SafeMachine
malware needs love, too

Martin Hron, Jakub Jermář
AVAST Software, research

vb 2014
SEATTLE
24 - 26 September 2014

Dynamic malware detection

Dynamic:

dy·nam·ic *adjective* \dī-'na-mik\: changing; active; in motion

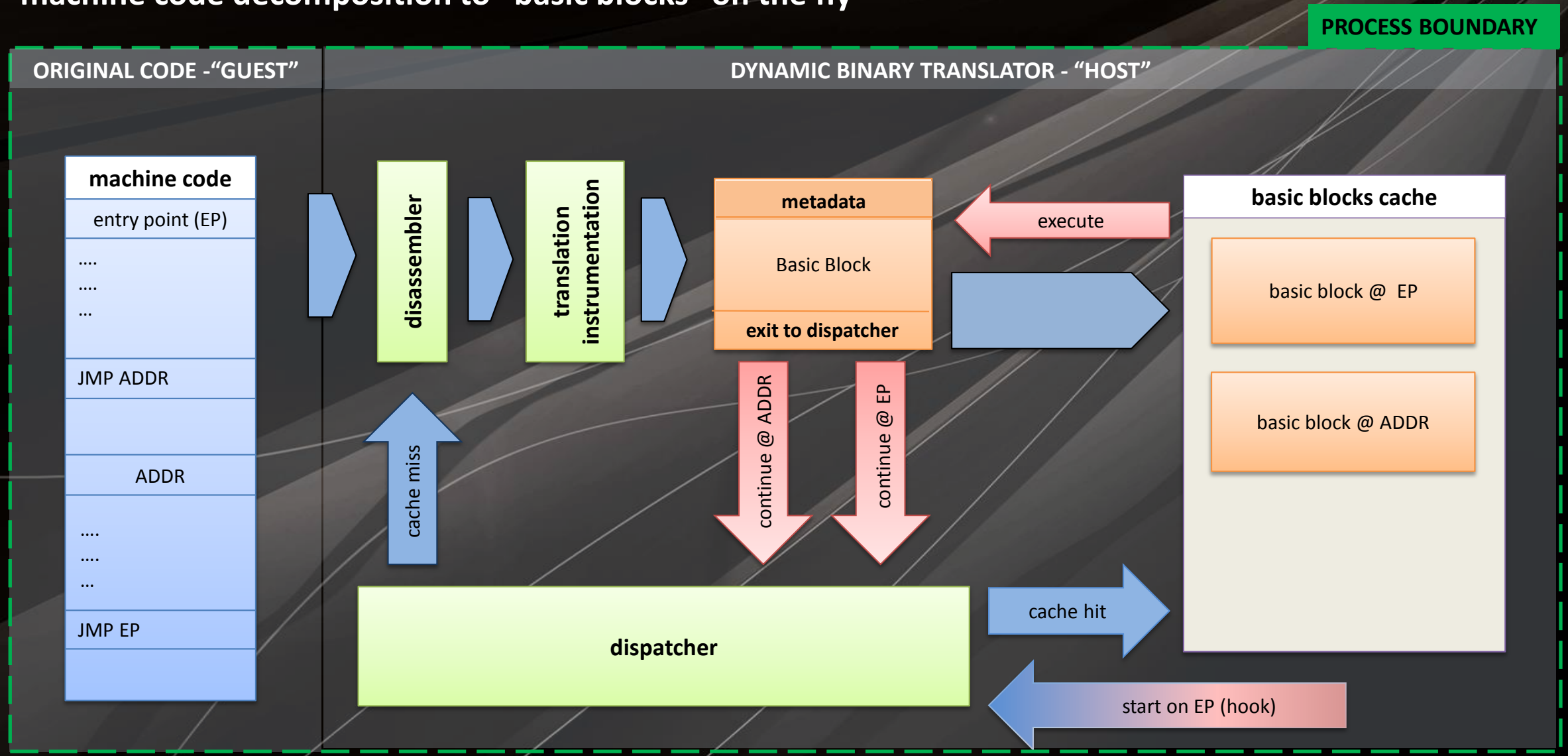
In a context of the digital “pest”: safely run it and watch what’s going inside

Tools for dynamic malware detection widely use binary instrumentation to be able to observe guest behavior on instruction level.

They may use some kind of sandboxing or virtualization to isolate the running guest.

Binary translation (instrumentation)

machine code decomposition to “basic blocks” on the fly



Existing GENERAL purpose frameworks

Pin - A Dynamic Binary Instrumentation Tool
developed and maintained by Intel®. Closed source.

Pin 2.14 User Guide:

"The following Pin switches are supported:
-smc_strict [0][1] Enable (1) or disable (0) support for SMC inside basic blocks. By default (0), pin assumes that basic blocks do not modify their own code."

Existing GENERAL purpose frameworks

DynamoRIO - Dynamic Instrumentation Tool Platform

created at MIT and HP in 2001. Open-sourced in February 2009

2001: Bruening, D., Duesterwald, E., Amarasinghe, S.: Design and Implementation of a Dynamic Optimization Framework for Windows

"We expected to have problems both with exception contexts and with self-modifying code, but neither have occurred in any of the large Windows programs we have been running."

Existing GENERAL purpose frameworks

DynamoRIO - Dynamic Instrumentation Tool Platform

created at MIT and HP in 2001. Open-sourced in February 2009

2005: Bruening, D., Amarasinghe, S.: Maintaining Consistency and Bounding Capacity of Software Code Caches

"While true self-modifying code is only seen in a few applications, such as Adobe Premiere and games like Doom, general code modification is surprisingly prevalent."

Existing special purpose frameworks

?

Existing special purpose frameworks

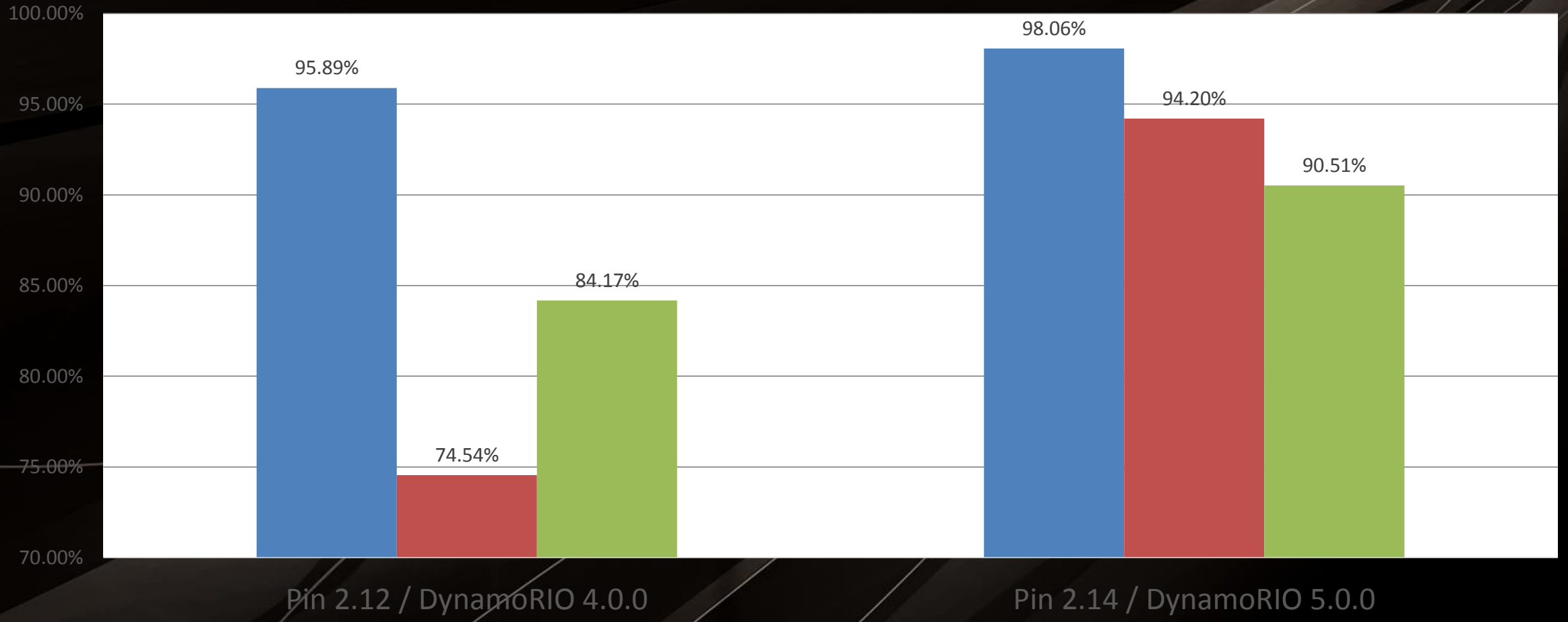
SafeMachine – Dynamic binary malware introspection

Developed by AVAST Software. Currently closed source.

"The general purpose frameworks can handle
*even the most complex instances of self-modifying code**.*
*But there is more, much more***."*

* Eventually and when pushed
** SMC on stack
*** And very little is actually needed to exploit it

Framework comparison



Exploitable areas

Block cache vs. virtual memory state

correct invalidation of block cache on page protection state transitions

Program counter virtualization vs. CPU instructions

some CPU instructions leak the actual program counter

Inherent weakness of the write-protecting approach to SMC

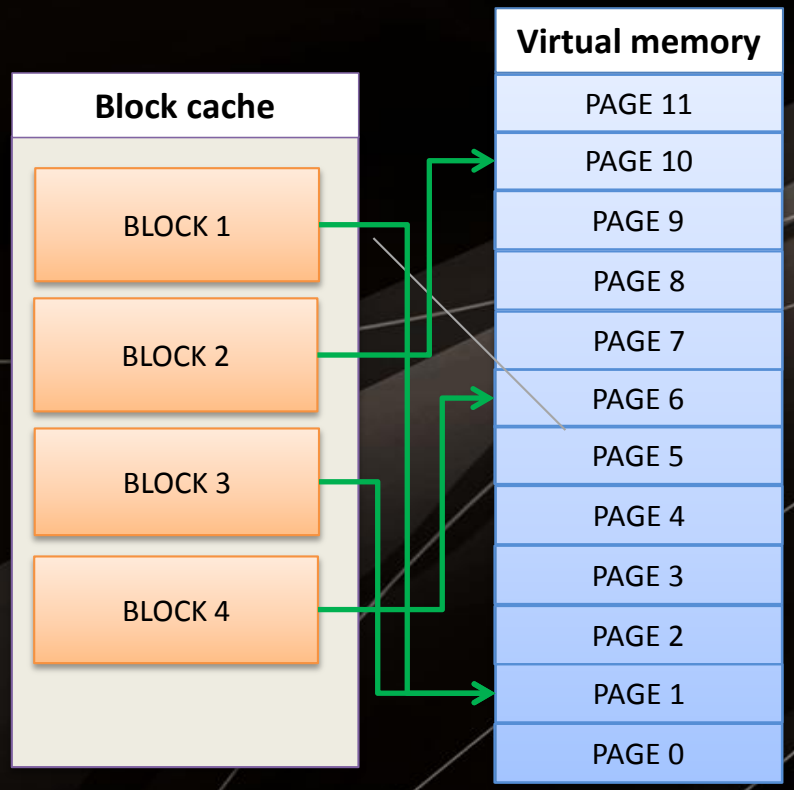
And many more

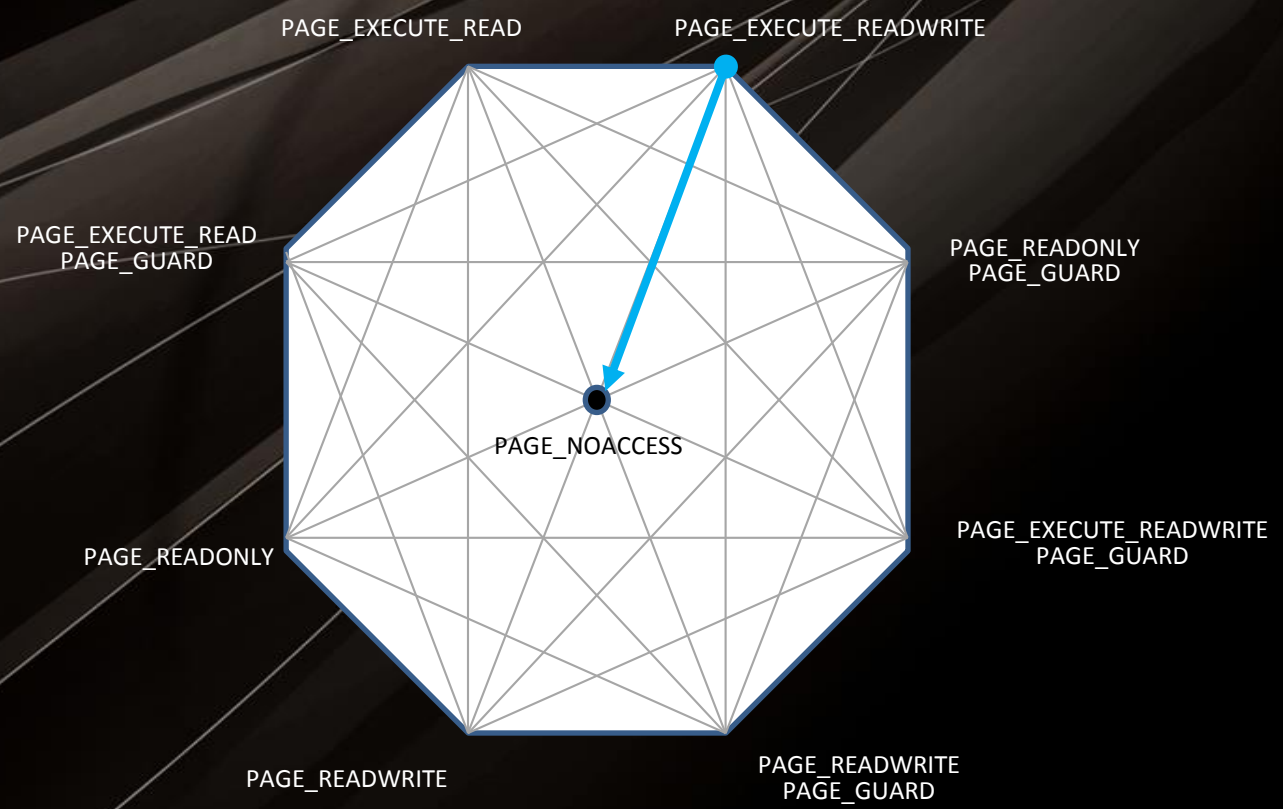
wrong syscall arguments, debug registers, segmentation, single-stepping, 0x66 & 0x67, ...

Block cache problem

Self-modifying code stresses block cache consistency wrt. virtual memory **contents**.

How about code that stresses block cache consistency wrt. virtual memory **state**?





Demo: ExecuteUnmap1.exe

```
PAGE 1:
```

```
start:   nop
         ret
```

```
VirtualProtect(start, 2, PAGE_EXECUTE_READ);

/* First round: create the basic block */
_asm call start

VirtualProtect(start, 2, PAGE_NOACCESS);

/* Second round: the block should no longer be there */
__try {
        _asm call start
        /* FAILED */
}
__except(EXCEPTION_EXECUTE_HANDLER) {
        /* PASSED */
}
```

Idea:

Test if the change of page protection to NOACCESS removes the block from the cache

DEMO

Result:

Both Pin and DR fail the test

Discovery:

Pin behaves differently if the page protection goes from EXECUTE_READWRITE directly to NOACCESS

Demo: TransientException1.exe

PAGE 1:

```
  …
start:
  mov byte ptr [pb - 1], 0x90
  nop
```

PAGE 2:

```
pb:
  nop
  ret
…
```

```
VirtualProtect(pb, 1, PAGE_EXECUTE_READWRITE|PAGE_GUARD);

__try {
        _asm call start
        /* FAILED */
}
__except(EXCEPTION_EXECUTE_HANDLER) {
        if (GetExceptionCode() == EXCEPTION_GUARD_PAGE)
                /* PASSED */
        else
                /* FAILED */
}
```

Idea:

Test if SMC handling preserves the guard page associated with the second page of the block.

DEMO

Result:

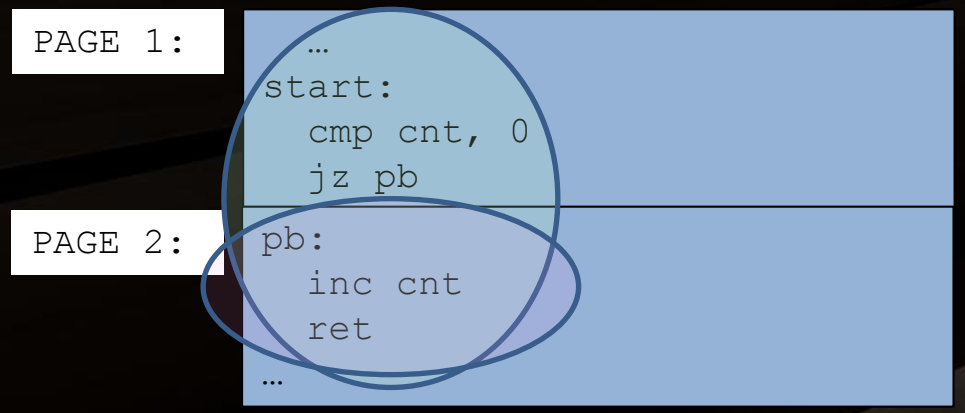
Pin fails the test.

DR crashes.

Discovery:

Both Pin and DR cannot deal with the PAGE_GUARD protection for some combinations of other protections.

Demo: TransientException2.exe




```
VirtualProtect(pb, 1, PAGE_EXECUTE_READWRITE|PAGE_GUARD);
__try {
        _asm call start
        /* FAILED */
} __except(EXCEPTION_EXECUTE_HANDLER) {
        if (GetExceptionCode() != EXCEPTION_GUARD_PAGE)
                /* FAILED */
}
__try {
        _asm call start
        /* PASSED */
} __except(EXCEPTION_EXECUTE_HANDLER) {
        /* FAILED */
}
```

Idea:

Test if the first of two overlapping blocks to hit a guard page consumes it.

DEMO

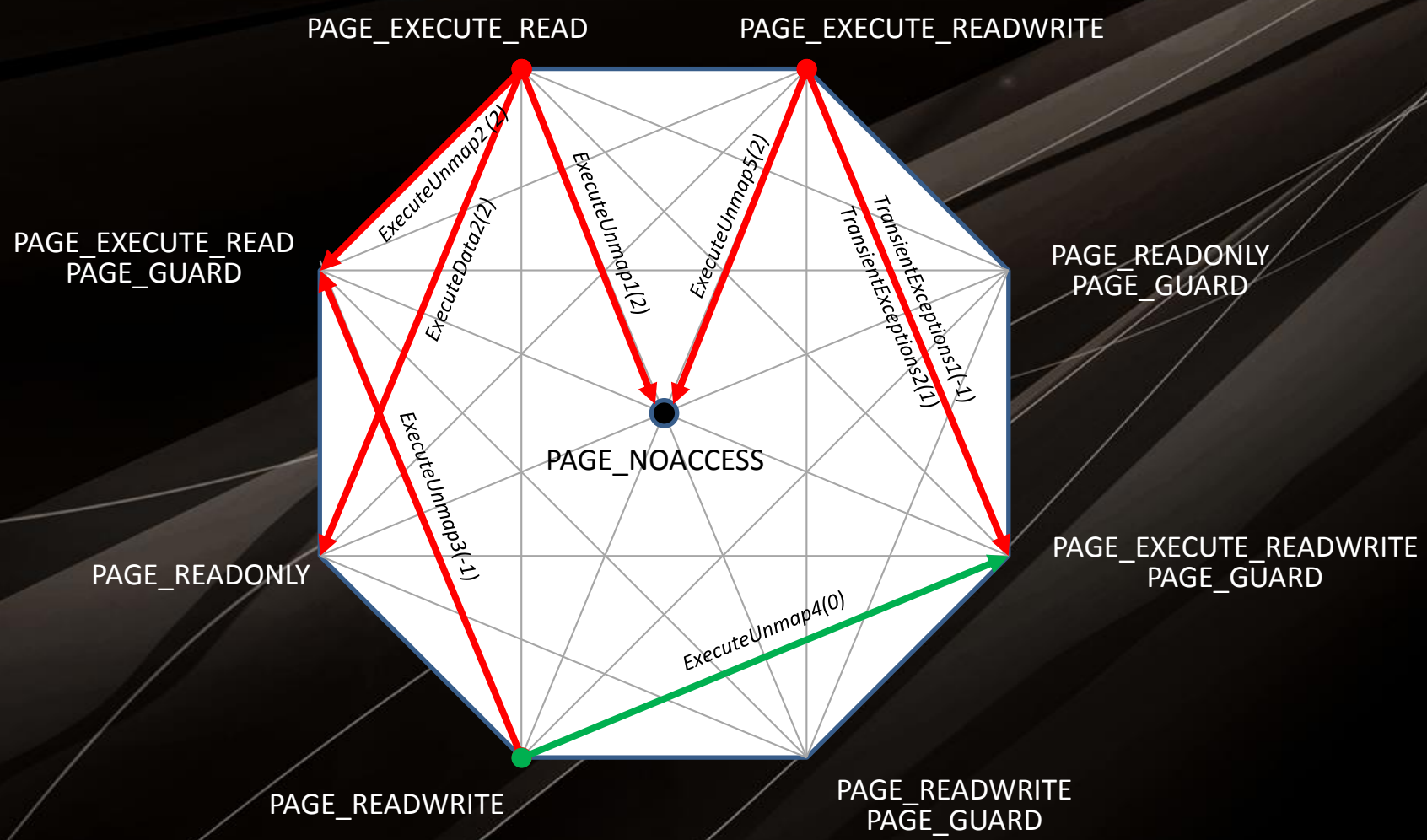
Result:

Both Pin and DR fail the test.

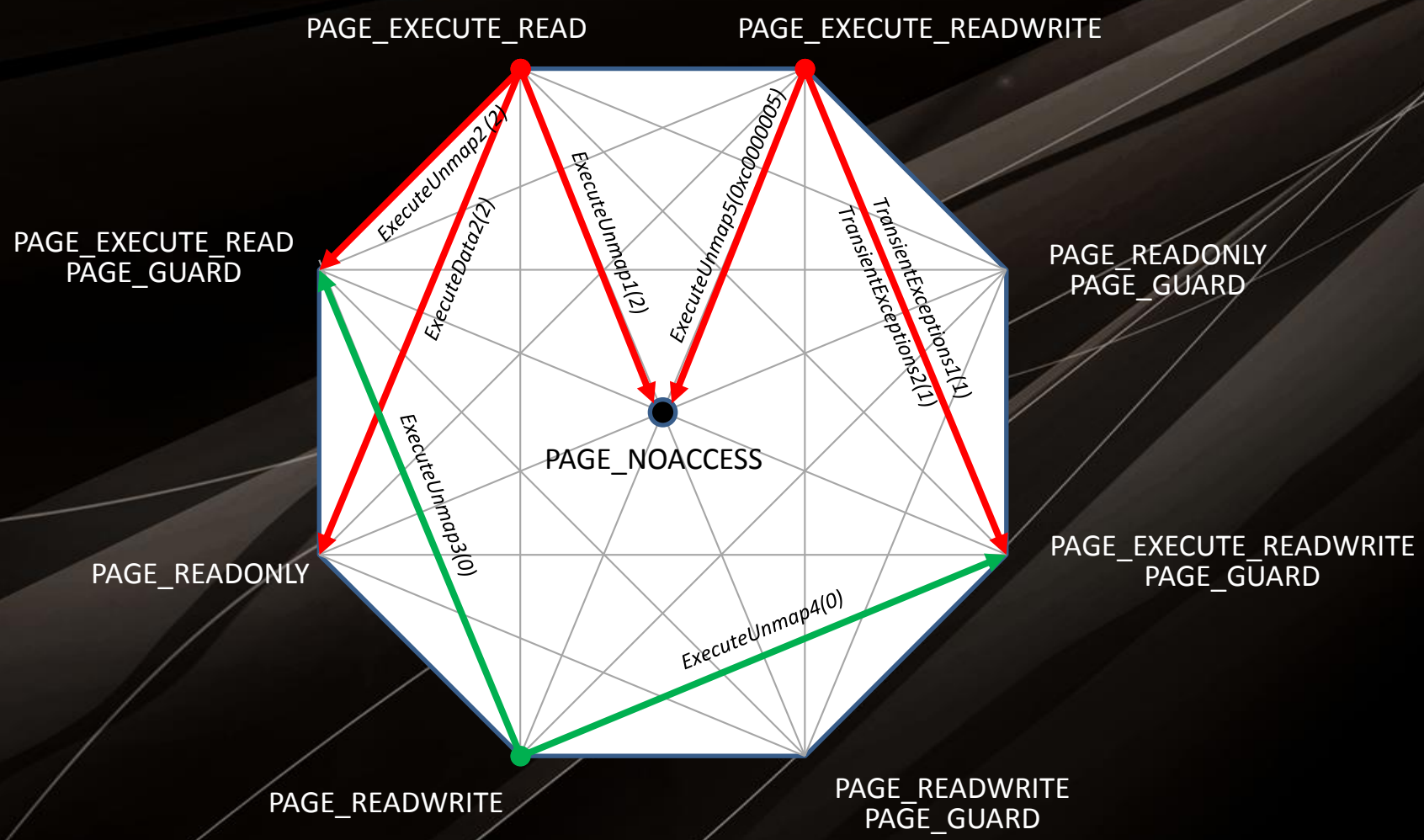
Discovery:

Both Pin and DR cannot deal with the PAGE_GUARD protection for some combinations of other protections.

Virtual memory state transitions (DynamoRIO 5.0.0)



Virtual memory state transitions (Pin 2.14)



Program counter virtualization problem

Binary translated guest code runs from a different address

Guest EIP different from block EIP

Some CPU instructions leak the program counter

FNXSAVE, FNSAVE, FNSTENV

```
010A1003 fsin
```

```
006ED3F1 fsin
006ED3F3 mov dword ptr ds:[0B1F5Ch],10A1003h
```

Demo: FPUContext1.exe

Idea:

Test if the EIP of the last FPU instruction is correctly virtualized.

DEMO

```
start:
          fsin
          fnstenv fpu_save_area
          lea eax, start
          cmp eax, dword ptr fpu_save_area[3 * 4]
          jnz FAILED
          jmp PASSED
```

Result:

Pin fails the test, DR passes.

Demo: FPUContext2.exe

Idea:

Test if the IP of the last FPU instruction is correctly virtualized.

DEMO 

```
start:
          fsin
          _emit OPERAND_SIZE_PREFIX
          fnstenv fpu_save_area
          lea eax, start
          cmp ax, word ptr fpu_save_area[3 * 2]
          jnz FAILED
          jmp PASSED
```

Result:

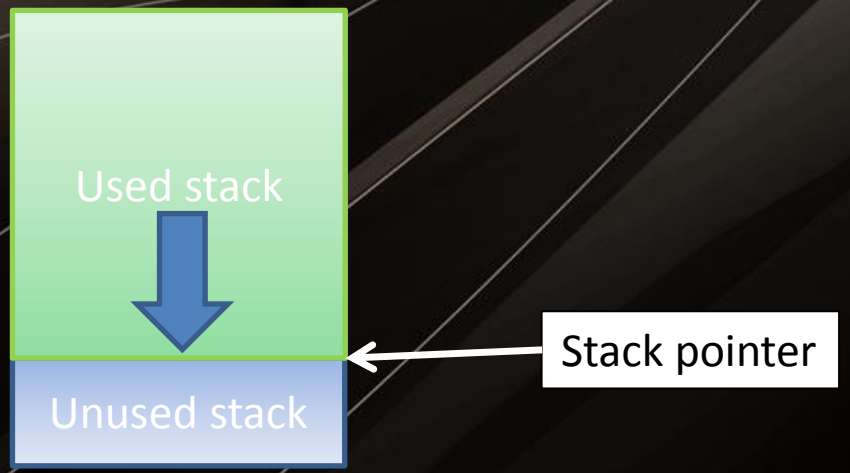
Both Pin and DR fail the test.

Inherent weakness in write-protecting

If SMC is detected by write-protecting…

…then each SMC generates a service exception…

…which smashes a part of the unused (guest) stack



Demo: ServiceException1.exe

```
        push 0xdeadbeef
        push 0xbadcafe1
        pop ecx
        pop eax
        mov byte ptr smc, 0x90
smc:
        _emit 0xcc
        cmp dword ptr [esp - 4], eax
        jnz FAILED
        cmp dword ptr [esp - 8], ecx
        jnz FAILED
        jmp PASSED
```

Idea:

Test if a pattern left on the stack is still there after SMC.

DEMO

Result:

DR and Sf(write-protect) fail the test.

Pin and Sf(memory-check) pass the test.

Observation:

To pass, the tool must either use memory checks or virtualize guest stack.

Conclusion

General purpose DBI frameworks made to work well with normal applications

SMC handling an after-thought (done well)

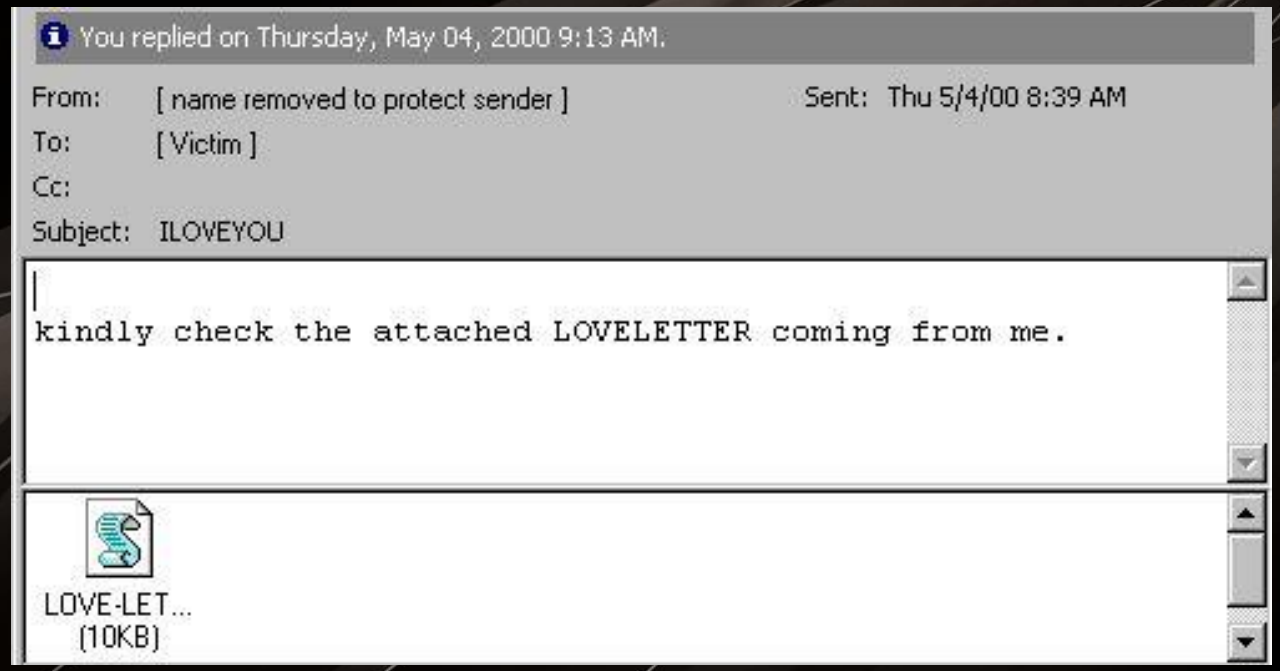
Many other corner cases not handled at all

Dealing with malware requires a DBI framework with a “malware mindset”

Conclusion

Because malware needs love, too

And it definitelly rewards you back ☺



Thank you

It's Q&A time!

Presented and additional samples can be downloaded:
https://github.com/sf2team/vb2014

Jakub Jermář:

E-mail: jermar@avast.com
Twitter: @jjermar

Martin Hron:

E-mail: hron@avast.com
Twitter: @thinkcz