Software-Based Techniques for Protecting Return Addresses

Never Stand Still

Changwei Zou

The Importance of Protecting Return Addresses

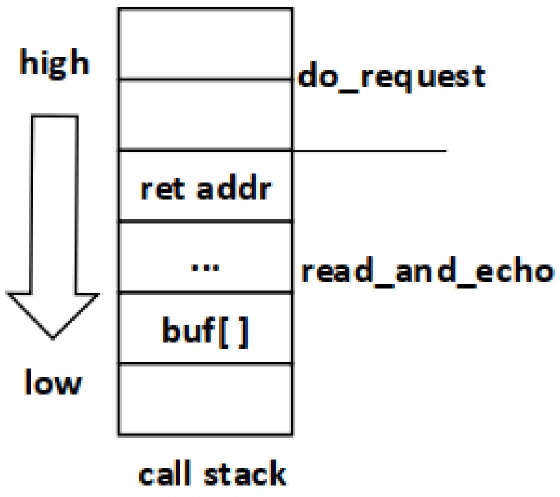
The Stack Conundrum

Proposed by Microsoft

- Most CFG improvements will provide little value-add until stack protection lands, attackers are unanimously corrupting the stack

CFG: Control-Flow Guard

Background: Control-Flow Hijacking



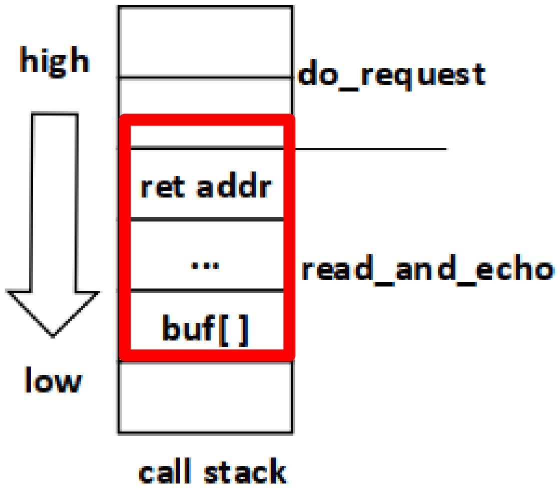

```
void read_and_echo(void){
 char buf[BUFSIZE];
 gets(buf);     // buffer overflow
 printf(buf);
}

void do_request(){
 while(1){
     read_and_echo();
 }
}
```

1. NO memory safety (buffer overflow, …).

   The input from user might be larger than the local buffer.

2. Trade security for performance (C/C++).

Background: Control-Flow Hijacking



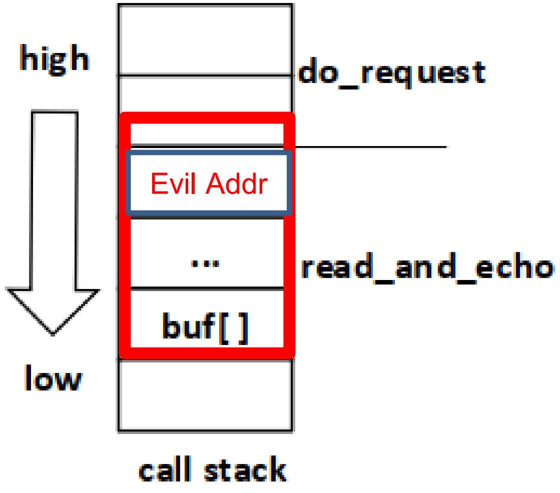

```
void read_and_echo(void){
 char buf[BUFSIZE];
 gets(buf);       // buffer overflow
 printf(buf);
}

void do_request(){
 while(1){
     read_and_echo();
  }
}
```

1. NO memory safety (buffer overflow, …).

   The input from user might be larger than the local buffer.

2. Trade security for performance (C/C++).

Background: Control-Flow Hijacking



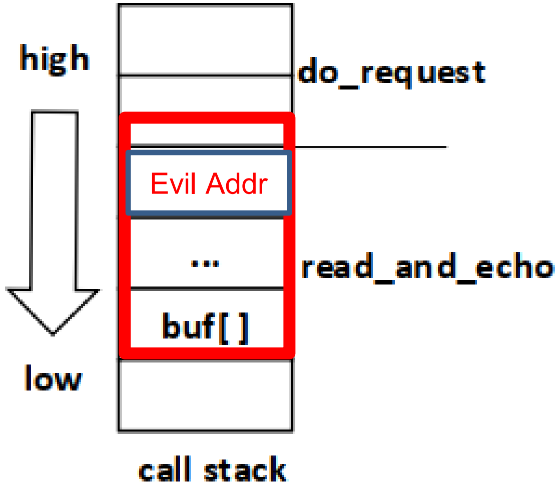

```
void read_and_echo(void){
 char buf[BUFSIZE];
 gets(buf);      // buffer overflow
 printf(buf);
}

void do_request(){
 while(1){
       read_and_echo();
  }
}
```

1. **NO memory safety (buffer overflow, …).**

   The input from user might be larger than the local buffer.

2. **Trade security for performance (C/C++).**

Background: Control-Flow Hijacking




```
void read_and_echo(void){
 char buf[BUFSIZE];
 gets(buf);
 printf(buf); // format string vulnerability
}

void do_request(){
 while(1){
      read_and_echo();
 }
}
```

Arbitrary reads/writes

1. **NO memory safety (buffer overflow, …).**

   The input from user might be larger than the local buffer.

2. **Trade security for performance (C/C++).**

Existing Techniques to mitigate attacks

- Stack Canary
  counteract buffer overflows
  vulnerable to arbitrary writes and information leakage

- (Re-)Randomization
  Make the attacker hard to guess the required addresses

- Control Flow Integrity (CFI)
  Only a set of predefined return targets allowed

- Shadow Stack
  Hide the return addresses in a hidden shadow stack

Challenges in Runtime Re-Randomization

When code sections are remapped at runtime, all code pointers need to be updated.

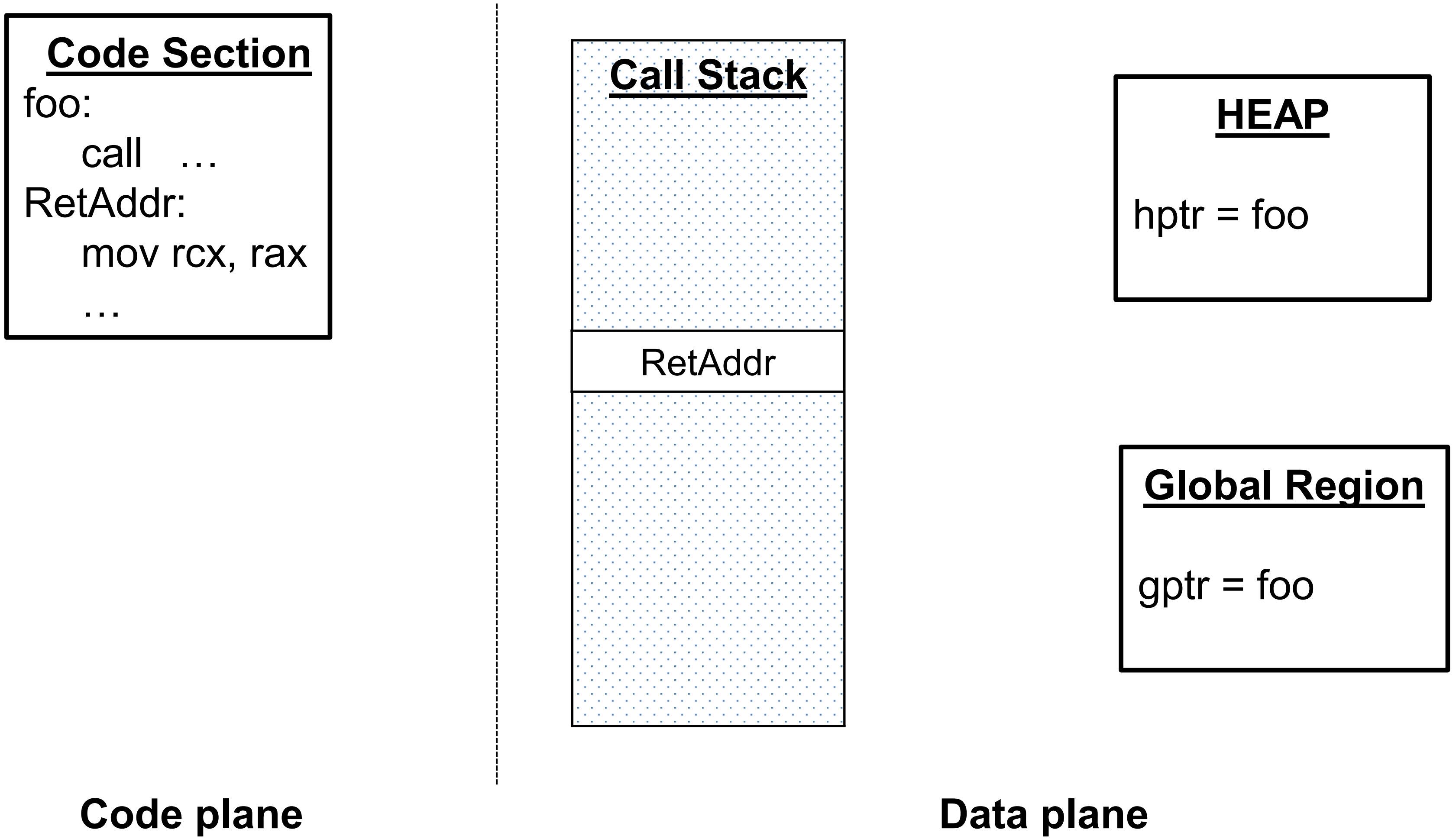
1. **Code-pointer tracking is expensive.**

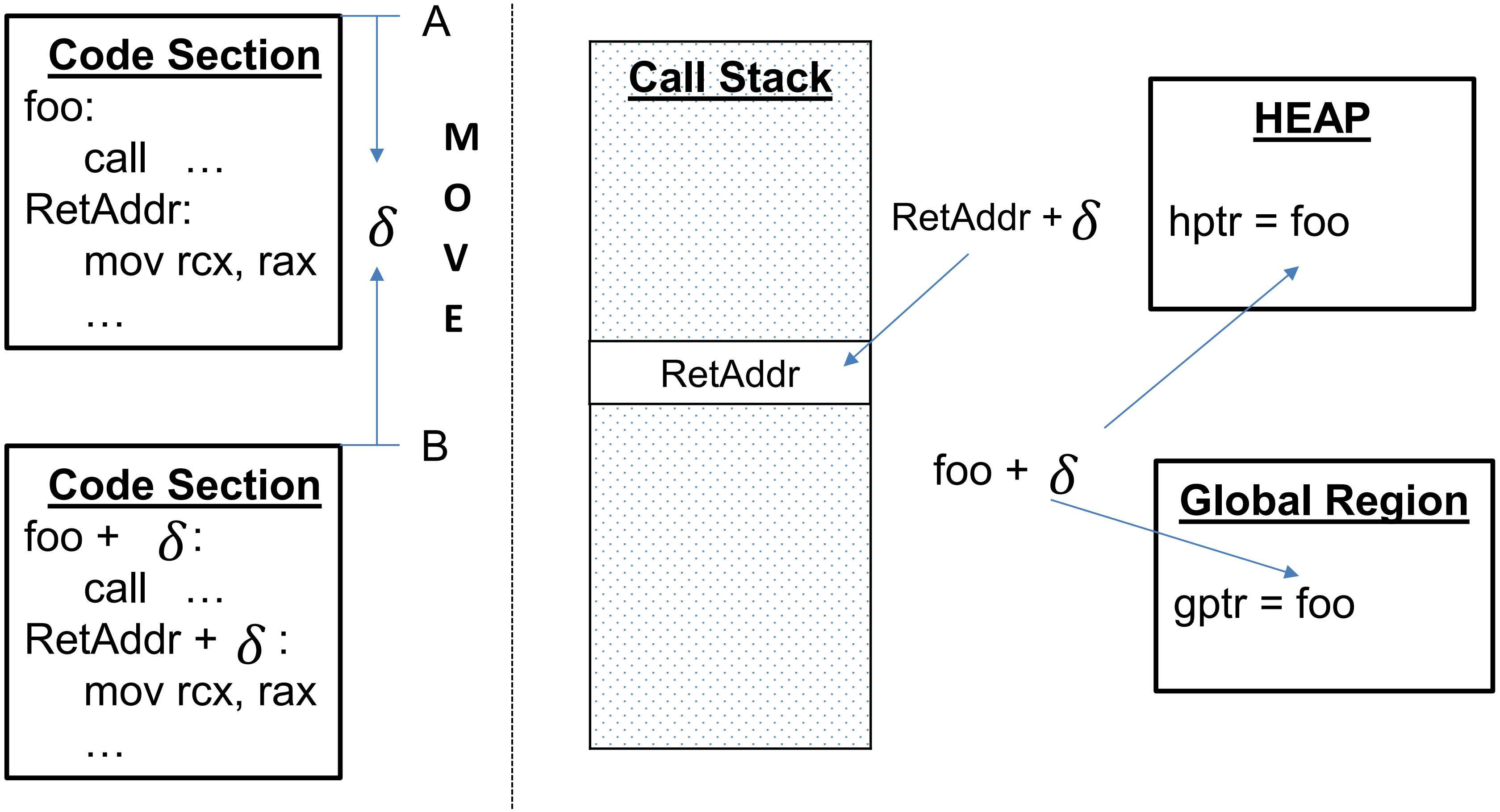
   e.g., **seconds** needed , RUNTIMEASLR (NDSS 16)

2. **Code-pointer tracking is difficult.**

   Both false positives and false negatives may exist.

3. **Should be conducted as frequently as possible**


Code Section
foo:
call …
RetAddr:
mov rcx, rax
…
Call Stack
RetAddr
HEAP
hptr = foo
Global Region
gptr = foo
Code plane
Data plane



Moving in code plane

Pointer tracking and updating in data plane

Limitation of Control Flow Integrity in Protecting Return Addresses

Still leaving enough leeway for attackers

e.g., the return of printf() in FreeBSD has over 5,000 allowed targets

Thus, fully-precise shadow stacks are often recommended to be used for protecting return addresses

The challenges of Implementing Software-Based Shadow Stacks

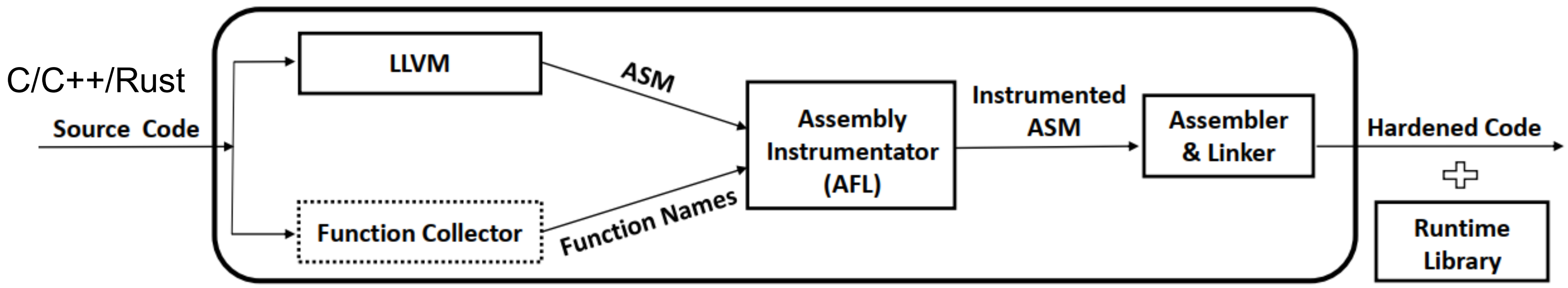
Need to keep a balance among:

(1) COMPATIBILITY: support multi-threading; protected + unprotected code may co-exist

(2) PERFORMANCE: should be efficient

(3) SECURITY: How to mitigate time-of-check to time-of-use attacks on x86? …

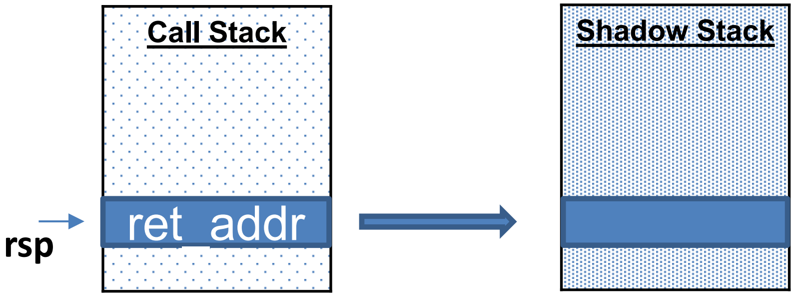
FLASHSTACK : fast-moving parallel shadow stacks on x86-64
(A software-hardening tool based on the LLVM compiler)



The workflow of FLASHSTACK

How to Mitigate Time-Of-Check To Time-Of-Use (TOCTTOU attacks) on x86?

Time-Of-Check To Time-Of-Use (TOCTTOU attacks) on x86




```
            call foo
ret_addr:
            ....
```

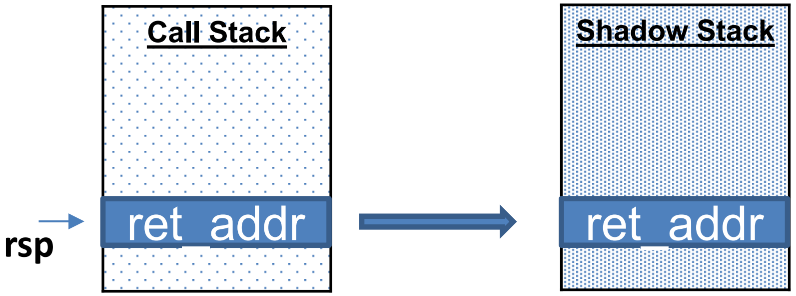
foo:

```
01 # Prologue
02 mov     (%rsp), %r11
03 mov     %r11, -0x800000(%rsp)

....

04 # Epilogue
05 mov     -0x800000(%rsp),%r11
06 add     $0x8, %rsp
07 jmpq    *%r11
```

A protected function foo()

(1) ret_addr is pushed on the call stack by the call instruction
(2) Copied to the shadow stack at the prologue of foo(), lines 2-3
(3) Restore the return address from the shadow stack, lines 5-7

Time-Of-Check To Time-Of-Use (TOCTTOU attacks) on x86




```
          call foo
ret_addr:
           ....
```

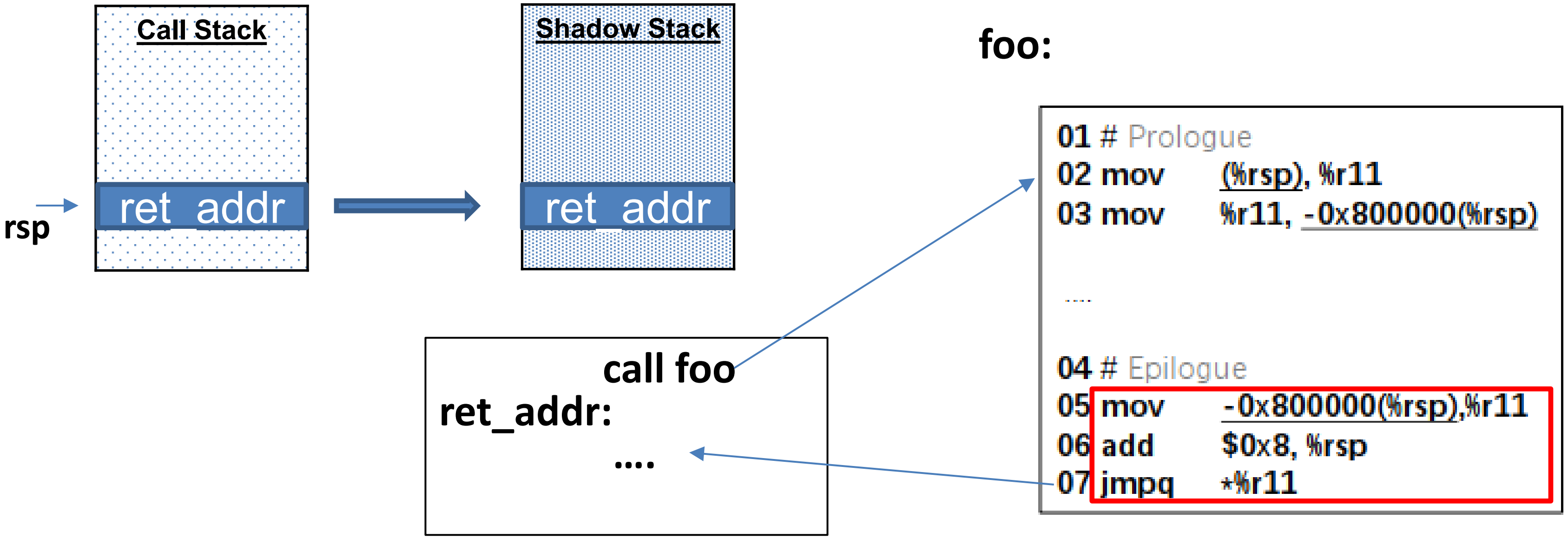
foo:

```
01 # Prologue
02 mov     (%rsp), %r11
03 mov     %r11, -0x800000(%rsp)

....

04 # Epilogue
05 mov     -0x800000(%rsp),%r11
06 add     $0x8, %rsp
07 jmpq    *%r11
```

A protected function foo()

(1) ret_addr is pushed on the call stack by the call instruction
(2) Copied to the shadow stack at the prologue of foo(), lines 2-3
(3) Restore the return address from the shadow stack, lines 5-7

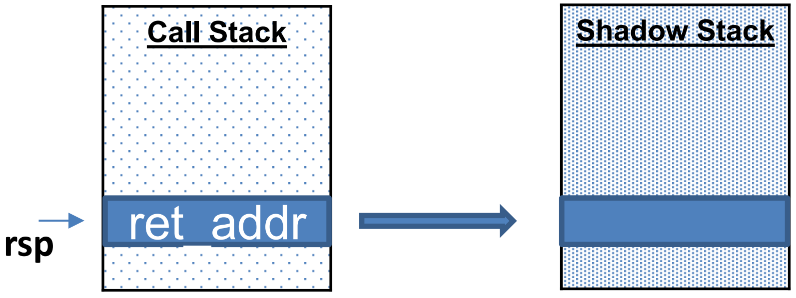
Time-Of-Check To Time-Of-Use (TOCTTOU attacks) on x86



A protected function foo()

(1) ret_addr is pushed on the call stack by the call instruction
(2) Copied to the shadow stack at the prologue of foo(), lines 2-3
(3) Restore the return address from the shadow stack, lines 5-7

But an attack might happen between (1) and (2) in client-side multi-threaded programs



foo:

```
01 # Prologue
02 mov      (%rsp), %r11
03 mov      %r11, -0x800000(%rsp)

....

04 # Epilogue
05 mov      -0x800000(%rsp),%r11
06 add      $0x8, %rsp
07 jmpq     *%r11
```

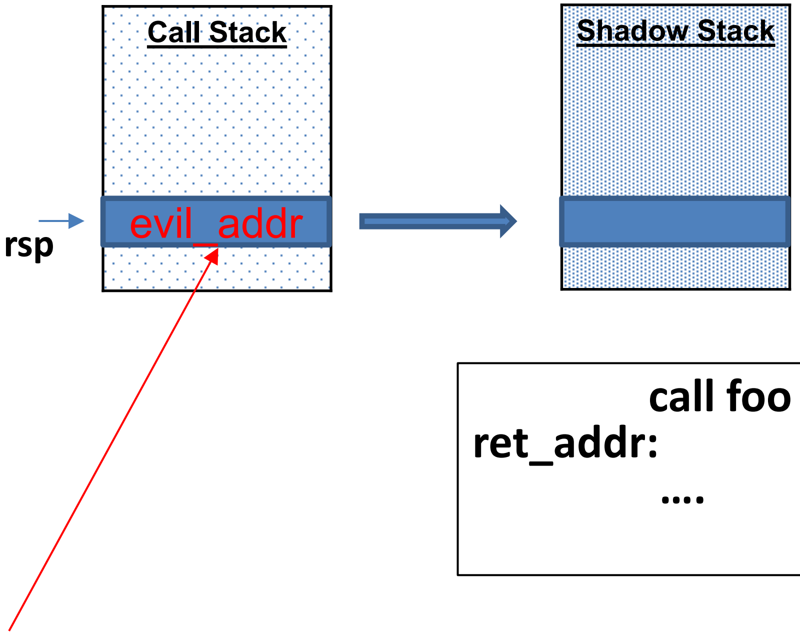
A protected function foo()

```
            call foo
ret_addr:
            ....
```

(1) ret_addr is pushed on the call stack by the call instruction

(2) Copied to the shadow stack at the prologue of foo(), lines 2-3

But an attack might happen between (1) and (2) in client-side multi-threaded programs




```
            call foo
ret_addr:
                ....
```

foo:

```
01 # Prologue
02 mov      (%rsp), %r11
03 mov      %r11, -0x800000(%rsp)

....

04 # Epilogue
05 mov      -0x800000(%rsp),%r11
06 add      $0x8, %rsp
07 jmpq     *%r11
```

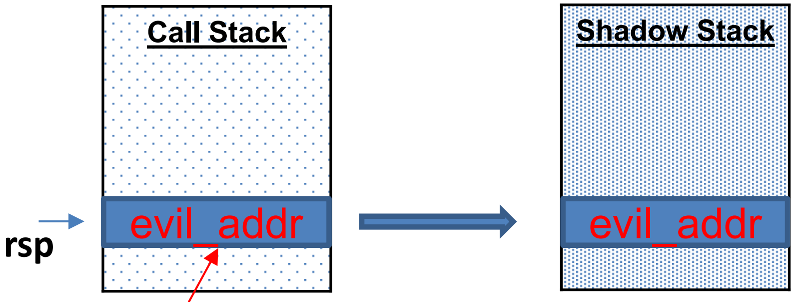
A protected function foo()

Attack from another thread

(1) ret_addr is pushed on the call stack by the call instruction

(2) Copied to the shadow stack at the prologue of foo(), lines 2-3

But an attack might happen between (1) and (2) in client-side multi-threaded programs




```
        call foo
ret_addr:
        ....
```

foo:

```
01 # Prologue
02 mov     (%rsp), %r11
03 mov     %r11, -0x800000(%rsp)

....

04 # Epilogue
05 mov     -0x800000(%rsp),%r11
06 add     $0x8, %rsp
07 jmpq    *%r11
```

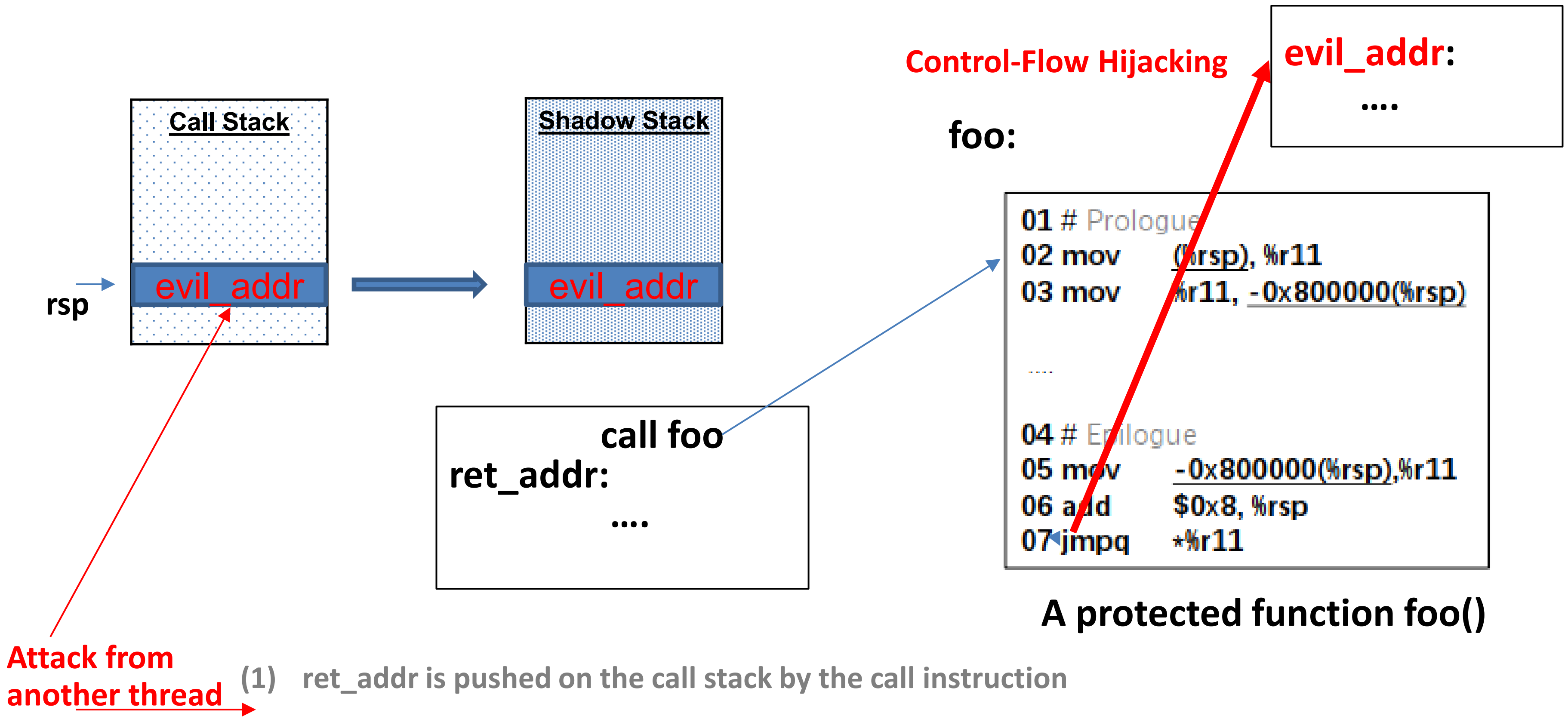
A protected function foo()

Attack from
another thread

(1) ret_addr is pushed on the call stack by the call instruction

(2) Copied to the shadow stack at the prologue of foo(), lines 2-3

But an attack might happen between (1) and (2) in client-side multi-threaded programs



A protected function foo()

(1) ret_addr is pushed on the call stack by the call instruction

(2) Copied to the shadow stack at the prologue of foo(), lines 2-3

(3) Restore the return address from the shadow stack, lines 5-7

Exploiting the TOCTTOU window was thought to be hard.

"Any such attack would rely on accurately timing the victim process and manipulating the OS scheduler to pause the victim's execution precisely between the call and mov instruction. "

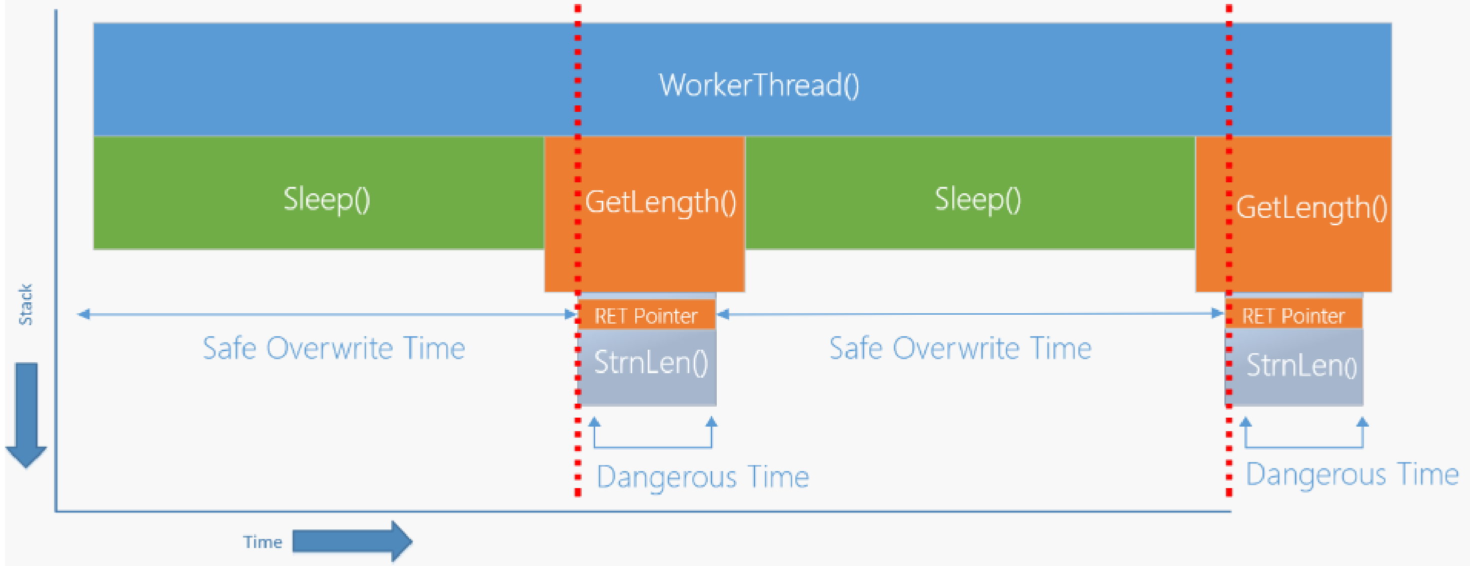
Exploiting the TOCTTOU window was thought to be hard.

"Any such attack would rely on accurately timing the victim process and manipulating the OS scheduler to pause the victim's execution precisely between the call and mov instruction. "

But Microsoft's red team gave a TOCTTOU attack in client-side multi-threaded programs (e.g., browsers)

Microsoft's Time-of-Check To Time-of-Use (TOCTTOU) Attack

RFG Learnings: Winning the Race



Thread 1: In the above sleep() -> GetLength() -> sleep() loop
Thread 2: Constantly writing to the virtual address of "RET Pointer" of the strnlen function

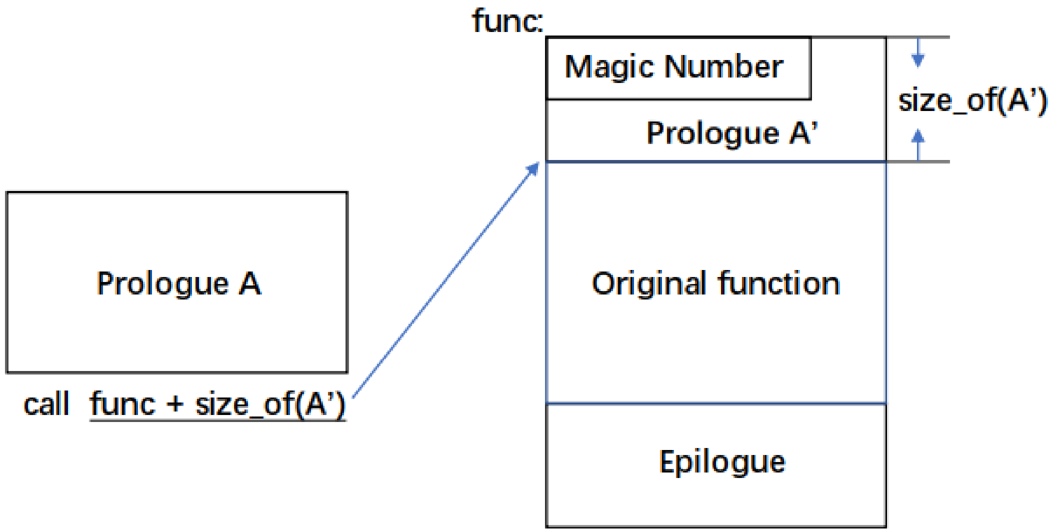
By attacking a leaf function, 99.99% of the "writes" are harmless. When the leaf function is entered, you have a very high probability of winning the race.

```
void MaliciousThread(void *arg){
  while(1){
    *ret_slot = evil_addr;
  }
}

void GetLength(){
  ...
  // StrnLen is a leaf function
  StrnLen();
  ...
}

void WorkerThread(void *arg){
  while(!stop){
    Sleep();
    GetLength();
  }
}
```

Proposed by Microsoft

Leading to the deprecation of Microsoft's shadow stack (i.e., RFG)

Dual Prologues: our approach for mitigating TOCTTOU attacks



Dual Prologues

Prologue A: move the ret address to shadow stack before a call instruction and skip the Prologue A', thus closing the TOCTTOU window

Prologue A': only left for compatibility

Our approach for mitigating TOCTTOU attacks

```
# direct call
#   callq     foo

# modified as follows
# save the return address to r11
    leaq      1f(%rip), %r11
    movq      $-0x800000000000, %r10
# move r11 to shadow stack
    movq      %r11, %gs:-8(%rsp, %r10, 1)
# skip the prologue
    callq     foo+19
 1:
    ...
```

NO stack access here !

Secure + Performant

19 bytes

```
foo:
# Prologue
400850: 4c 8b 1c 24   movq (%rsp),%r11
400854: 49 ba 00 00   movq $-0x800000000000,%r10
        00 00 00 80
        ff ff
40085e: 65 4e 89 1c   movq %r11,%gs:(%rsp,%r10,1)
        14
   ...
# Epilogue
4008a1: 49 ba 00 00   movq $-0x800000000000,%r10
        00 00 00 80
        ff ff
4008ab: 48 83 c4 08   addq $0x8,%rsp
4008af: 65 42 ff 64   jmpq *%gs:-0x8(%rsp,%r10,1)
        14 f8
```

Dual Prologues

Is it possible to reserve a general-purpose register (e.g., r15/xmm15) in 64-bit Linux?

(For support multi-threading)

Leading to program crashes.

Program	LLVM 7.0	
	VANILLA	MODIFIED
Firefox	12	0
libsoftokn3.so	72	0
libssl3.so	18	0
libmozavutil.so	3	0
libfreeblpriv3.so	365	$\boxed{61}$
libxul.so	14,781	$\boxed{2{,}491}$
libmozavcodec.so	4,948	$\boxed{4{,}218}$
Other 14 shared libraries	0	0

The number of occurrences of register xmm15 after reserving it in LLVM

It does not work well on x86-64.

(1) Assembly code (584 assembly files in Firefox79)

(2) Closed-source libraries

(3) Unprotected code (incremental deployment)

Albeit sacrificing some compatibility,
only the segment register gs appears to be reserved practically in 64-bit Linux.

How to implement shadow stacks efficiently (PERFORMANCE) ?

(1) Reduce memory accesses

(Memory is much slower than CPU)

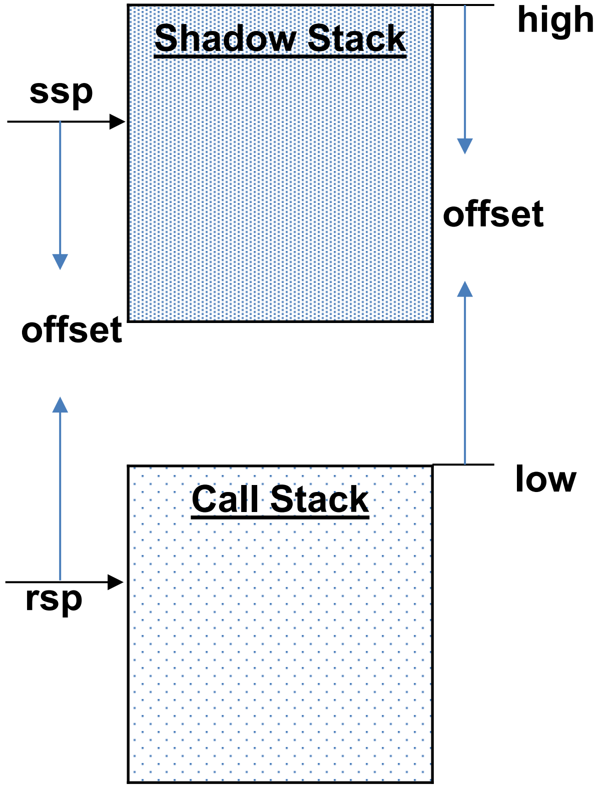
(2) Microsoft's Return Flow Guard (RFG) is efficient.

(i.e., the SEGMENT+RSP scheme)

(3) But SEGMENT+RSP does not work well in 64-bit Linux

A new mapping mechanism, SEGMENT+RSP-S, **in 64-bit Linux**

Why Microsoft's SEGMENT+RSP scheme does not work well in 64-bit Linux?




```
// gs_base = offset
// 0 <= offset <= 0x7FFFFFFFEFFF
arch_prctl(ARCH_SET_GS, offset);

 # prologue
 mov (%rsp), %r11
 # ssp = gs_base + rsp
 mov %r11, %gs:(%rsp)
 ...
 # epilogue
 # ssp = gs_base + rsp
 mov %gs:(%rsp), %r11
 cmp (%rsp), %r11
 jne fastfail
 ret
fastfail:
 ud2
```

Microsoft's RFG rewritten in 64-bit Linux

Constraints:

(1) The system call arch_prctl()

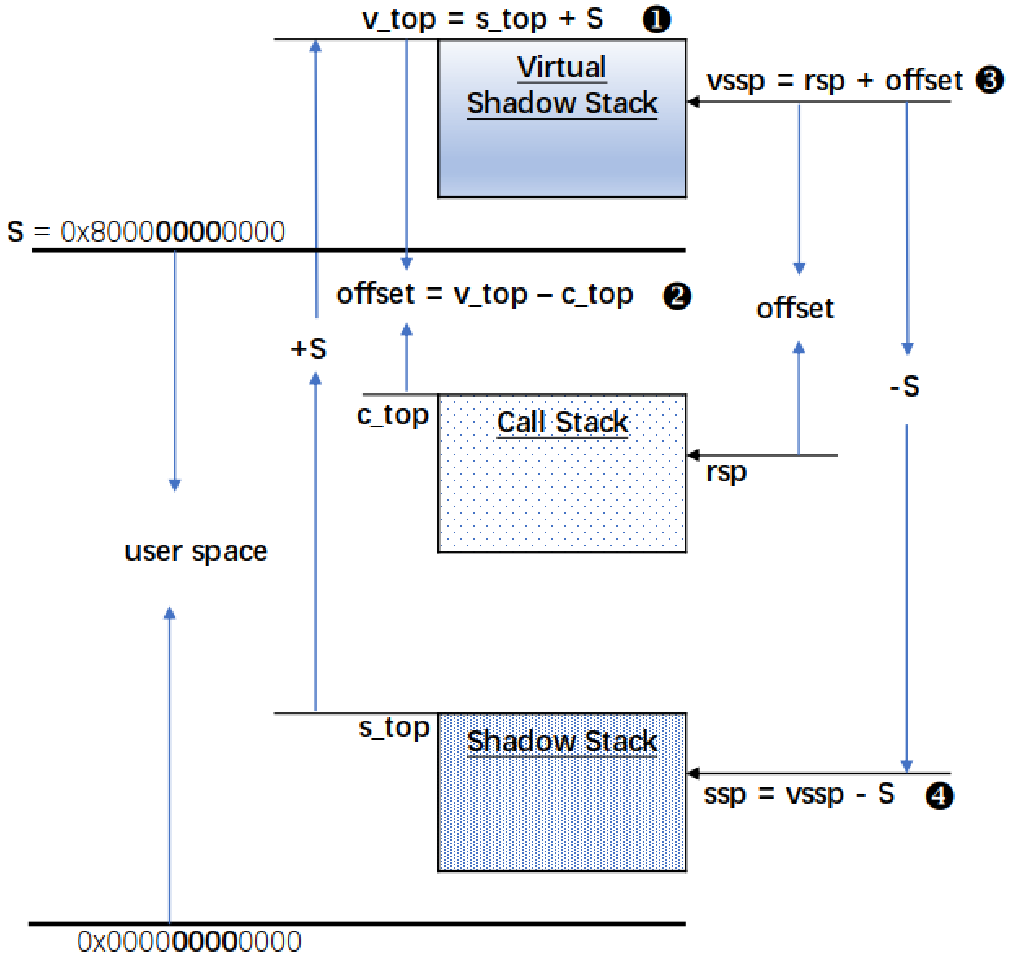
A shadow stack should be placed higher than its call stack.

(2) Address Space Layout Randomization

But a call stack (with 40-bit entropy at the top) might randomly appear at the highest position in user space

(no or small space left for the shadow stack)

Our solution: SEGMENT+RSP-S in 64-bit Linux



Constraint:

// 0 <= offset <= 0x7FFFFFFFEFFF
arch_prctl(ARCH_SET_GS, offset);

Solution:

Add a virtual shadow stack above the user space,

$$offset = s\_top + S - c\_top$$

$$ssp = rsp + offset - S$$

Coding:

```
movq $-0x800000000000, %r10
movq %r11, %gs:(%rsp, %r10, 1)
```

How to protect our shadow stacks (SECURITY) ?

(1) Fast-moving (shuffling) shadow stacks continuously

(Make attackers hard to expose movable shadow stacks)

(2) Use the system call mremap() for remapping

(No memory copy needed, only changing page tables in the OS kernel.)

About **13 microseconds per shuffling**

Performance and compatibility

(1) Server Programs

Concurrency	Vanilla Nginx Requests/Second	SHADESMAR Overhead	FLASHSTACK Overhead
c=1	14731.38	1.11%	1.65%
c=2	25526.54	0.04%	-0.30%
c=4	26077.47	-0.01%	0.31%
c=8	26868.31	0.12%	-0.01%

(2) Browsers (e.g., Firefox 79.0, about 20 million lines of code)

Due to register clashing, SHADESMAR (the state-of-the-art r15-based shadow stack, SP'19) cannot run Firefox.
By contrast, our FLASHSTACK can.

Browser Benchmark	#Subtests	Average Overhead
Octane 2.0	17	3.44%
JetStream2	64	7.04%

Do our shadow stacks consume an excessively high amount of physical memory in practice ?

Group	Application	#Threads/#Processes	Call Stack Size (Bytes)			Call Stack Depth		
			MIN	MAX	AVG	MIN	MAX	AVG
Tools	clang-7	1,665	19,344	163,424	60,326	28	358	88
	clang	553	19,840	19,888	19,864	23	23	23
	clang++	1,131	19,840	19,888	19,864	23	23	23
	/usr/bin/find	19	5,696	5,696	5,696	15	15	15
	/usr/bin/ld	19	13,728	13,728	13,728	21	24	22
	/usr/bin/xargs	19	3,336	3,336	3,336	15	15	15
	/bin/hostname	1	3,336	3,336	3,336	13	13	13
	/bin/rm	57	3,336	3,336	3,336	12	12	12
	/bin/sh	432	3,336	5,984	3,452	14	14	14
	runspec	1	270,640	270,640	270,640	222	222	222
	specmake	57	17,968	53,040	21,627	82	82	82
IDEs	Eclipse	3	1,872	44,304	16,592	9	233	88
	jdk1.8/bin/java	86	1,216	958,864	31,292	9	8,702	177
	Python3.6	1	80,560	80,560	80,560	183	183	183
Browsers	Firefox79.0	121	560	133,344	13,546	7	277	37
	Chrome84.0	41	1,056	112,640	15,033	9	276	40
Servers	Nginx1.18	5	9,488	9,488	9,488	24	43	39
	Apache Httpd2.4.46	28	4,416	35,376	32,251	12	30	28

Based on a PIN tool developed

Only tens of KBs per shadow stack on average

Questions