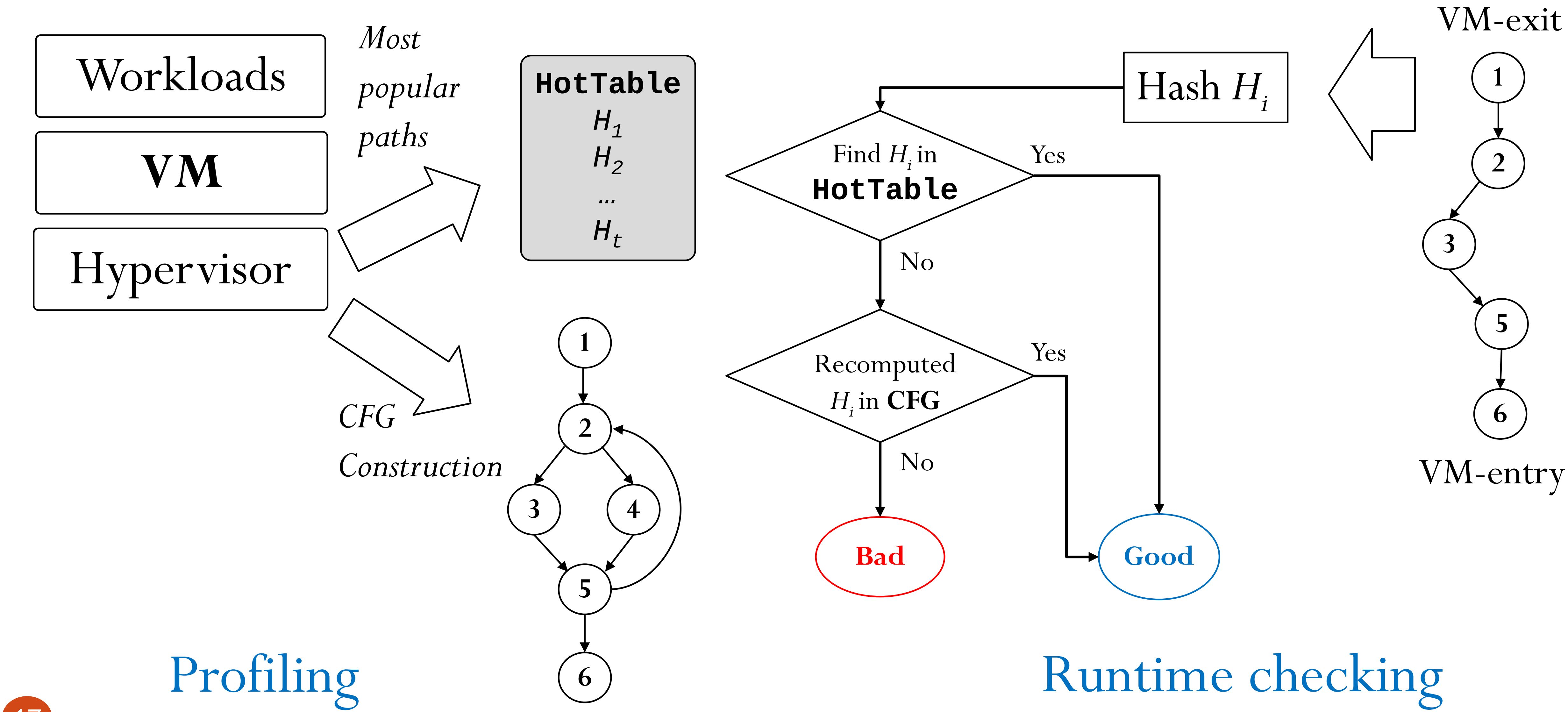
Control Flow Graph (CFG)

# hShield Approach

- ***Dynamic analysis* to construct CFG**
  - Full system coverage
  - Load-time binary rewriting compatible
- **Check at the sink of executions**
  - Reduce runtime overhead
- **Use hashes of basic blocks instead of addresses**
  - ASLR compatible



# hShield Approach



Workloads

**VM**

Hypervisor

*Most popular paths*

**`HotTable`**

$H_1$

$H_2$

…

$H_t$

*CFG Construction*

Profiling

VM-exit

VM-entry

Hash $H_i$

Find $H_i$ in **`HotTable`**

Yes

No

Recomputed $H_i$ in **CFG**

Yes

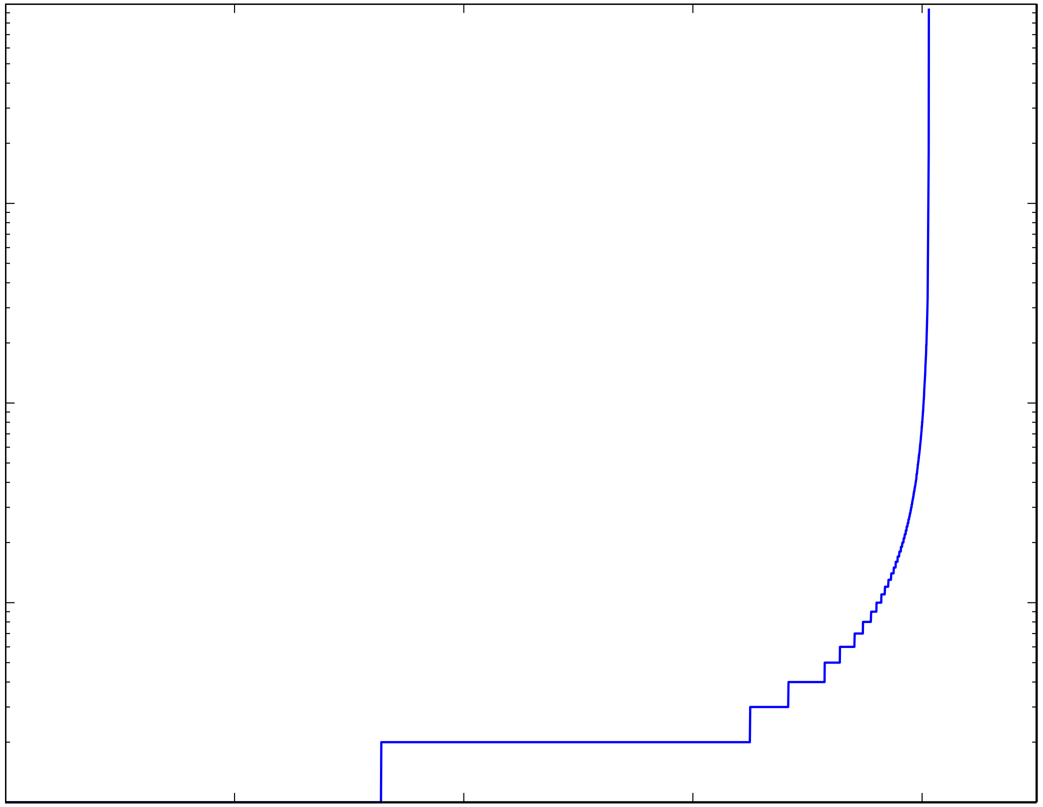
No

**Bad**

**Good**

Runtime checking

# Profiling Result: **Path Popularity**



**Setup**: Qemu (HW) – Qemu (Host) – Linux VM (Guest)

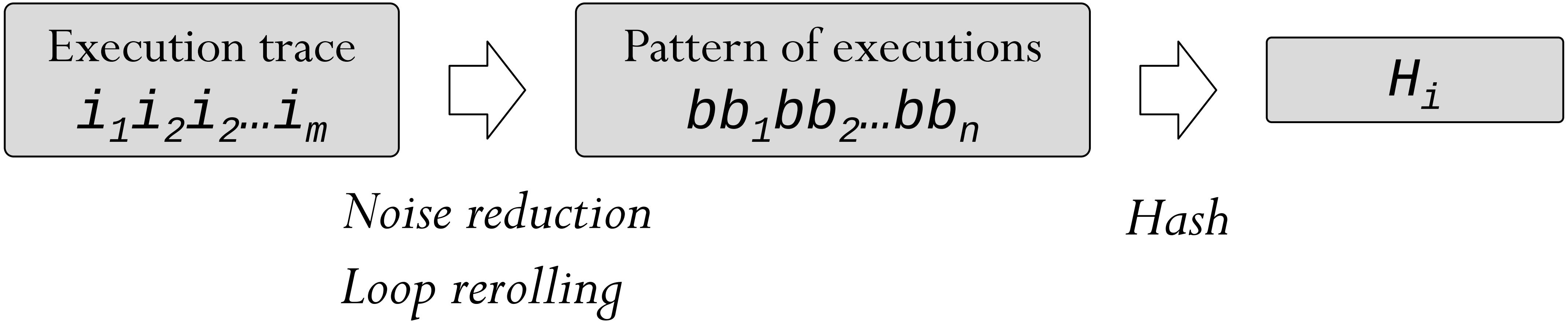
**Workloads**: Boot Linux kernel + UnixBench

**High hit rate of HotTable**

- 1% paths – 97% of exits
- 0.1% paths – 95% of exits

# Increase HotTable Hit Rate

- Execution Pattern Inference: Hash = a pattern of similar executions
  - Noise reduction (e.g., exclude interrupt handlers)
  - Loop rerolling



Execution trace

$i_1 i_2 i_2 \ldots i_m$

*Noise reduction*

*Loop rerolling*

Pattern of executions

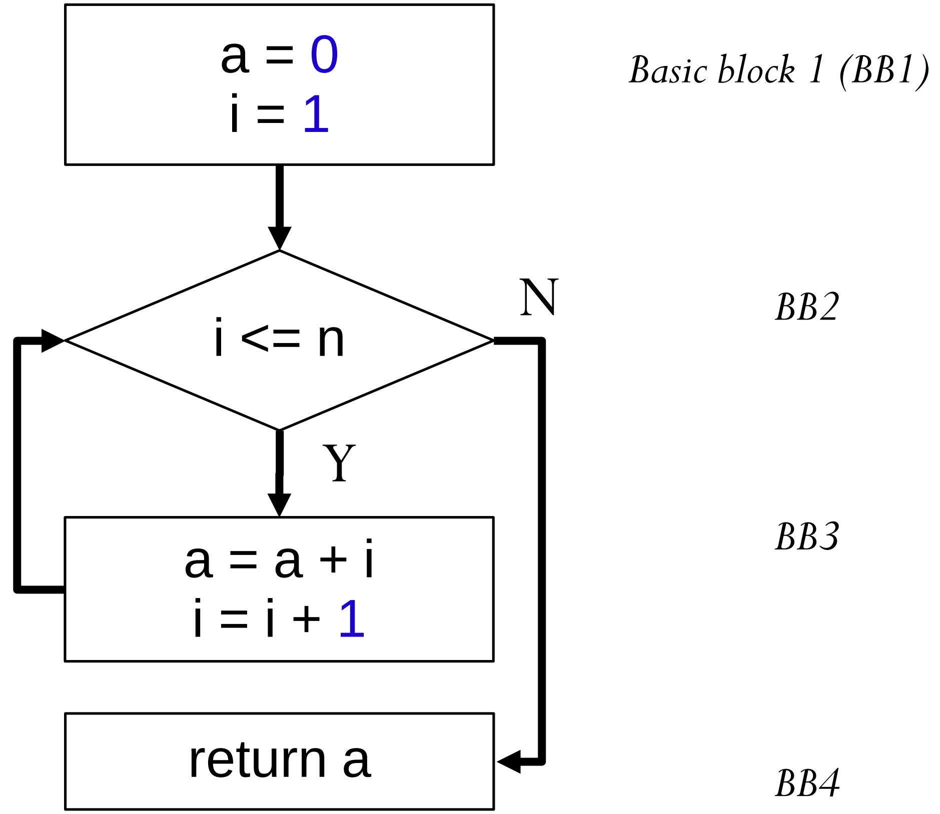
$bb_1 bb_2 \ldots bb_n$

*Hash*

$H_i$

# Loop Rerolling Example

```
a = 0;
for (i = 1..n)
    a = a + i;
return a;
```



BB1 BB2 BB4

BB1 ***BB2 BB3*** BB4

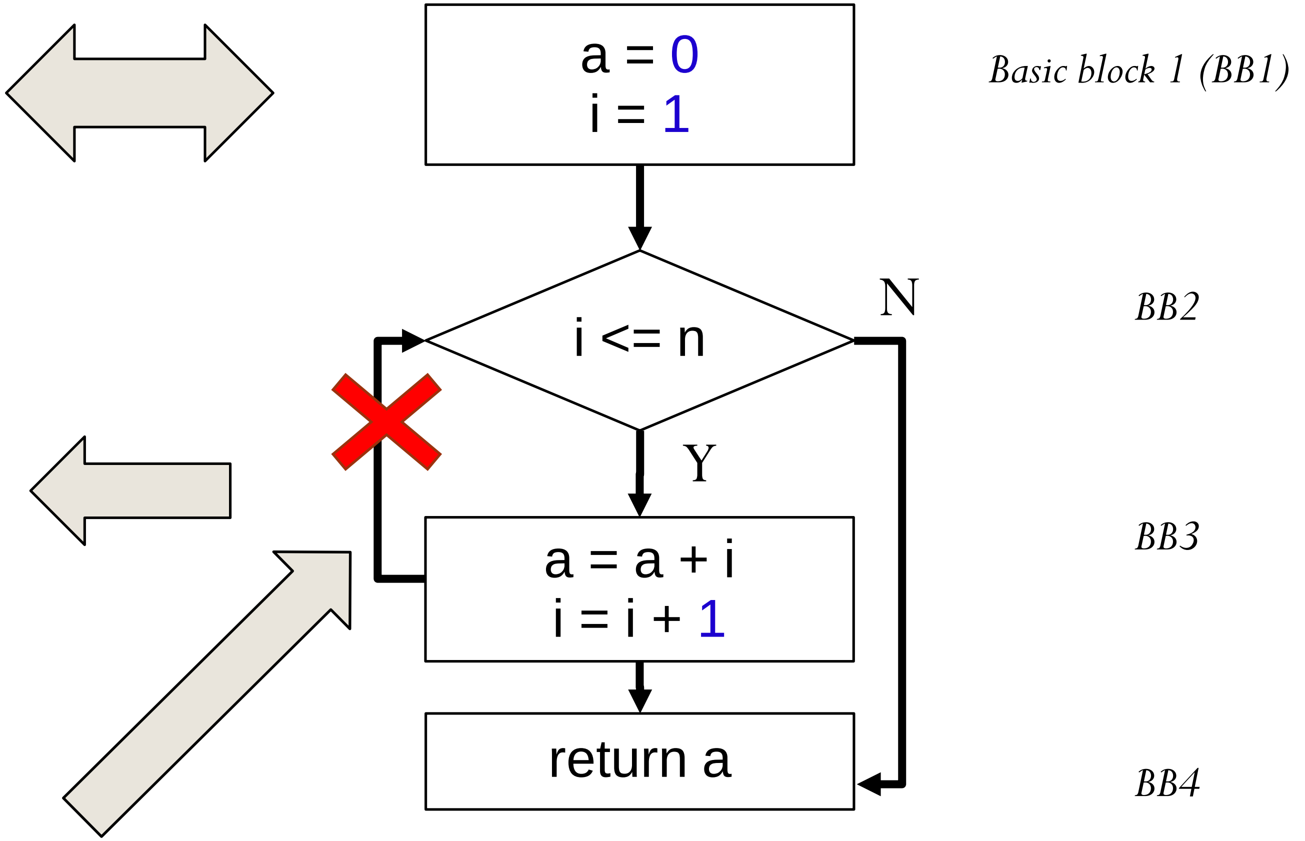
BB1 ***BB2 BB3 BB2 BB3*** BB4

…

`|Paths| = 1 + |Range(n)|`

# Loop Rerolling Example

```
a = 0;
for (i = 0..n)
    a = a + i;
return a;
```



BB1 BB2 BB4

BB1 ***BB2 BB3*** BB4

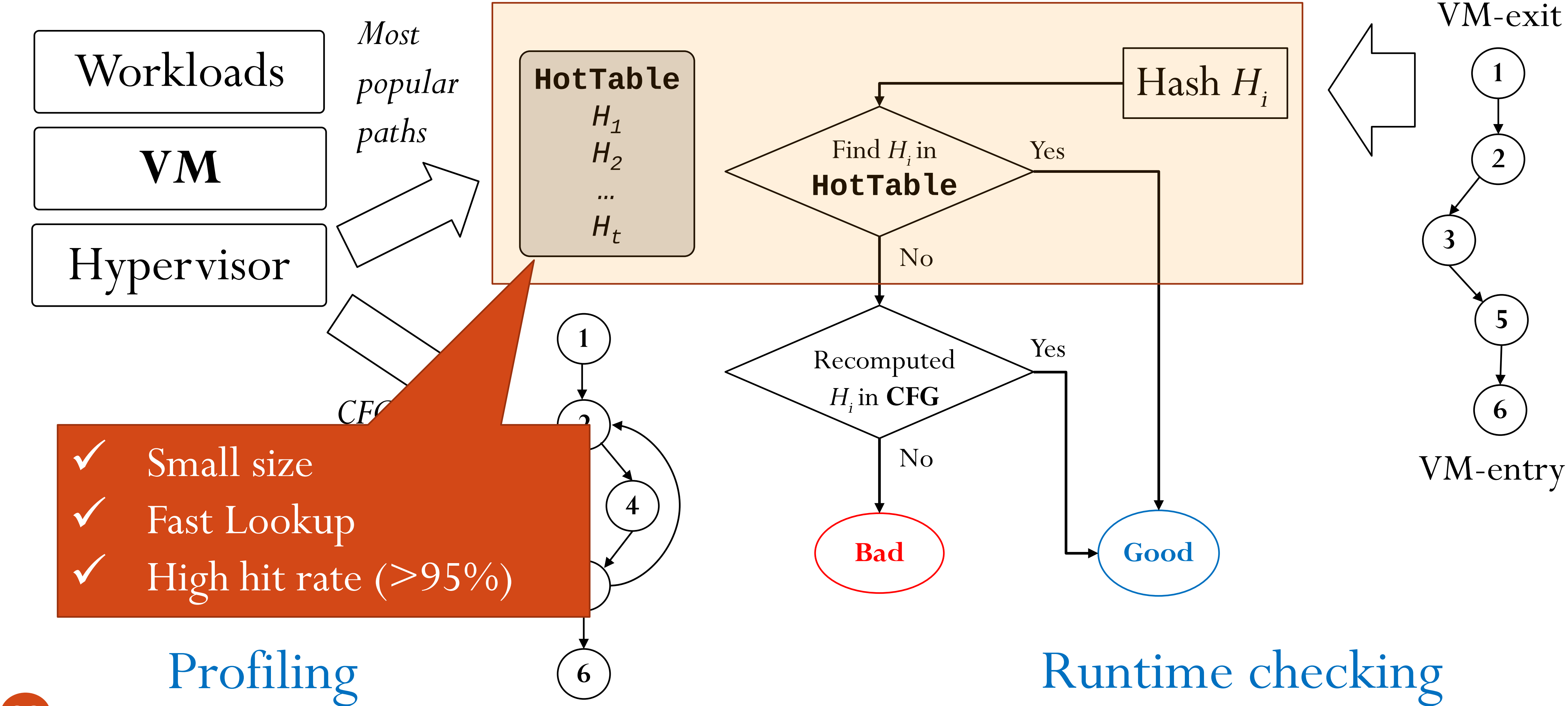
BB1 ***BB2 BB3 BB2 BB3*** BB4

…

`|Paths| = 2`

Solution: Loop rerolling

# hShield Approach

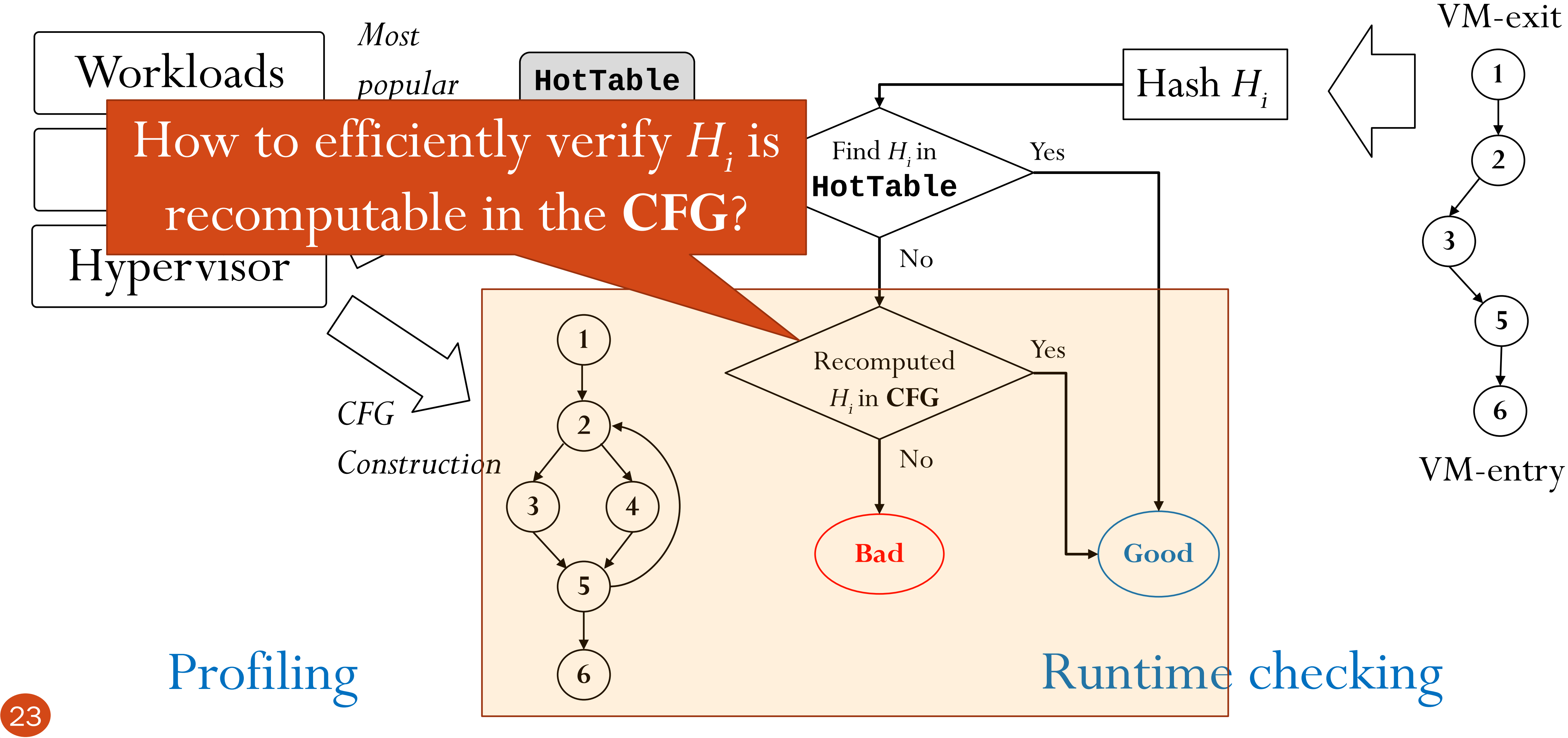


- ✓ Small size
- ✓ Fast Lookup
- ✓ High hit rate (>95%)

Profiling

Runtime checking

# hShield Approach



How to efficiently verify $H_i$ is recomputable in the **CFG**?

Workloads

*Most popular*

`HotTable`

Hypervisor

*CFG Construction*

Hash $H_i$

VM-exit

VM-entry

Find $H_i$ in **`HotTable`**

Yes

No

Recomputed $H_i$ in **CFG**

Yes

No

**Bad**

**Good**

Profiling

Runtime checking

# Execution Path Reconstruction

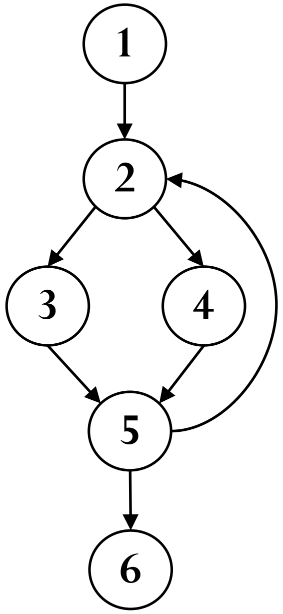
- Input Hash $H_i$ and CFG $G=<V, E>$
- Question: Exist a path $P \in G: Hash(P) = H_i$

- Naïve solution: Traverse $G$ until $P$ is found - impractical

- **hShield Solution**:
  - Using incremental hashing to efficiently **reconstruct** $P$ **from** $H_i$ **and** $G$

# Execution Path Reconstruction

**Source**



**Sink**

Is ① first basic block?

Yes, *update*(① , $H_i$) == $H_i$

Is ② first basic block?

Yes, *update*(② , $H_i$) == $H_i$

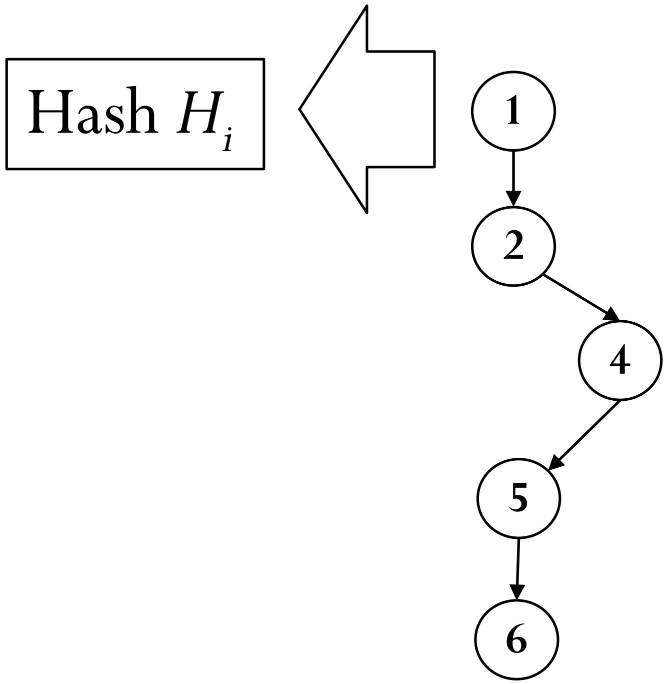
Is ③ first basic block?

No, *update*(③ , $H_i$) != $H_i$

Is ④ first basic block?

Yes, *update*(④ , $H_i$) == $H_i$

…. end at ⑥


Hash $H_i$

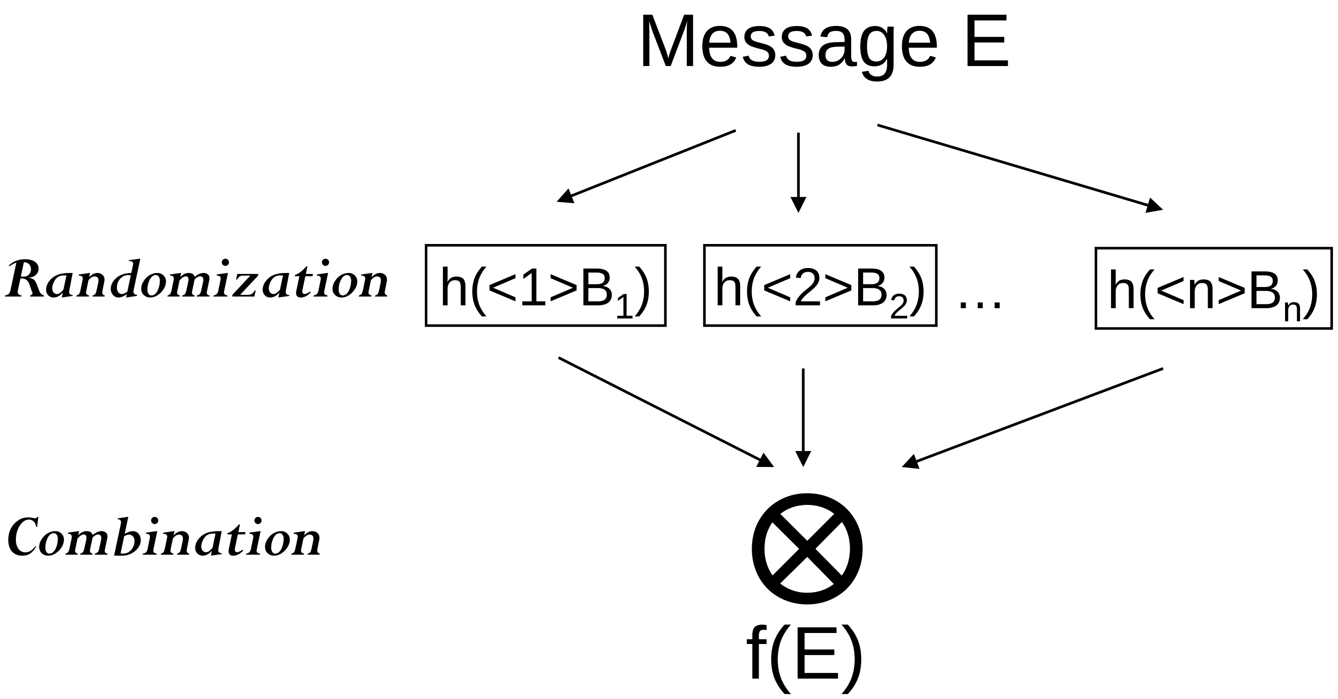
# Execution Representation

$$f: Exe \rightarrow Range$$

- $E \in Exe: E = I_1 I_2 .. I_n$
  - $I_i$: Instruction byte code (e.g., x86)
- *Range*: Fixed length output

- $f$ requirements
  - Collision resistant
  - Interactive – online construction
  - Incremental – online update
  - Facilitate loop rerolling implementation

# Incremental Collision-free Hashing [1]



Message E

***Randomization*** h(<1>$B_1$) h(<2>$B_2$) … h(<n>$B_n$)

***Combination*** $\otimes$

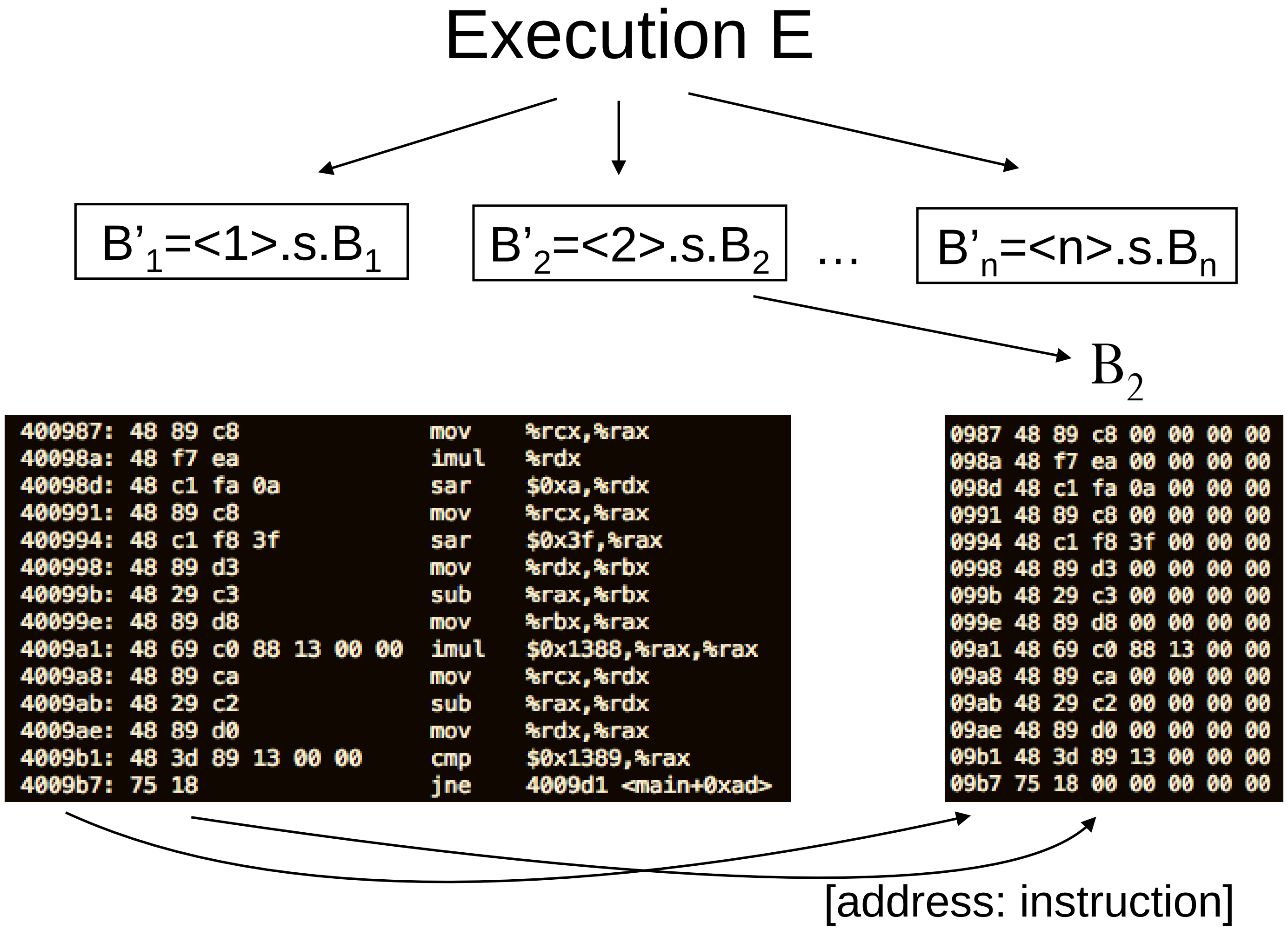
f(E)

- ***Randomization*** – Derived from standard cryptographic functions (e.g., SHA, MD5)
- ***Combination*** – Algebraic operation
  - *Incrementality*: Allow *update* results when a portion of input changed without re-computing from scratch
- ***Collision-free*** [1]

---

[1] M. Bellare and D. Micciancio, “A new paradigm for collision-free hashing: Incrementality at reduced cost,” in *Advances in Cryptology EU- ROCRYPT97*. Springer, 1997, pp. 163–192.

# hShield Counter Hash Function



Execution E

$B'_1=\langle 1\rangle.s.B_1$ $B'_2=\langle 2\rangle.s.B_2$ … $B'_n=\langle n\rangle.s.B_n$

$B_2$

```
400987: 48 89 c8                mov    %rcx,%rax
40098a: 48 f7 ea                imul   %rdx
40098d: 48 c1 fa 0a             sar    $0xa,%rdx
400991: 48 89 c8                mov    %rcx,%rax
400994: 48 c1 f8 3f             sar    $0x3f,%rax
400998: 48 89 d3                mov    %rdx,%rbx
40099b: 48 29 c3                sub    %rax,%rbx
40099e: 48 89 d8                mov    %rbx,%rax
4009a1: 48 69 c0 88 13 00 00    imul   $0x1388,%rax,%rax
4009a8: 48 89 ca                mov    %rcx,%rdx
4009ab: 48 29 c2                sub    %rax,%rdx
4009ae: 48 89 d0                mov    %rdx,%rax
4009b1: 48 3d 89 13 00 00       cmp    $0x1389,%rax
4009b7: 75 18                   jne    4009d1 <main+0xad>
```

```
0987 48 89 c8 00 00 00 00
098a 48 f7 ea 00 00 00 00
098d 48 c1 fa 0a 00 00 00
0991 48 89 c8 00 00 00 00
0994 48 c1 f8 3f 00 00 00
0998 48 89 d3 00 00 00 00
099b 48 29 c3 00 00 00 00
099e 48 89 d8 00 00 00 00
09a1 48 69 c0 88 13 00 00
09a8 48 89 ca 00 00 00 00
09ab 48 29 c2 00 00 00 00
09ae 48 89 d0 00 00 00 00
09b1 48 3d 89 13 00 00 00
09b7 75 18 00 00 00 00 00
```

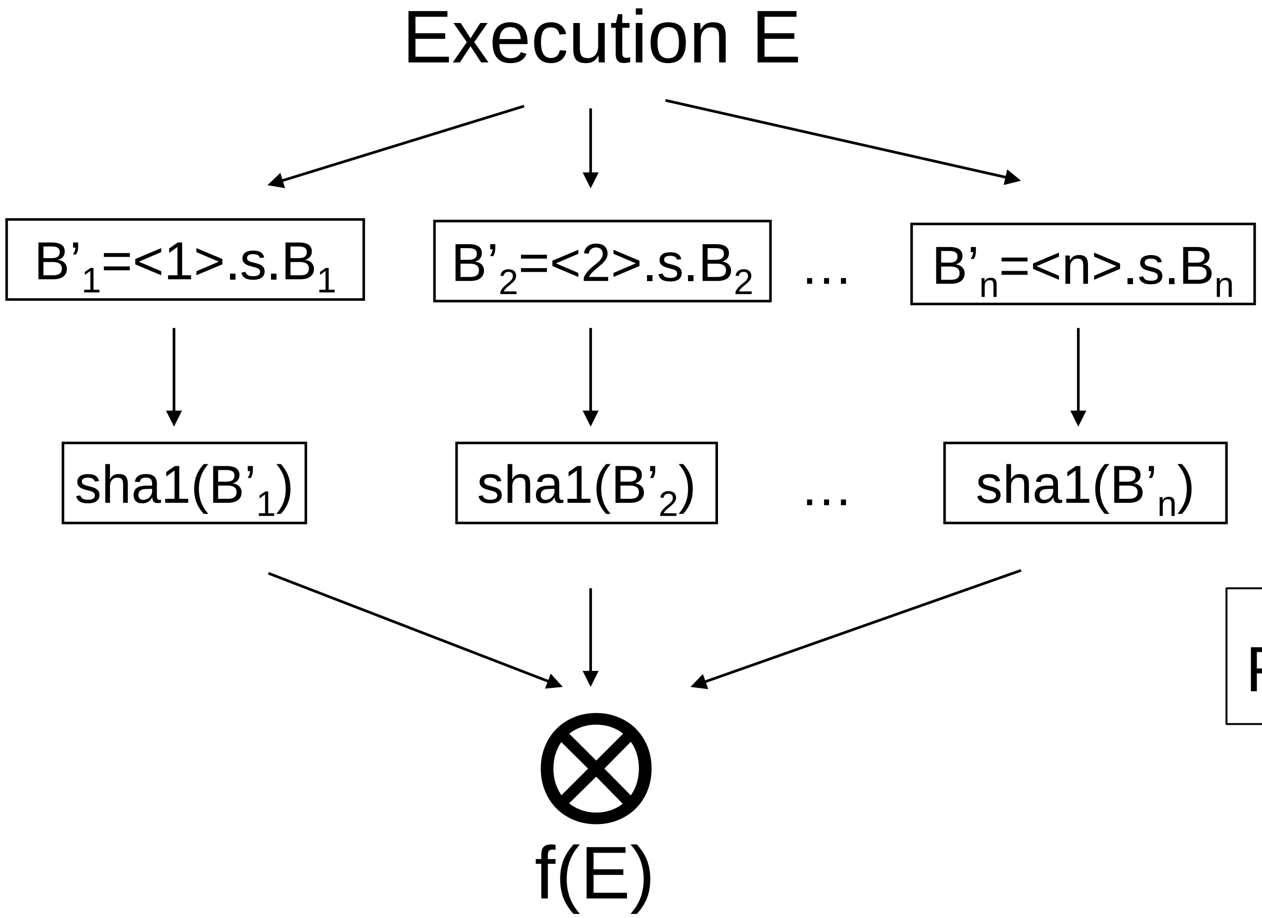
[address: instruction]

$$\mathbf{y_2 = sha1(salt.<2>.b_2)}$$

⇓

$$\mathbf{MuHASH(y_1, y_2) = y_1 \otimes y_2}$$

# hShield Counter Hash Function



Execution E

$B'_1 = \langle 1 \rangle.s.B_1$ $B'_2 = \langle 2 \rangle.s.B_2$ … $B'_n = \langle n \rangle.s.B_n$

$\mathrm{sha1}(B'_1)$ $\mathrm{sha1}(B'_2)$ … $\mathrm{sha1}(B'_n)$

$\otimes$

f(E)

- B = Basic block – facilitate loop rerolling
- S: salt – individualize target
- $\otimes$: modular multiplication (MuHash)

$$F(E) = \bigotimes_{i=1..n}(\mathrm{sha1}(\langle i \rangle.s.B_i))$$

# Security Evaluation

| | E | h | ⊗ | salt | F(E) |
|---|---|---|---|---|---|
| Know | ✓ | ✓ | ✓ | ✓ | ✓ |
| Change | ✓ | × | × | × | × |

## Attack Model

- Change execution E —> E': f(E') ∈ F(E)

## Solution

- Find E' complexity = Discrete Log problem (assuming h is ideal)
- Must harder to find an E' which is valid x86 code

# Scopes

- **Focus on design of the monitoring framework**
  - What/Where/How to monitor *(Answers:VM Exit/Hardware/Whitelist)*
- **Design for flexible future hardware implementation**
  - Make best effort to conduct measurement on actual hardware
    - *E.g., obtained supporting data on physical systems.*
  - Make best effort to anticipate problems in actual hardware implementation.
    - *E.g., issues with speculative execution, memory size constraints.*
  - Assume that we can place hooks in some basic signals in the hardware.
    - *E.g., intercept all interrupts and exceptions.*
- **Prototype the proposal in QEMU**
  - Software emulation of x86 processors and many external devices.

# More to come...

- Archirectural Design
- Evaluation with real attacks
- Performance evaluation