Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection

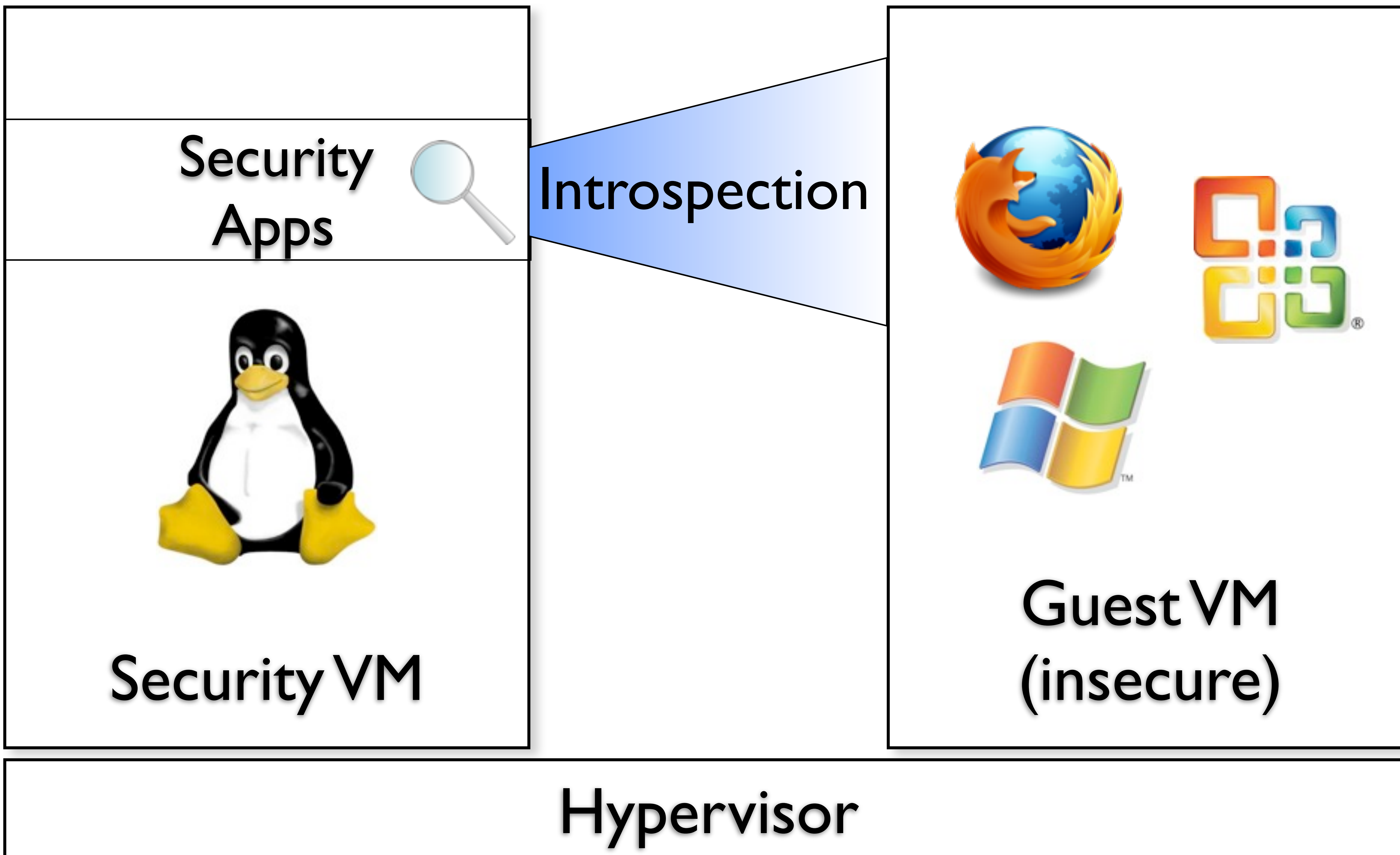
Brendan Dolan-Gavitt*, Tim Leek†, Michael Zhivich†, Jonathon Giffin*, and Wenke Lee*

* Georgia Institute of Technology
† MIT Lincoln Laboratory

{brendan,giffin,wenke}@cc.gatech.edu
{tleek,mzhivich}@ll.mit.edu

This work was sponsored by IARPA under Air Force Contract FA8721-05-C-0002. Opinions, interpretation, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Virtual Machine Introspection



Standard virtualization security layout: insecure guests, isolated security apps in their own VM.

Open Problem: The Semantic Gap

- Isolation can provide security
- Isolation makes it hard to see what's going on
- View exposed by VMM is low-level (physical memory, CPU state)
- Need to reconstruct high-level view using *introspection routines*

Isolation is not a panacea; it makes it hard to see what's going on. The view the VMM gives is not what we want.

What You Want...

Files

Processes

Networking

Drivers

These are the objects relevant to security monitoring.

What You Get


```
00483a0: 7828 837f 1400 750c ff74 240c 57e8 a2ba ffff eb24 57e8 38ba ffff 8b4c 240c 32d2  x(....u..t$.W......$W.8....L$.2.
00483c0: c741 1810 0000 c0ff 158c a056 f757 e845 baff ffb8 1000 00c0 5fc2 0800 558b ec53  .A.........V.W.E........_...U..S
00483e0: 8b5d 0883 a3c8 0000 0000 53e8 b0b4 ffff 8d45 0850 53e8 f8ed ffff 85c0 0f8c bb00  .]........S......E.PS...........
0048400: 0000 568b 7508 578d bbd0 0000 00a5 a56a 00ff 7508 a5ff 1524 a156 f753 e85f b2ff  ..V.u.W........j..u....$.V.S._..
0048420: ff53 e893 0600 0085 c0be 2502 00c0 bf34 0000 c07d 083b c674 043b c775 7e53 e881  .S........%....4...}.;.t.;.u~S..
0048440: c6ff ff85 c07c 7453 e8b3 ebff ff85 c07c 07e8 0cad ffff eb08 3bc6 7404 3bc7 755b  .....|tS.......|........;.t.;.u[
0048460: 53e8 e8eb ffff 3dbb 0000 c075 0c53 e851 c6ff ff85 c07d 0ceb 4285 c07d 0653 e8c3  S.....=....u.S.Q.....}..B..}.S..
0048480: ebff ff83 bbc8 0000 0000 7406 53e8 5004 0000 53e8 e0fd ffff 8bf0 85f6 7c1b 53e8  ..........t.S.P...S.........|.S.
00484a0: f6b1 ffff 8b43 0c89 4310 33c0 4053 8943 0c89 4314 e8eb b9ff ff8b c65f 5e5b 5dc2  .....C..C.3.@S.C..C........_^[].
00484c0: 0800 5072 6f63 6573 736f 7220 6472 6976 6572 2064 6f65 7320 6e6f 7420 7375 7070  ..Processor driver does not supp
00484e0: 6f72 7420 4952 505f 4d4e 5f53 5552 5052 4953 455f 5245 4d4f 5641 4c0a 00cc 558b  ort IRP_MN_SURPRISE_REMOVAL...U.
0048500: ec83 ec50 a100 b056 f753 8945 fc8b 4508 568b 7028 837e 0c06 578b 7d0c 8b5f 6075  ...P...V.S.E..E.V.p(.~..W.}..._`u
0048520: 19be 5600 00c0 32d2 8bcf 8977 18ff 158c a056 f78b c6e9 3601 0000 56e8 b2b8 ffff  ..V...2....w.....V....6...V.....
0048540: 0fb6 4301 83f8 057f 7874 5083 e800 7455 4874 4848 7432 4874 0d48 7579 b810 0000  ..C.....xtP...tUHtHHt2Ht.Huy....
0048560: c0e9 0a01 0000 837e 0c04 57ff 7608 0f85 b600 0000 e891 fcff ff8b d885 db0f 8cd8  .......~..W.v...................
0048580: 0000 00e9 c000 0000 57ff 7608 e879 fcff ff8b d885 db0f 8dc0 0000 00bb 1000 00c0  ........W.v..y..................
00485a0: e9b6 0000 0057 ff76 08e8 5cfc ffff 8bd8 85db 0f8c a300 0000 5756 e81d feff ffeb  .....W.v..\...............WV......
00485c0: 6e83 f806 746d 83f8 0974 5b83 f814 7456 83f8 1774 1efe 4723 8347 6024 8b4e 088b  n...tm...t[...tV...t..G#.G`$.N..
00485e0: d7ff 1510 a156 f756 8bf8 e829 b8ff ff8b c7eb 7d83 65b0 0033 c08d 7db4 abab 6a0e  .....V.V...)......}.e..3..}...j.
0048600: ab59 bec2 d456 f78d 7dc0 f3a5 6a00 8d45 c050 8d45 b050 66a5 6800 0001 006a 4ca4  .Y...V..}...j..E.P.E.Pf.h....jL.
0048620: ff15 18a1 56f7 57ff 7608 e8db fbff ff8b d8eb 2857 ff76 08e8 cefb ffff 8bd8 85db  ....V.W.v.......(W.v............
0048640: 7c19 837e 0c02 7513 8b46 1056 8946 0cc7 4614 0100 0000 e849 b8ff ff32 d28b cf89  |..~..u..F.V.F..F.........2....
0048660: 5f18 ff15 8ca0 56f7 56e8 aab7 ffff 8bc3 8b4d fc5f 5e5b e86d c6ff ffc9 c208 00cc  _.....V.V........M._^[.m........
0048680: 558b ec53 8b5d 0856 578b 7b28 57e8 60b7 ffff 837f 0c06 7515 8b4d 0cbe 5600 00c0  U..S.].VW.{(W.`.......u..M..V...
00486a0: 32d2 8971 18ff 158c a056 f7eb 458b 750c 8d45 0850 5653 8d87 9800 0000 50e8 70c6  2..q.....V..E.u..E.PVS......P.p.
00486c0: ffff 8bd8 8b45 0885 c074 2583 f801 7416 fe46 2383 4660 248b 4f08 8bd6 ff15 10a1  .....E...t%...t..F#.F`$.O.......
00486e0: 56f7 8bd8 eb0a 32d2 8bce ff15 8ca0 56f7 8bf3 57e8 20b7 ffff 5f8b c65e 5b5d c208  V.....2.......V...W. ..._..^[]..
0048700: 00cc 837c 240c 0176 0783 7c24 0c03 7507 b8c6 0200 c0eb 05b8 9502 00c0 6a00 6a00  ...|$..v..|$..u.............j.j.
0048720: 50ff 7424 14ff 7424 14e8 0ac6 ffff c21c 00cc 837c 240c 0176 0783 7c24 0c03 7507  P.t$..t$..........|$..v..|$..u.
0048740: b8c6 0200 c0eb 05b8 9502 00c0 6a00 6a00 50ff 7424 14ff 7424 14e8 dac5 ffff c218  ............j.j.P.t$..t$........
0048760: 00cc 558b ec51 8b45 0853 8b58 288b 4510 5633 f62b c657 8975 fc74 6048 7455 4848  ..U..Q.E.S.X(.E.V3.+.W.u.t`HtUHH
0048780: 0f85 e600 0000 39b3 c800 0000 0f84 da00 0000 6a0c 5839 4520 8945 fc72 2c53 e8dd  ......9..........j.X9E .E.r,S..
00487a0: aeff ff8b 8304 0100 008b 8bc8 0000 008b 7d24 8d04 408d 7481 10a5 a553 a5e8 d8ae  ................}$..@.t....S....
00487c0: ffff 33f6 e9a8 0000 00be 2300 00c0 e99e 0000 0089 75fc e996 0000 0039 b3c8 0000  ..3.......#.........u......9....
00487e0: 000f 8485 0000 0053 e893 aeff ff8b 83c8 0000 008b 400c 488d 0440 c1e0 028d 481c  .......S............@.H..@....H.
0048800: 83c0 3439 4520 8945 fc72 548b 9304 0100 008b 4524 8910 8b93 0c01 0000 8950 048b  ..49E .E.rT.......E$.........P..
0048820: 9310 0100 0089 5008 8b93 0801 0000 8950 0c8b 93fc 0000 0089 5010 0fb6 9300 0100  ......P........P........P.......
0048840: 0089 5014 8bb3 c800 0000 8d78 188b c1c1 e902 f3a5 8bc8 83e1 03f3 a433 f6eb 05be  ..P........x................3....
0048860: 2300 00c0 53e8 30ae ffff eb05 be95 0200 c08b 45fc 8b4d 1c6a 0050 56ff 750c 8901  #...S.0..........E..M.j.PV.u....
```


This is the view exposed by the VMM -- physical memory. This is from a memory dump of a Windows 2003 system.

Introspection Challenges

- Introspection routines are currently built *manually*
- Building routines requires detailed knowledge of OS internals
  - Often requires reverse engineering
- OS updates and patches break existing introspection utilities

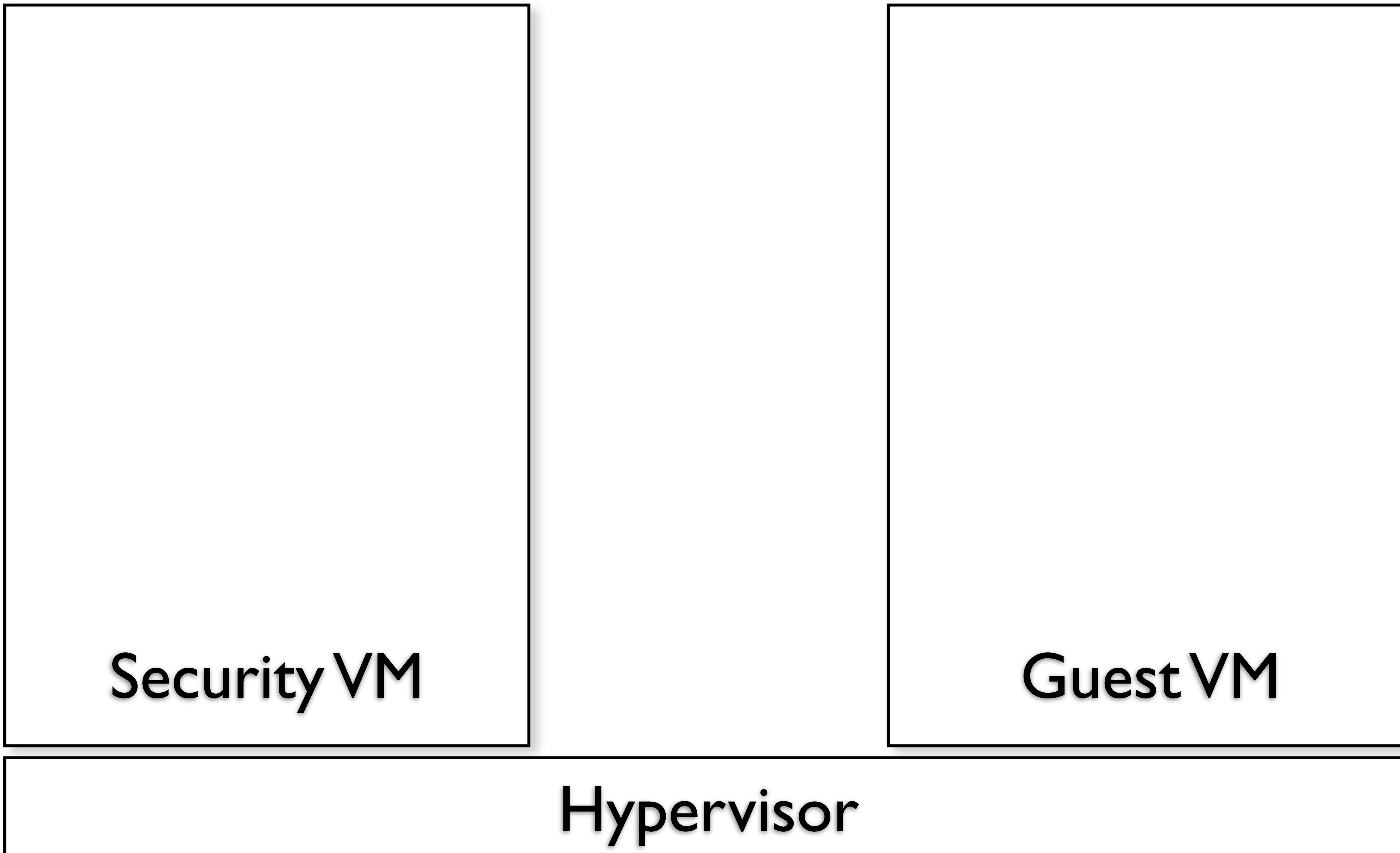
Note story about sec. vendor that had to spend 60 hours reverse engineering Vista's new TCP/IP stack. Virtuoso can reduce this to a few minutes by a non-expert.

Contributions

- We generate introspection routines *automatically*
- No knowledge of OS internals or reverse engineering required
- Routines can be regenerated easily for new OS versions / patches

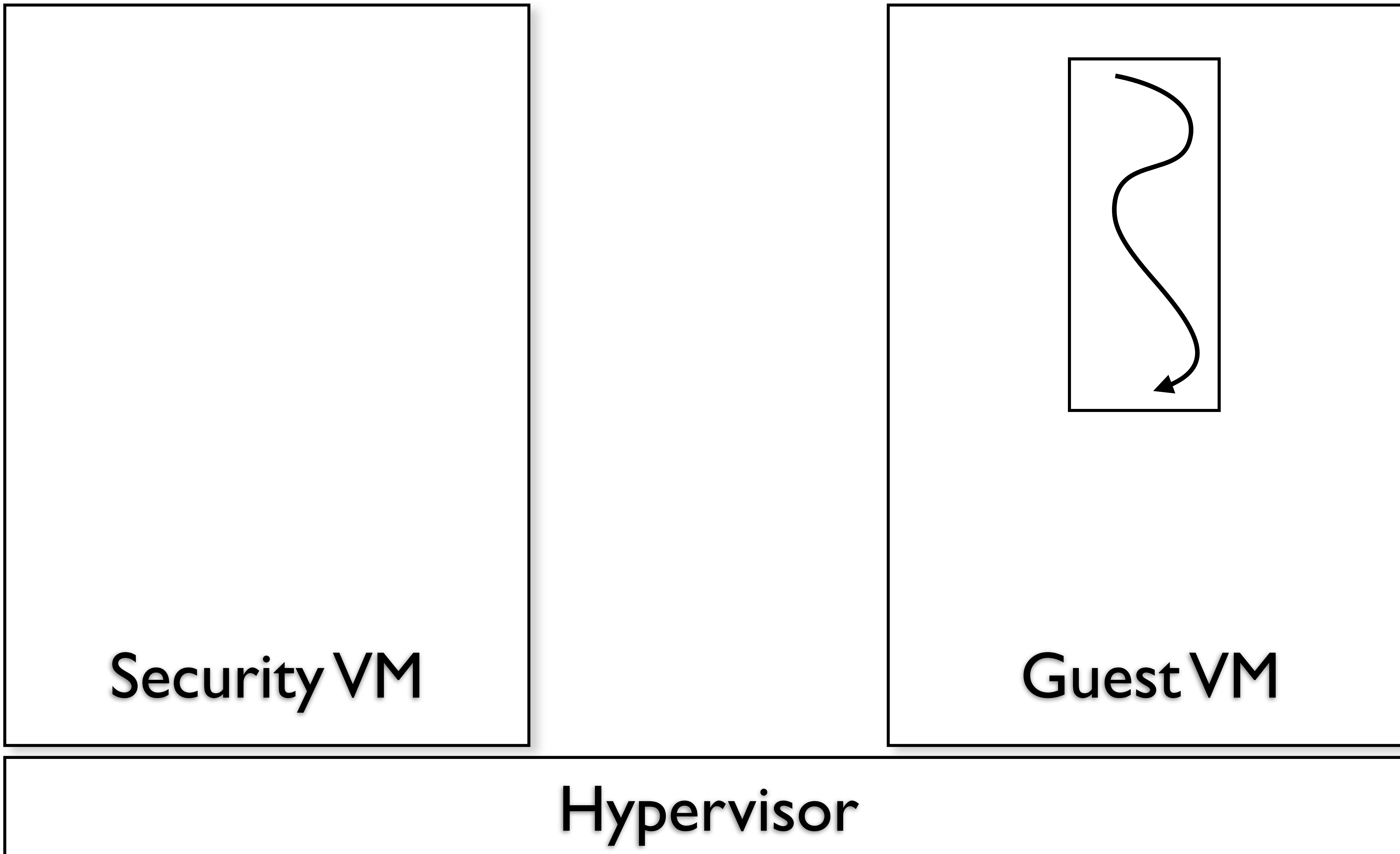
Only programmer's knowledge of public system APIs needed.

Idea: Code Extraction



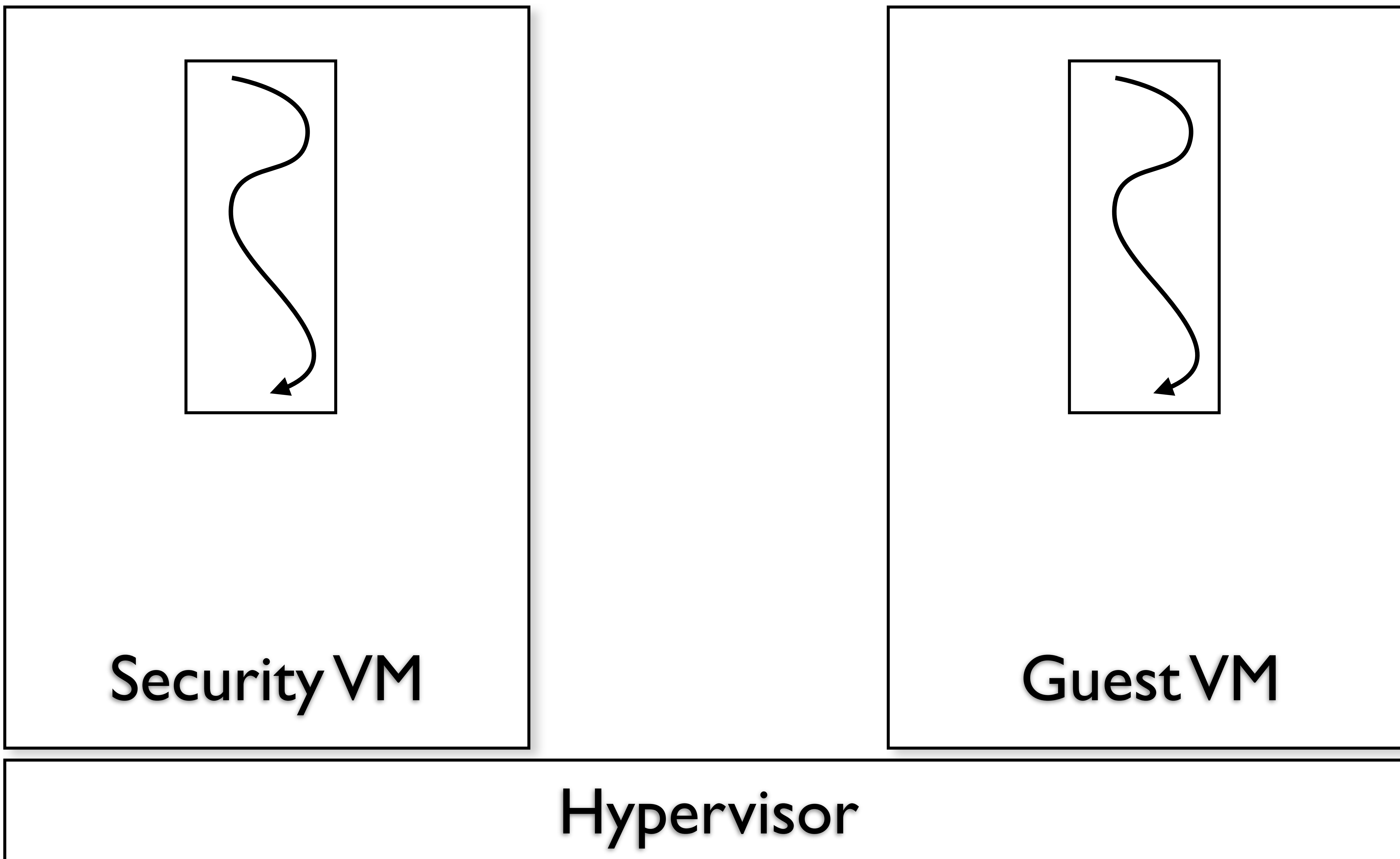
Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.

Idea: Code Extraction



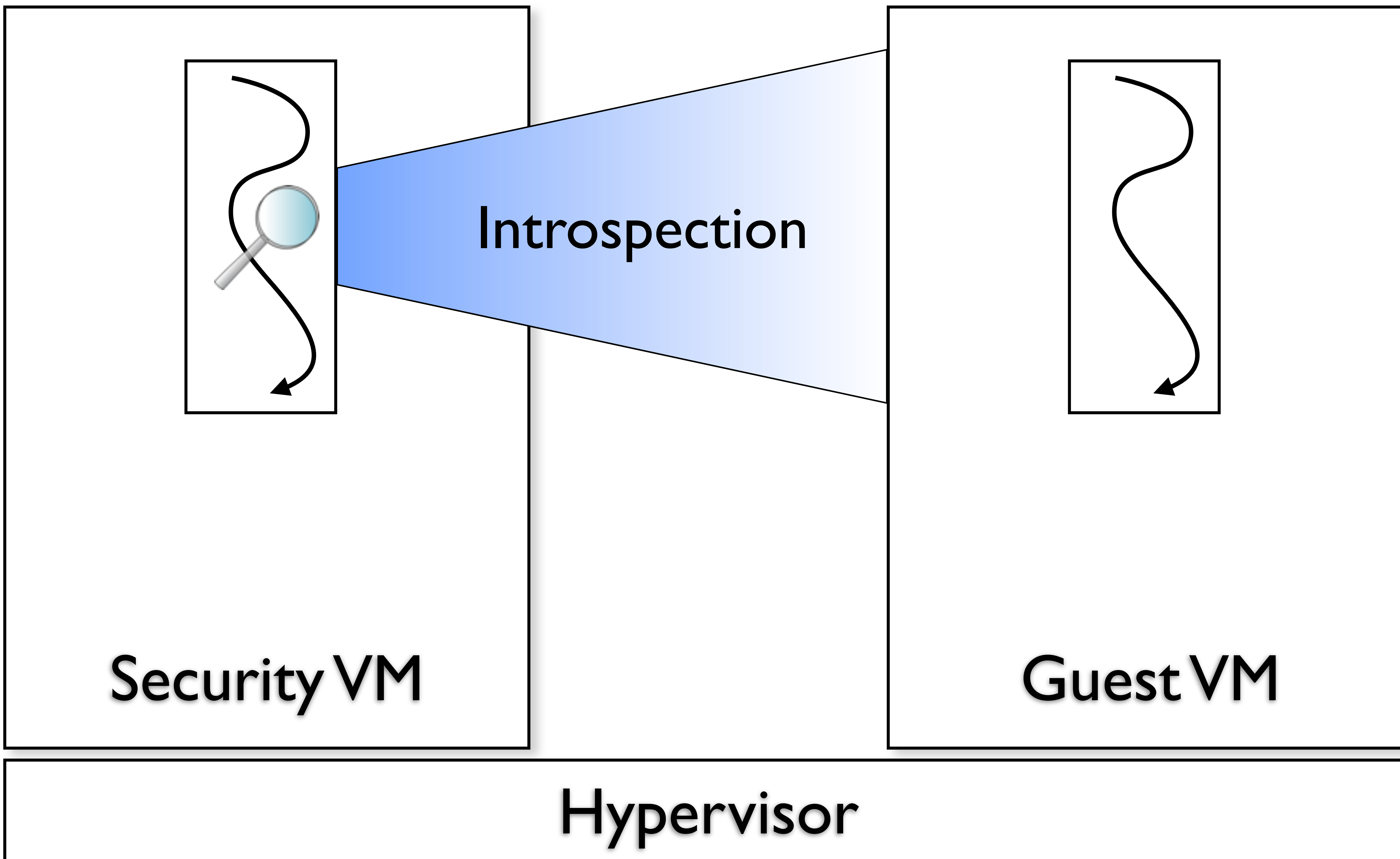
Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.

Idea: Code Extraction



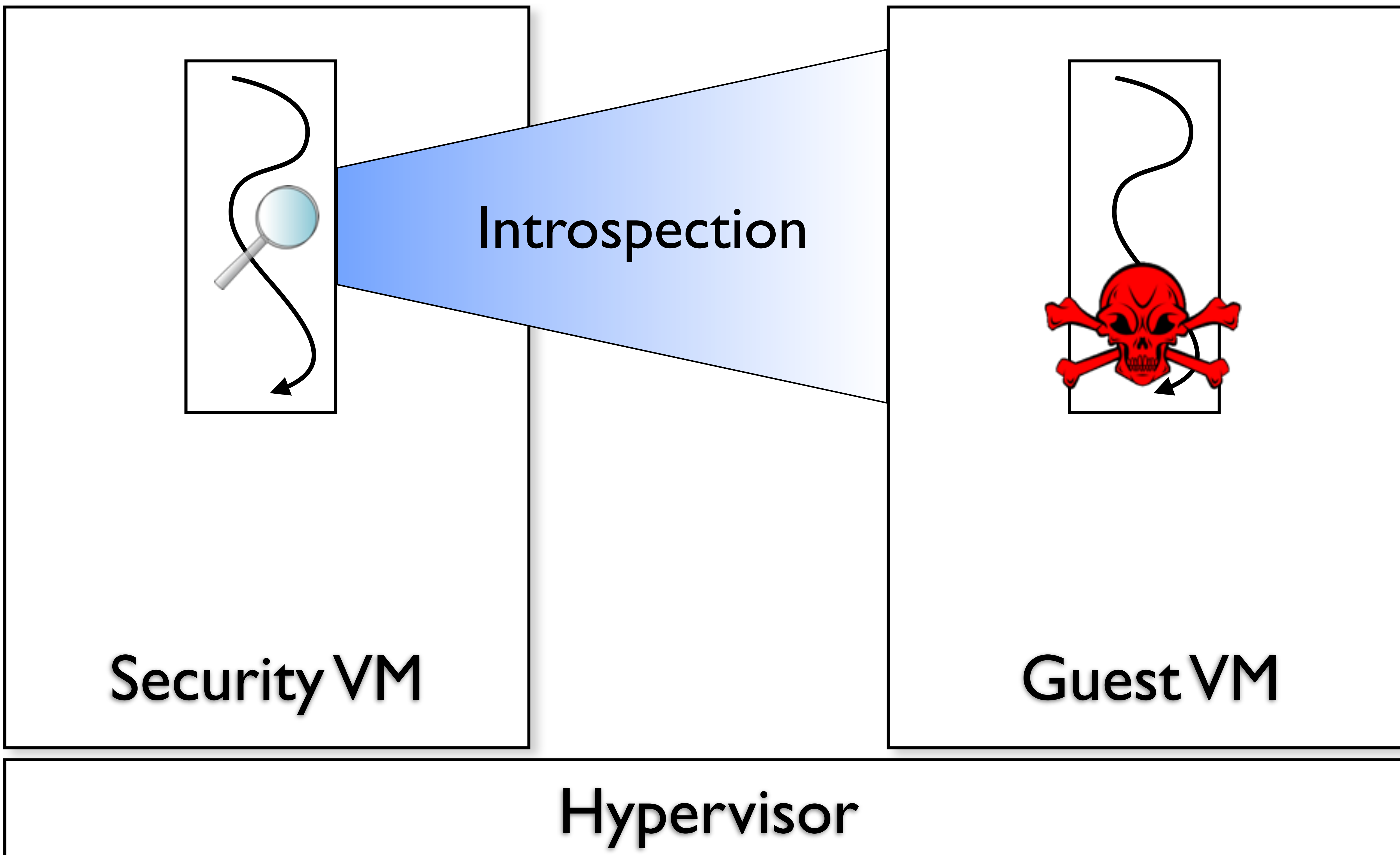
Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.

Idea: Code Extraction



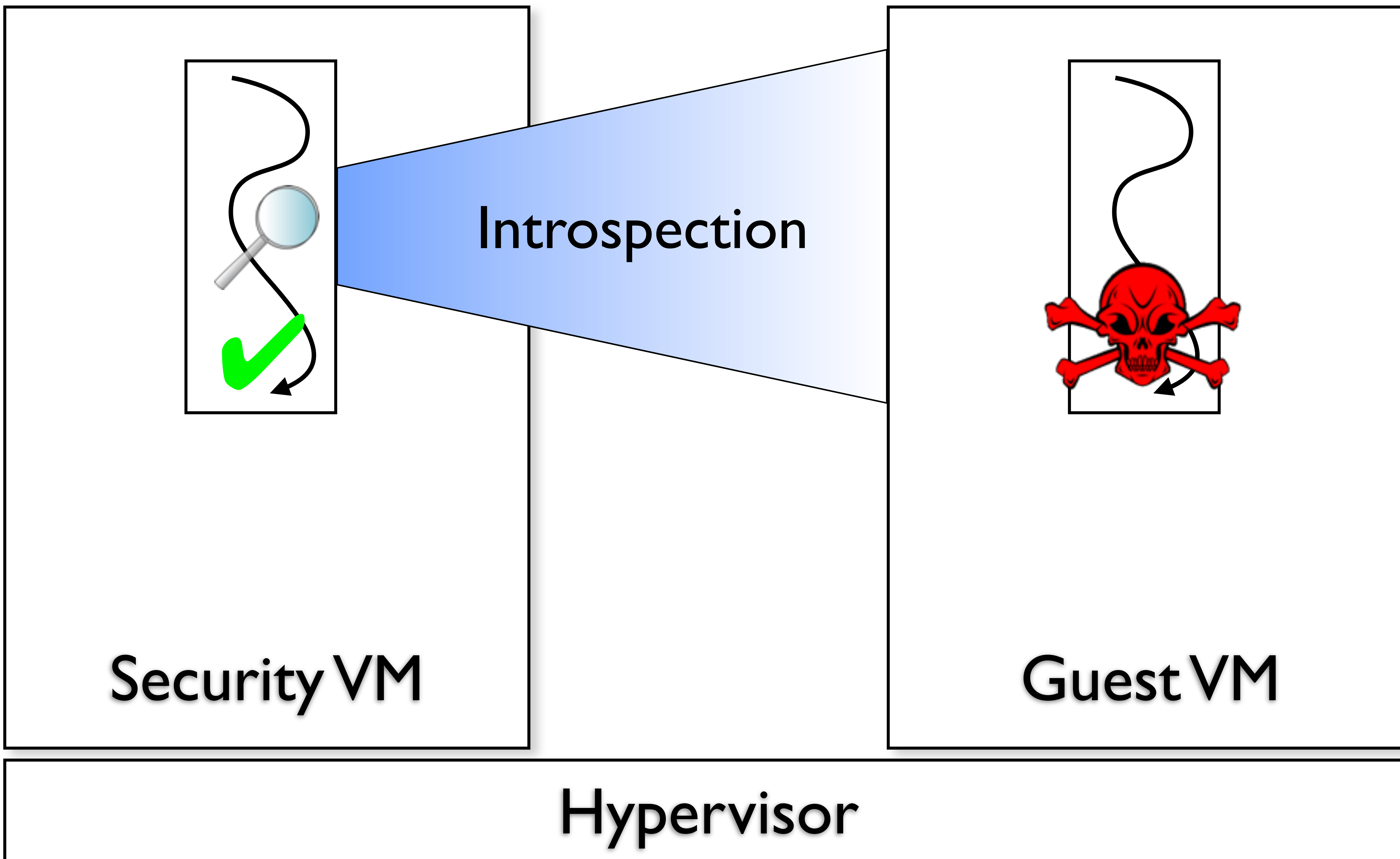
Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.

Idea: Code Extraction



Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.

Idea: Code Extraction



Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.

Goals

- Generality: generate useful introspection programs on multiple operating systems
- Reliability: generate working programs using dynamic analysis
- Security: ensure that programs are unaffected by guest compromise

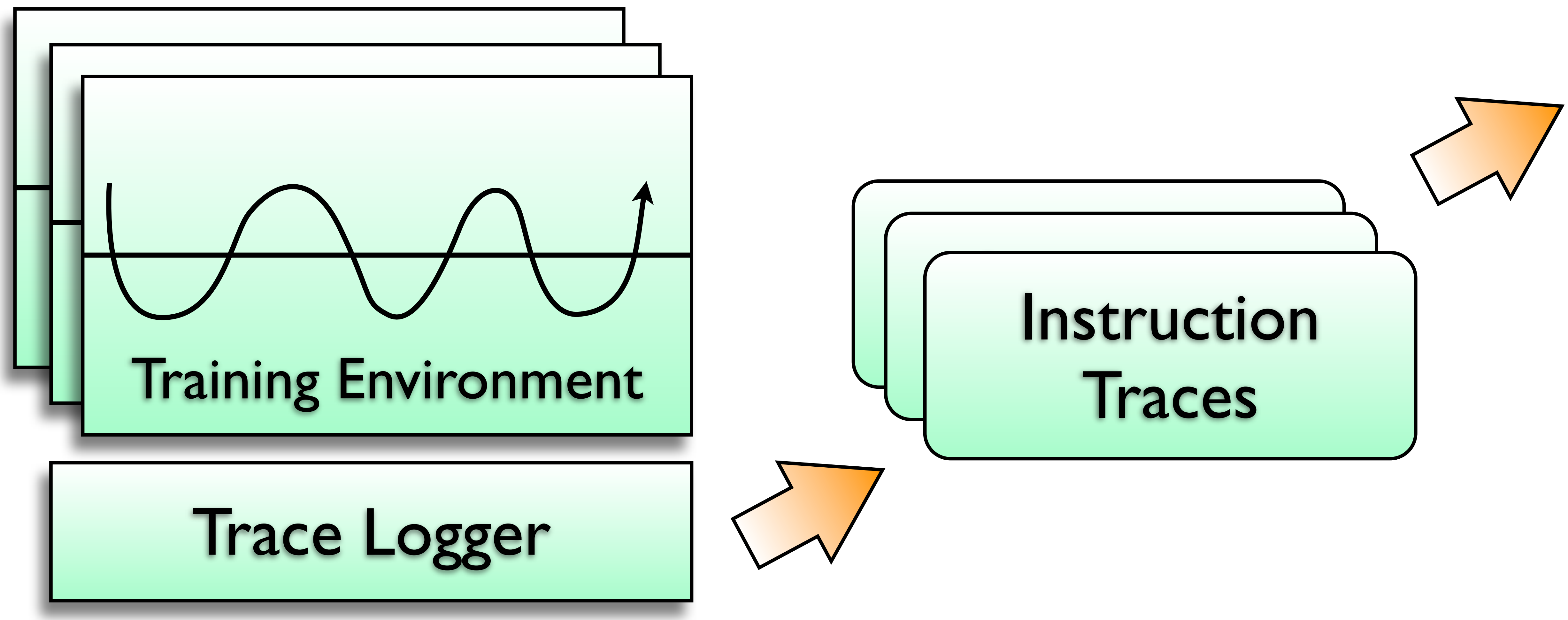
These goals mirror the results we present later on.

Challenges

- Assume no prior knowledge of OS internals
- Code extraction must be *whole-system*
  - Much of the code we want is in the kernel
  - Existing work (BCR, Inspector Gadget) only extracts small pieces of userland code

We don't quite get to no a priori knowledge, but close (mallocs).

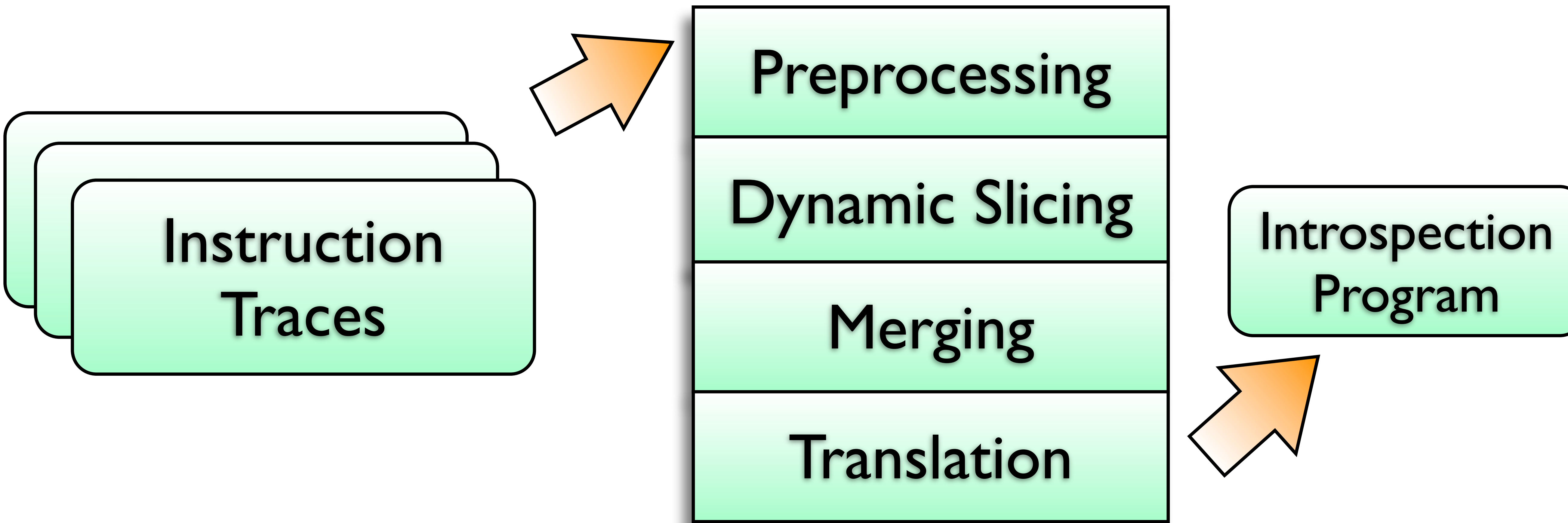
Overview



Training Phase

Make clear here that we write a small in-guest program that gets the data we want!

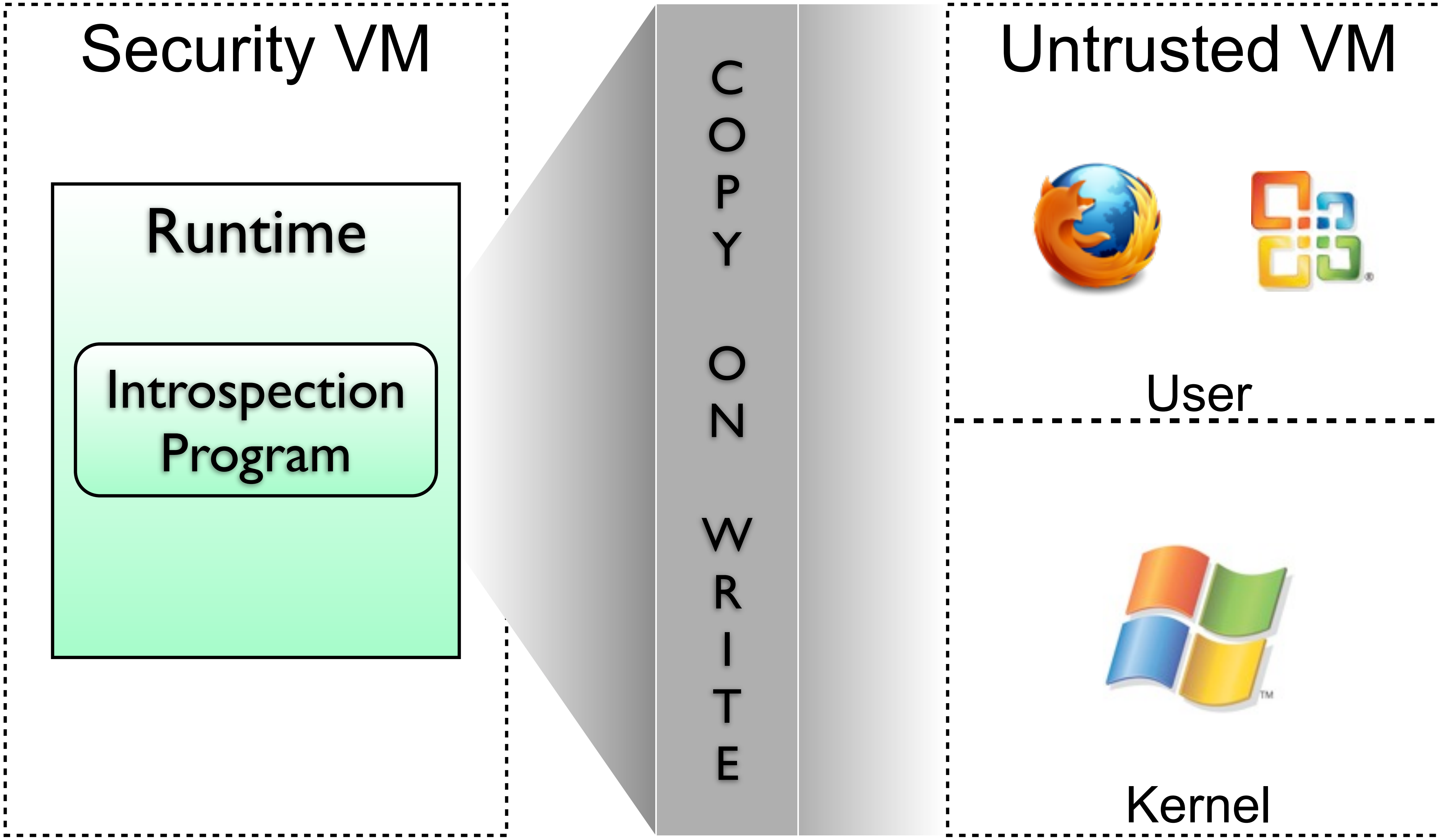
Overview



Analysis Phase

Once we have the traces, we process them and translate them into an out-of-guest introspection program.

Overview



Runtime Phase

The introspection program can then be deployed to a Security VM to monitor our untrusted VM and applications.

Training

- Write in-guest *training program* using system APIs

```
#define __WIN32_LEAN_AND_MEAN__
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include <stdio.h>
#include "vmnotify.h"

int main(int argc, char **argv) {




  EnumProcesses(pids, 256, &outcb);

  return 0;
}
```


Writing the in-guest program to list processes is easy: just call EnumProcesses.

Training

- Write in-guest *training program* using system APIs

```
#define __WIN32_LEAN_AND_MEAN__
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include <stdio.h>
#include "vmnotify.h"

int main(int argc, char **argv) {
  DWORD *pids = (DWORD *) malloc(256);
  DWORD outcb;


  EnumProcesses(pids, 256, &outcb);

  return 0;
}
```


Of course, you need a little bit of boilerplate.

Training

- Annotate program with start/end markers

```c
#define __WIN32_LEAN_AND_MEAN__
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include <stdio.h>
#include "vmnotify.h"

int main(int argc, char **argv) {
  DWORD *pids = (DWORD *) malloc(256);
  DWORD outcb;

  vm_mark_buf_in(&pids, 4);
  EnumProcesses(pids, 256, &outcb);
  vm_mark_buf_out(pids, 256);
  return 0;
}
```


Next, inform Virtuoso of where logging should begin and end, and where the buffer containing the output of the introspection is.

Training

- Run program in QEMU to generate *instruction trace*
- Traces are in QEMU µOp format

```
INTERRUPT(0xfb,0x200a94,0x0)
TB_HEAD_EIP(0x80108028)
MOVL_T0_IM(0x0)
OPREG_TEMPL_MOVL_A0_R(0x4)
SUBL_A0_4()
OPS_MEM_STL_T0_A0(0x1,0xf186fe8,0x8103cfe8,
                  0xffffffff,0x215d810,0x920f0,0x0)
OPREG_TEMPL_MOVL_R_A0(0x4)
MOVL_T0_IM(0xfb)
OPREG_TEMPL_MOVL_A0_R(0x4)
SUBL_A0_4()
OPS_MEM_STL_T0_A0(0x1,0xf186fe4,0x8103cfe4,
                  0xffffffff,0x215d810,0x920f0,0xfb)
```


This produces instruction traces. They're not x86, but QEMU.

Whole-System Traces

- Includes all instructions between start and end markers
- Includes software and hardware interrupts and exceptions
- Includes concrete addresses of memory reads/writes

Memory reads/writes are necessary so that we can do data flow analysis later.

Trace Analysis

- What subset of this trace is relevant?
- Initial preprocessing:
  - Remove hardware interrupts
  - Replace malloc/realloc/calloc with *summary functions*
- Next, *executable dynamic slicing* (Korel and Laski, 1988) is done to identify *relevant* instructions

System is doing a lot of other stuff that may not be relevant to the introspection at hand. HW interrupts: clearly not relevant. Malloc & friends: implementation artifact.

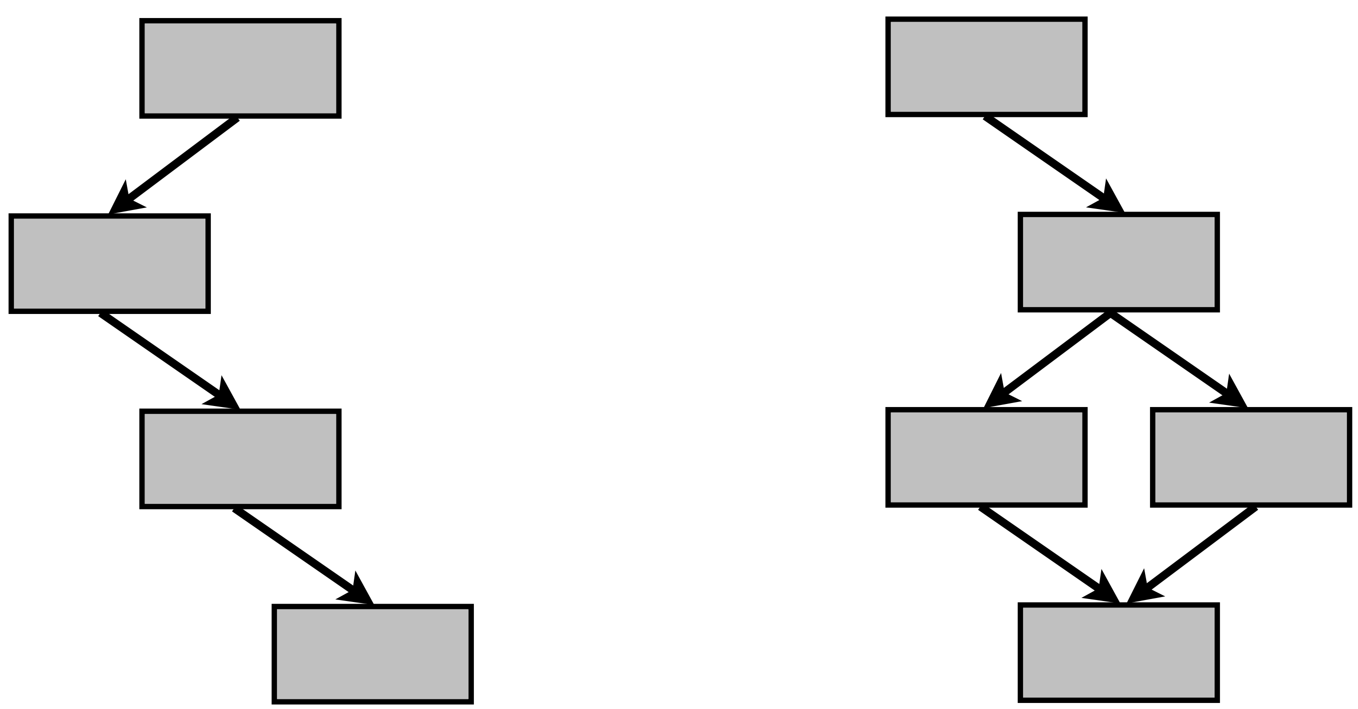
Executable Dynamic Slicing

1. Follow data def/use chain backward, starting with output buffer
2. Examine CFG and add necessary control flow statements to slice (and their dependencies)
3. Perform *slice closure*:
   - If *any* instance of an instruction is included in the slice, *all* instances of that instruction must be marked

Data def/use chains give us our initial set of relevant instructions without control flow. Next step operates on dynamic CFG. Slice closure is what makes it executable.

Trace Merging

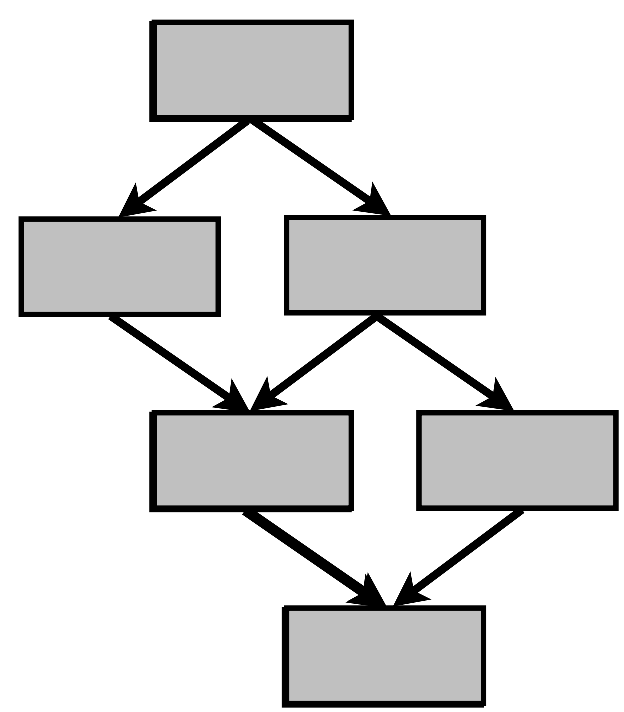
- Since analysis is dynamic, we only see one path through program
- So: run program multiple times and then merge results



Note why we didn't go with static analysis here: too much domain knowledge.

Trace Merging

- Since analysis is dynamic, we only see one path through program
- So: run program multiple times and then merge results



Note why we didn't go with static analysis here: too much domain knowledge.

Program Translation

- Goal: convert in-guest ➔ out-of-guest
- Generates Python code that runs inside Volatility memory analysis framework
- Changes:
  - Memory reads come from guest VM
  - Memory writes are copy-on-write
  - CPU registers become local vars

Why Python? Volatility written in Python. Any code generation would do though. Copy-on-write is necessary so as not perturb the guest.

Translation Example

Original x86

```
test byte [ebp+0x1c],0x10
mov edi,ebx
jnz 0xc02533a9
```

QEMU µOps

```
[TB @0xc0253368L *]
   IFLO_TB_HEAD_EIP(0xc0253368)
   IFLO_INSN_BYTES(0xc0253368,'f6451c10')
 * IFLO_OPREG_TEMPL_MOVL_A0_R(0x5)
 * IFLO_ADDL_A0_IM(0x1c)
 * IFLO_OPS_MEM_LDUB_T0_A0(...)
 * IFLO_MOVL_T1_IM(0x10)
 * IFLO_TESTL_T0_T1_CC()
   IFLO_INSN_BYTES(0xc025336c,'89df')
 * IFLO_OPREG_TEMPL_MOVL_T0_R(0x3)
 * IFLO_OPREG_TEMPL_MOVL_R_T0(0x7)
   IFLO_INSN_BYTES(0xc025336e,'7539')
 * IFLO_SET_CC_OP(0x16)
 * IFLO_OPS_TEMPLATE_JZ_SUB(0x0,0x1)
   IFLO_GOTO_TB1(0x60afcab8)
   IFLO_MOVL_EIP_IM(0xc0253370)
   IFLO_MOVL_T0_IM(0x60afcab9)
   IFLO_EXIT_TB()
```


Example: a test and a conditional jump. Asterisks mean “included in slice”.

Translation Example

QEMU μOps

```
[TB @0xc0253368L *]
   IFLO_TB_HEAD_EIP(0xc0253368)
   IFLO_INSN_BYTES(0xc0253368,'f6451c10')
 * IFLO_OPREG_TEMPL_MOVL_A0_R(0x5)
 * IFLO_ADDL_A0_IM(0x1c)
 * IFLO_OPS_MEM_LDUB_T0_A0(...)
 * IFLO_MOVL_T1_IM(0x10)
 * IFLO_TESTL_T0_T1_CC()
   IFLO_INSN_BYTES(0xc025336c,'89df')
 * IFLO_OPREG_TEMPL_MOVL_T0_R(0x3)
 * IFLO_OPREG_TEMPL_MOVL_R_T0(0x7)
   IFLO_INSN_BYTES(0xc025336e,'7539')
 * IFLO_SET_CC_OP(0x16)
 * IFLO_OPS_TEMPLATE_JZ_SUB(0x0,0x1)
   IFLO_GOTO_TB1(0x60afcab8)
   IFLO_MOVL_EIP_IM(0xc0253370)
   IFLO_MOVL_T0_IM(0x60afcab9)
   IFLO_EXIT_TB()
```

Python

```
A0 = EBP
A0 += UInt(0x1c)
T0 = ULInt8(mem.read(A0,1))
T1 = UInt(0x10)
CC_DST = T0 & T1
T0 = EBX
EDI = T0
CC_OP = 0x16
if (Byte(CC_DST) == 0):
    raise Goto(0xc0253370)
raise Goto(0xc02533a9)
```


Translation to Python: conditional jump is now Python if statement.

Results: Generality

- Generated 6 useful introspection programs on each of 3 operating systems

Windows: everyone uses it. Linux: we use it. Haiku: we don’t know its internals, no temptation to cheat.

Introspection Programs

getpid	Gets the PID of the currently running process.
pslist	Gets a list of PIDs of all running processes.
getpsfile	Gets the name of an executable from its PID.
lsmod	Gets the base addresses of all kernel modules.
getdrvfile	Gets the name of a kernel module from its base address.
gettime	Gets the current system time.

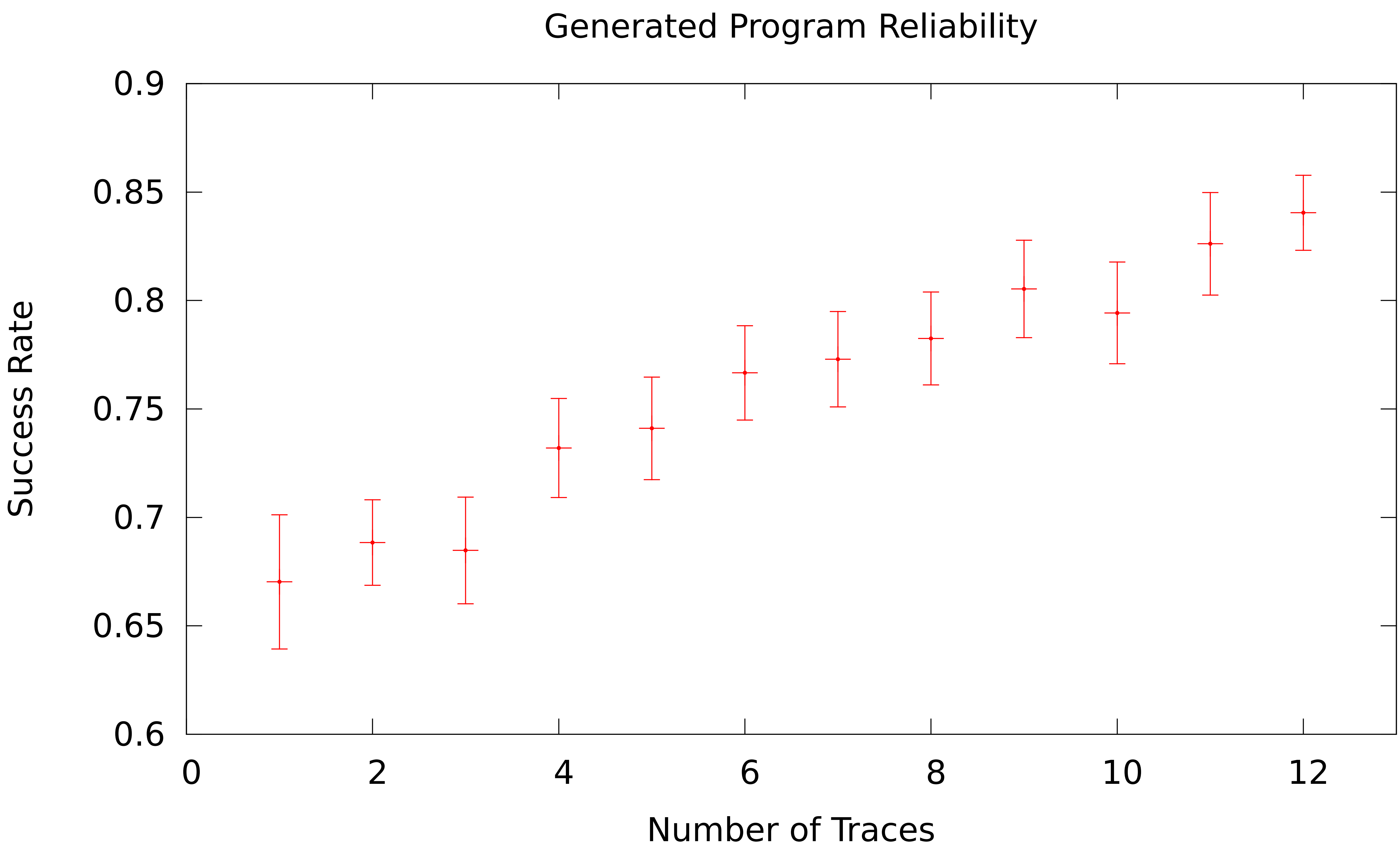
Describe these by group and why relevant to security: examine features of processes and drivers.

Results: Reliability

- Analysis is dynamic, so programs may be incomplete
- How many traces are needed to produce reliable programs?
- Complicating factors: caching, difficulty of deciding ground truth for coverage

Caching: early runs may execute much more code. Difficulty of ground truth: hard to say what the complete set of code is, or how many paths (program testing has this problem too).

Windows pslist Reliability



This is cross-evaluation: take 24 traces, and then take differently sized random subsets to create final program. Describe axes, then walk through one program => not reliable, 12 programs => pretty reliable. Mention caching effect again as explanation for why this graph

Results: Security

- Verified that introspection programs are not affected by in-guest code manipulation
- Training program (pslist) generated on clean system
- Resulting introspection program still detects processes hidden by Hacker Defender
- Note: DKOM attacks can still be effective against Virtuoso

DKOM is something we'll look at in future work.

Limitations

- Multiple processes/IPC
- Multithreaded code (synchronization)
- Code/data relocation (ASLR)
- Self-modifying code

Multiple processes: key problem is that we don't know where data for a specific process might be at runtime. Multithreaded code: VM is paused, so waiting on a lock is bad. Relocation: where's our data? Self-modifying code: code is only translated once (kernel's

Conclusions

- Programs generated by Virtuoso can be useful, reliable, and secure
- Uses novel whole-system executable dynamic slicing and merging
- Virtuoso can greatly reduce time and effort needed to create introspection programs
  - Weeks of reverse engineering vs. minutes of computation

Stop here for questions, etc.