Shadowclone: Thwarting and Detecting DOP Attacks with Stack Layout Randomization and Canary

EECS 583

Yunjie Pan, Shibo Chen, Cheng Chi, Yifan Guan

Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo

Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo

DOP Example

Modeled after FTP server

Uses a stack buffer overflow vulnerability to control a few stack variables

Affected Variables

```
struct server{ int * cur_max, total, typ;} *srv;
int connect_limit = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12];
...
while(connect_limit--){
    readData(sockfd, buf);          // stack bof
    if(*type == NONE) break;
    if(*type == STREAM) {           // condition
        *size = *(srv->cur_max);// dereference
    } else {
        srv->typ = *type;           // assignment
        srv->total += *size;        // addition
    }
    ...
}
```

Buffer Overflow

Gadget Dispatcher

Gadgets

Simple Example

Simple Stack Overflow

If successful, var1 and var 2 will be changed to 583

```c
#include <stdio.h>
#include <stdlib.h>

void func(){
	double var1 = 483.0;
	int var2 = 483;
	char var3 = 'e';
	char buff[4];
	char* buff_ptr = &buff[0];
	size_t size = 1024;
	getline(&buff_ptr, &size, stdin);
	printf("var1: %f\n", var1);
	printf("var2: %i\n", var2);
	printf("var3: %c\n", var3);
	printf("buff: %s\n", buff);
	printf("This is function func.\n");
}

int main(){
	printf("Calling func:\n");
	func();
	printf("func returned.\n");
}
```

Affected Variables

Buffer Overflow

Prior work - Smokestack

Randomizes the order of stack variables during runtime with P-BOX

+ Much harder to deliver DOP attacks

+ Negligible memory overhead

- Runtime performance overhead

- Cannot detect attacks when happening

Goals

- Reduce runtime overhead by compile time randomization

- Detect attacks when happening

Threat Model

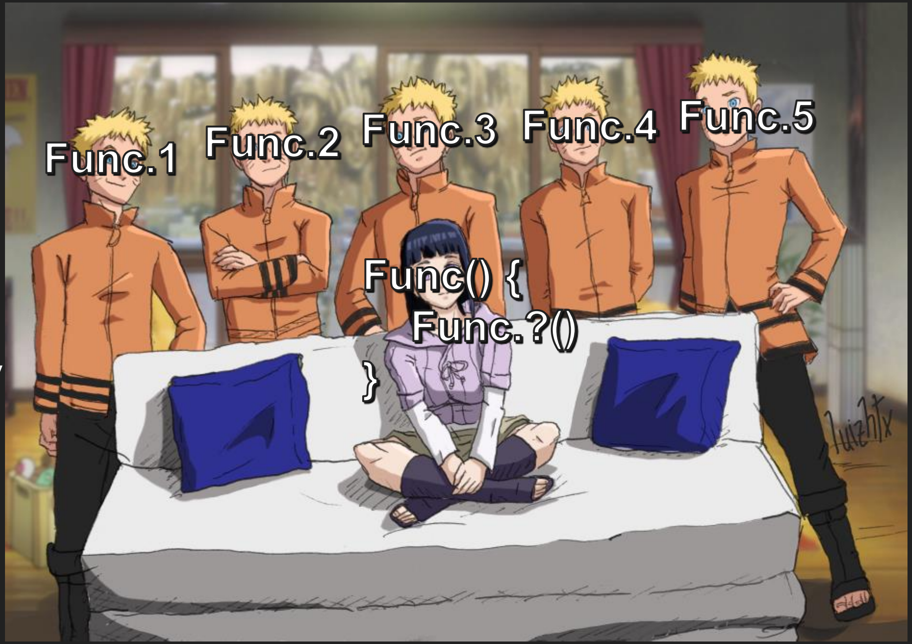
- CFI (Control Flow Integrity) defenses deployed
- Stack buffer overflow vulnerability
- Attackers cannot see the code, but can learn gradually

Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo

Shadowclone

- Generate compile-time randomized clones of vulnerable functions
- Insert compile-time random canary into stack variables and check before the function returns
- Randomly select copy to execute in run time



```c
#include <stdio.h>
#include <stdlib.h>

void func(){
    double var1 = 483.0;
    int var2 = 483;
    char var3 = 'e';
    char buff[4];
    char* buff_ptr = &buff[0];
    size_t size = 1024;
    getline(&buff_ptr, &size, stdin);
    printf("var1: %f\n", var1);
    printf("var2: %i\n", var2);
    printf("var3: %c\n", var3);
    printf("buff: %s\n", buff);
    printf("This is function func.\n");
}

int main(){
    printf("Calling func:\n");
    func();
    printf("func returned.\n");
}
```

Affected Variables

Buffer Overflow

Randomized Stack order

```c
void func1(){
    int var2 = 483;
    uint32_t canary = 1092384;
    double var1 = 483.0;
    char buff[4];
    char var3 = 'e';
    char* buff_ptr = &buff[0];
    size_t size = 1024;
    getline(&buff_ptr, &size, stdin);
    ...
    if (canary != 1092384){
        exit(1);
    }
}

...

void func_wrapper(){
    int fp_index = rand() % 3;
    if (fp_index == 0){
        func0();
    } else if (fp_index == 1){
        func1();
    } else {
        func2();
    }
}
```

Canary Var

Hard-coded canary check (cmp embedded constant)

Randomly select clone to execute

Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo

Implementation

1. Find all concrete functions (except main and *syscall*)
2. Find all alloca instructions
3. Clone a function min(*threshold*, num(alloca)!) times
4. Randomize the order of stack variables
5. Insert the canary and checks
6. Convert original function to randomly select a clone in run-time

Randomize the order of stack variables

Generate a random ordering (a configuration)

If this configuration already exists:

Continue

Else:

Apply this config to one of the clones

Repeat until all clones have been randomized

Insert canary and checks

1. Randomly select an insertion point and insert a 32-bit canary
2. Generate a random number and store it to the location of our canary
3. Insert a *compare-and-branch* duo before each *return* instruction

   (branch to the exception handler if compromise detected)

```
define void @func() #0 {
  %1 = alloca double, align 8
  %2 = alloca i32, align 4
  %3 = alloca i8, align 1
  %4 = alloca [4 x i8], align 1
  %5 = alloca i8*, align 8
  %6 = alloca i64, align 8
  store double 4.830000e+02, double*
  store i32 483, i32* %2, align 4
  store i8 101, i8* %3, align 1
  %7 = getelementptr inbounds [4 x i8
  store i8* %7, i8** %5, align 8
  store i64 1024, i64* %6, align 8
  %8 = load %struct.__sFILE*, %struct
  %9 = call i64 @getline(i8** %5, i64
  ret void
}
```

```
define dso_local void @func.1() #0 {
entry:
  %buff = alloca [4 x i8], align 1
  %var1 = alloca double, align 8
  %canary = alloca i32
  store i32 780689205, i32* %canary
  %var2 = alloca i32, align 4
  %buff_ptr = alloca i8*, align 8
  %var3 = alloca i8, align 1
  %size = alloca i64, align 8
  store double 4.830000e+02, double* %var1, align 8
  store i32 483, i32* %var2, align 4
  store i8 101, i8* %var3, align 1
  %arrayidx = getelementptr inbounds [4 x i8], [4 x i8]* %buff
  store i8* %arrayidx, i8** %buff_ptr, align 8
  store i64 1024, i64* %size, align 8
  %0 = load %struct._IO_FILE*, %struct._IO_FILE** @stdin, alig
  %call = call i64 @getline(i8** %buff_ptr, i64* %size, %struc
  %1 = load i32, i32* %canary
  %2 = icmp eq i32 %1, 780689205
  br i1 %2, label %3, label %func_exit

3:                                                  ; preds = %e
  ret void

func_exit:                                          ; preds = %e
  call void @detect_breach()
  br label %3
}
```

Canary Var

Hard Coded CMP

Run-time Selection

get_rand() is defined in our run-time library

Generates a i32 random number with RDRAND instruction

```
define dso_local void @func() #0 {
rand_bb:
  %0 = call i32 @get_rand()
  %1 = icmp eq i32 %0, 0
  br i1 %1, label %func_func.1, label %ctrl0

func_func.1:                                         ; preds = %rand_bb
  call void @func.1()
  ret void

func_func.2:                                         ; preds = %ctrl0
  call void @func.2()
  ret void

func_func.3:                                         ; preds = %ctrl1
  call void @func.3()
  ret void

func_func.4:                                         ; preds = %ctrl1
  call void @func.4()
  ret void

ctrl0:                                               ; preds = %rand_bb
  %2 = icmp eq i32 %0, 1
  br i1 %2, label %func_func.2, label %ctrl1

ctrl1:                                               ; preds = %ctrl0
  %3 = icmp eq i32 %0, 2
  br i1 %3, label %func_func.3, label %func_func.4
}
```

Branch based on random number

Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo

Experiment Setup

Platform:

Xeon Gold 6126, Ubuntu 18.04 Linux, 256GB of memory

Benchmarks:

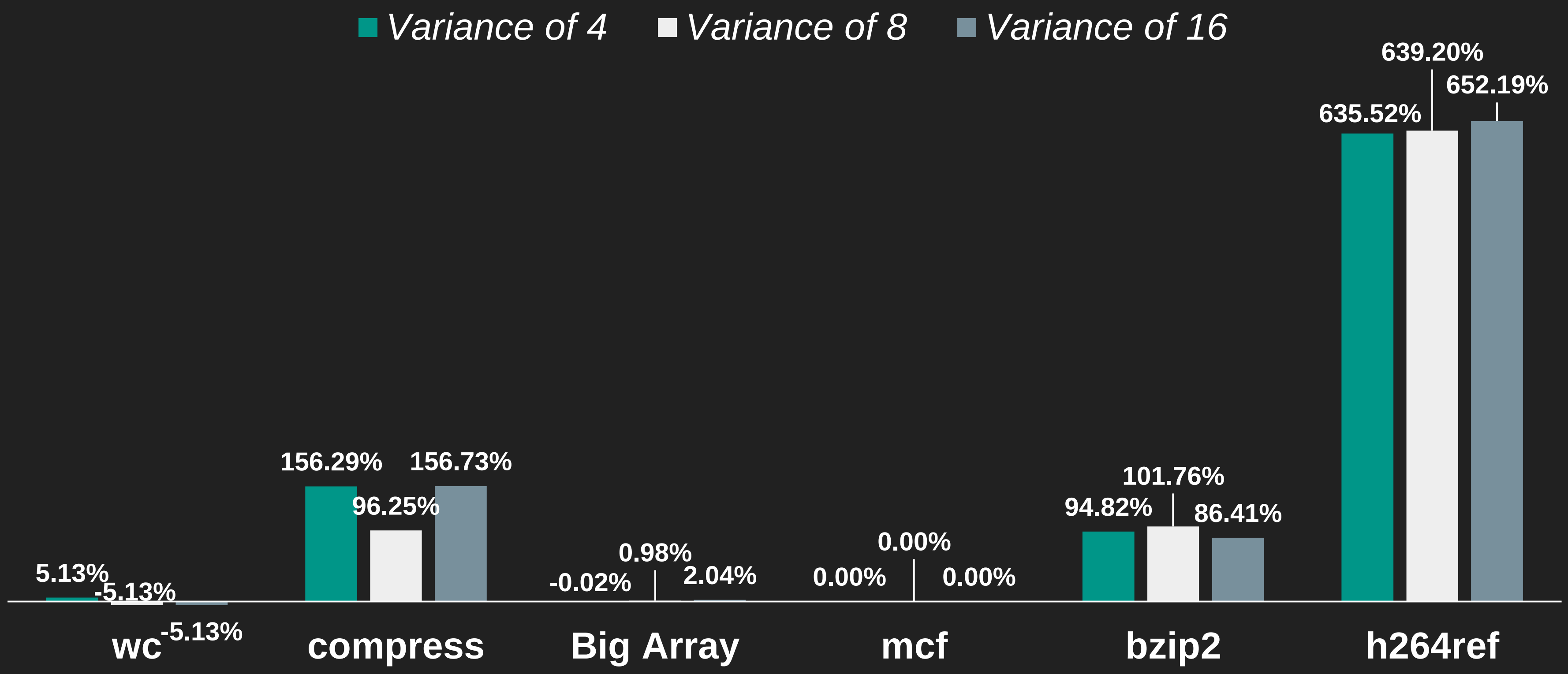
Three in-House testcases (*big_array, wc, and compress*)

Three Spec06 benchmarks (*bzip2, mcf, and h264ref*)

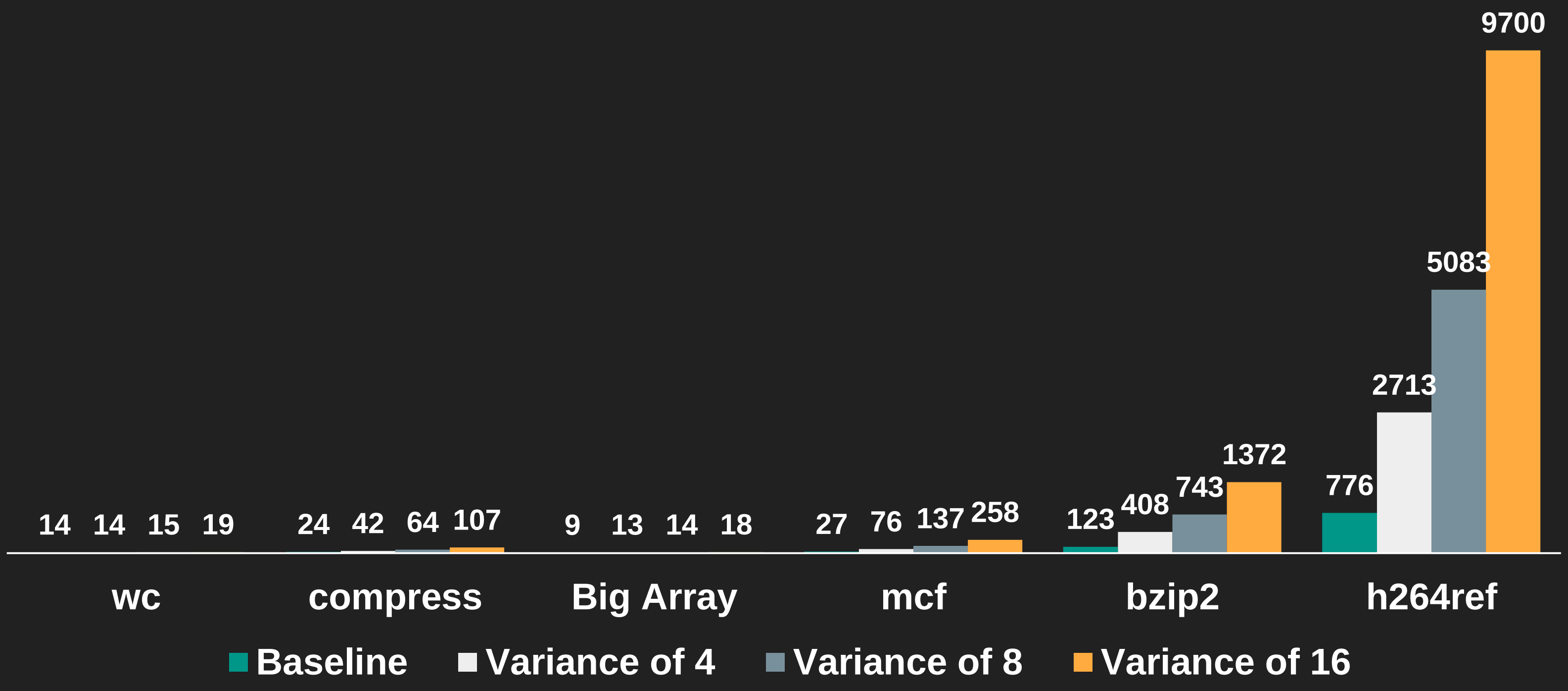
Source of random numbers:

RDRAND

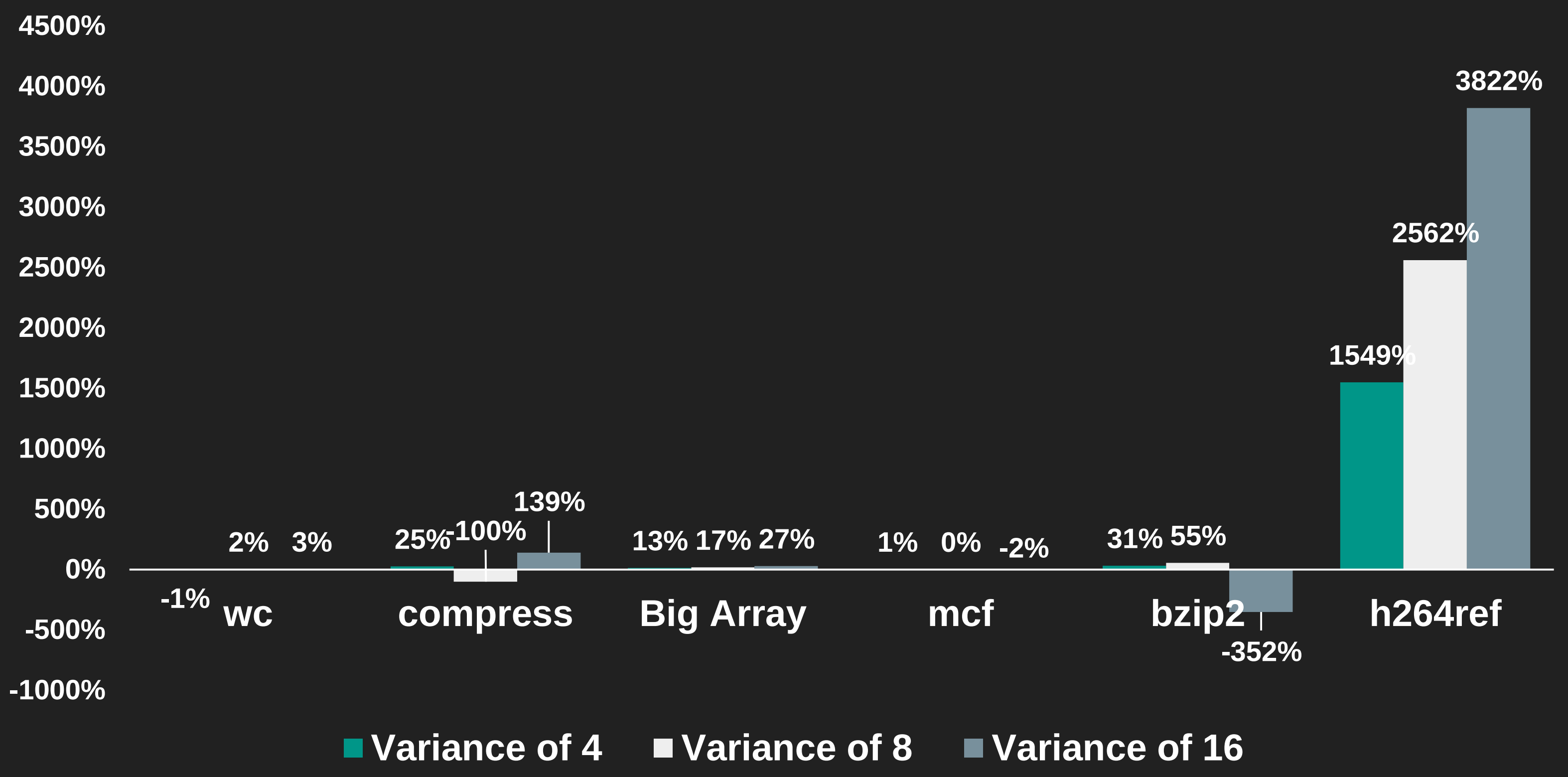
Performance Overhead



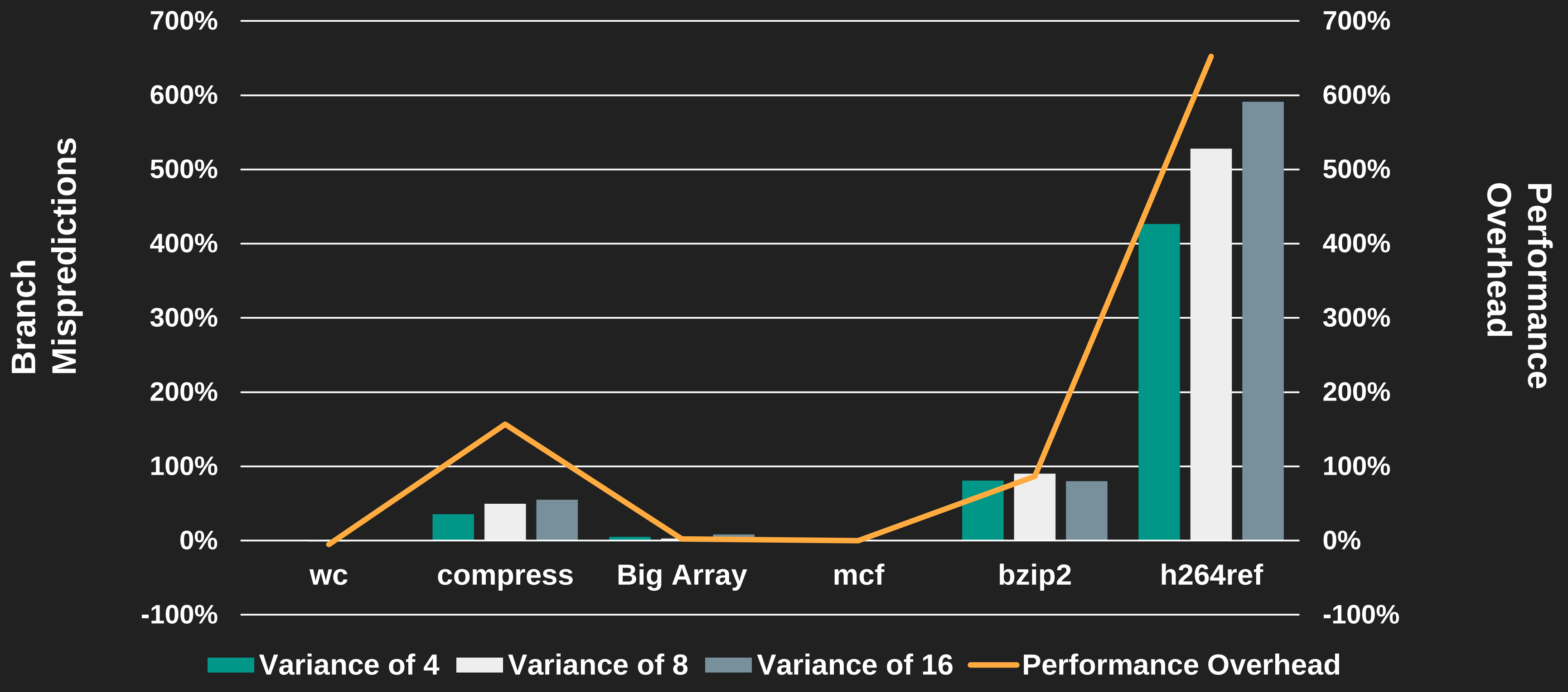
Spatial Locality: *Code Size (in KB)*



Spatial Locality: # of I-Cache Misses



Temporal Locality & Speculation: *# of Branch Mispredictions*



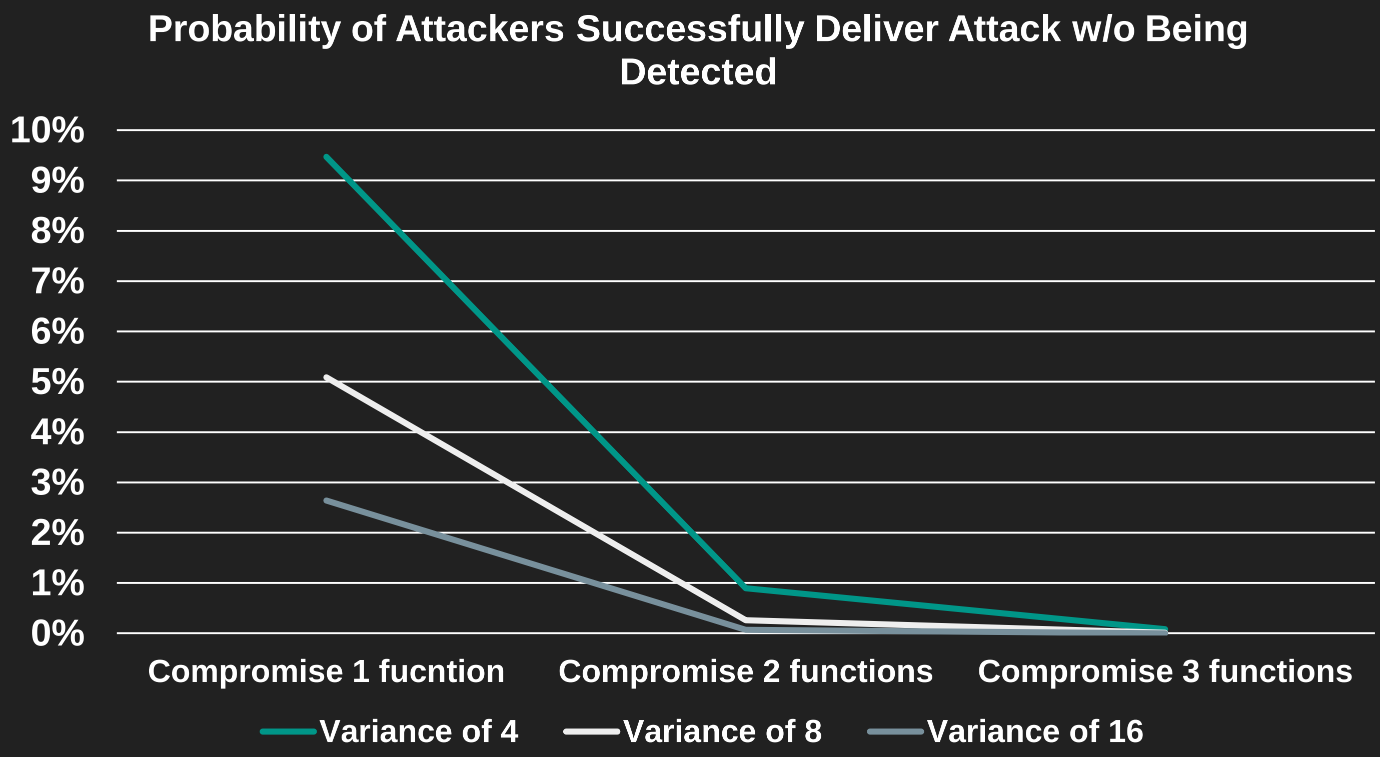
Security Analysis

- The attacker learns quickly
  - Learns about any configuration after this very configuration has been run only once
- The attacker doesn’t trigger any exception by accident

Metrics:

What’s the chance for an attacker to successfully compromise our system without being detected?

Security Analysis



Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo

Conclusion

- Shadowclone can efficiently thwarts and detects DOP attacks.

- Shadowclone has low performance overhead when running small programs. Its performance deteriorates as the size of program gets larger and the program gets more function calls.

Question?

Probability of Attackers Successfully Deliver Attack w/o Being Detected

- P(attacker succeeds withoutb being attacked)=

$$\sum_{k=1}^{k=inf} P(the\ first\ (k-1)\ times\ failed\ and\ without\ being\ attacked) * P(the\ kth\ time\ succeeds\ and\ not\ being\ detected)$$

- P(the first time succeed) = 1/N! P(the kth time succeed)= 1/M
    - (N is the average number of stack variables in a function, M is the number of clones)
    - N = 10 in the benchmarks we analyzed
- P(an attack would be detected) = 1/2