# Pmp: Cost-effective forced execution with probabilistic memory pre-planning

Maple Hu

# Content

- Background
- Motivation
- Memory pre-planning scheme
- Evaluation
- Discussion

# Background: Forced execution

- Dynamic Analysis
- Without requiring inputs for target program
- Crash-free execution model

Amazing but not cheap!

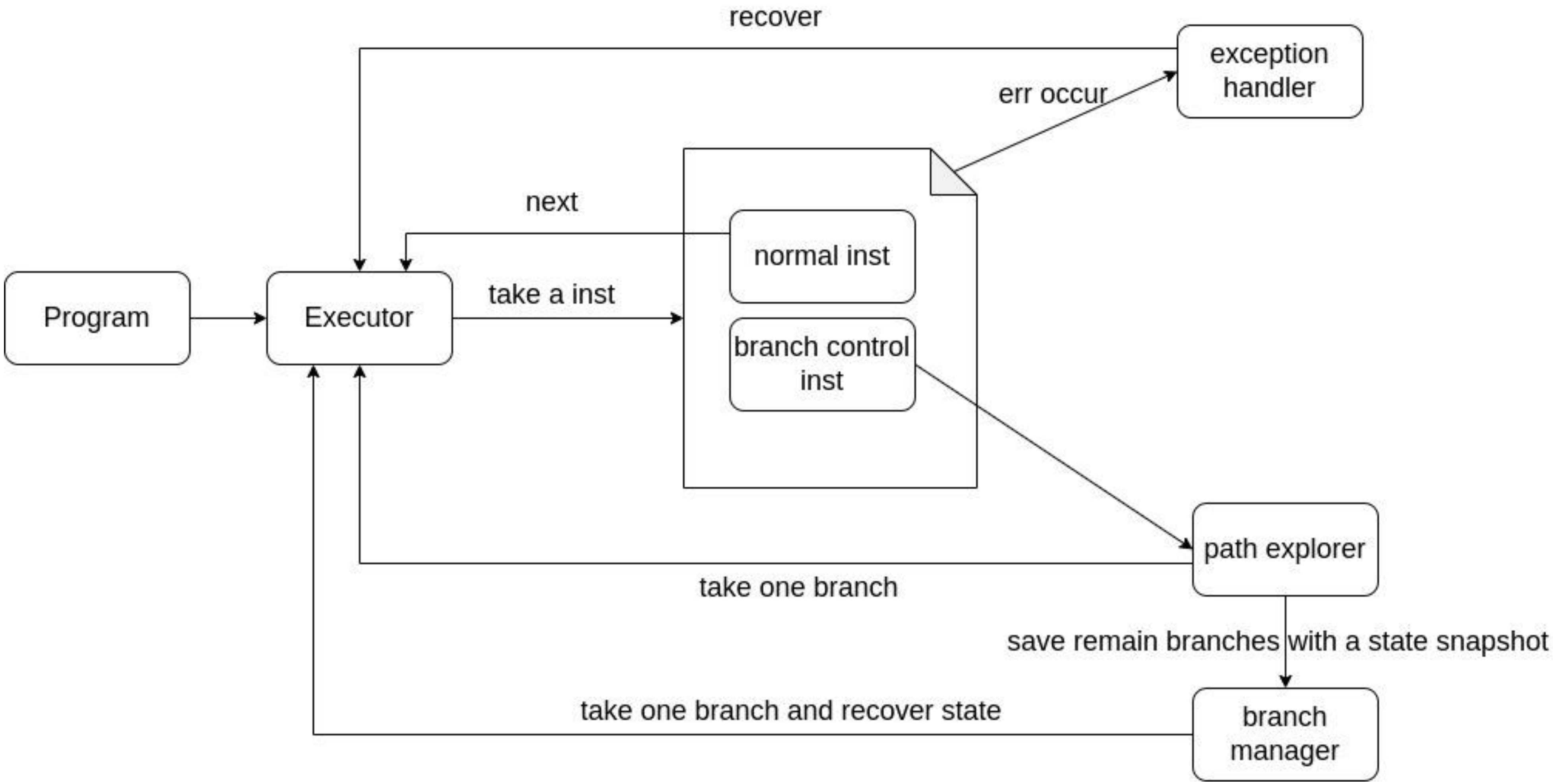
# Forced execution vs other dynamic analysis

- Dynamic symbolic execution or concolic execution
  - Testcases required
  - Symbolic and concrete execution selecting on the fly
  - SMT-solver required

- Fuzzing
  - Efficient testcases required
  - Testing

- Both not fit for detecting practical malware.

# A prototype of forced execution



# Some works

- *Exploring Multiple Execution Paths for Malware Analysis*. S&P'07.
- *A forced sampled execution approach to kernel rootkit identification*. RAID'07.
- *X-Force: Force-Executing Binary Programs for Security Applications*. Usenixsec'14
- *J-force: Forced execution on javascript*. WWW'17
- *SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets*. NDSS'21

# Motivation: two simple examples

```c
int test(int a, int b)
{
    if(a > b){
        foo = add;
    }else{
        foo = mul;
    }
    c = foo(a,b);
    return c;
}
```

Observation from CFG construction

```c
...
if(cmd == 0x101){
    //do something
}else if(cmd = 0x102){
    //do something
}else if(cmd = 0x103){
    //do something
}else if...
```

Observation from malwares

# Why forced execution for security?

- We want to get as many possible behaviors as we can, allow false positive.
  - Infeasible paths
- Suppose no pre-knowledge for any malware.
  - No pre-knowledge for inputs, or environment.
- Suppose adversary is powerful
  - Obfuscation
  - Anti-debugger

# Forced execution in binary (X-Forced)

- Crash-free execution model
  - Allocating memory on demand
  - No symbolic representation

- Consistency
  - Memory dependency (p  = q,  *p = a,  ….)
  - Control-flow dependency (if(x & 0x99){y = x ….})

# A simplified example in real-world

```
1  int main(){
2      int x = random_input();
3      int p, y, s;
4      if(C(x))
5          p = malloc(...);
6      if(D(x))
7          init(x, p)
8          table_put(x, p)
9      }
10     ...
11     table_put(...);
12     ...
13     //assume(x == y)
14     s = table_get(y);
15     if(s){
16         //trigger interesting behavior
17     }
18 }
```

- Normal run: feed a random int to x, suppose the predicate in 4, 6, 15 both are *false*.
- Forced run: run again with old x,  and we set the line 6 predicate to be *true* forcedly.  Note the pointer p is null in line 7, so a memory err will be occurred when *init* function modifies p.

  Under Allocating memory on-demand scheme, X-force will handle such err:
  - allocate a memory region for p.
  - resume execution.

Finally we get in the line 16!

# Consistency

- Memory dependency(p = q, …, *p = a):
  - If p is allocated on-demand, then q should also point new memory.
  - If p is type (int*), we should allocate 4 bytes at least for p.
  - If there is a statement q1 = p + 16 in the middle, we should allocate 68 bytes at least for p.

*We say two variables are **linearly correlated** if the value of one variable is computed from the value of the other variable by **adding or subtracting** a value.*

# Linear set computation rules

Table 1: Linear Set Computation Rules.

| Statement | Action[1,2] | Rule |
|---|---|---|
| initially | `foreach` (global address $t$)<br>`if` ($isAddr(*t)$) $SM(t) = \{t\}$; | L-INIT |
| $r := \mathtt{R}(r_a)$ | $SR(``r") \to nil$;<br>`if`($SM(r_a)$) $SR(``r") \to SM(r_a)$; | L-READ |
| $\mathtt{W}(r_a, r_v)$ | `if` ($SM(r_a)$) $SM(r_a) = SM(r_a) - \{r_a\}$<br>$SM(r_a) \to nil$;<br>`if` ($SR(``r_v")$)<br>$SR(``r_v") = SR(``r_v") \cup \{r_a\}$;<br>$SM(r_a) \to SR(``r_v")$; | L-WRITE |
| $r := a$ | $SR(``r") \to \{\}$ | L-ADDR |
| $r := c$<br>/*!$isAddr(c)$*/ | $SR(``r") \to nil$ | L-CONST |
| $r := r_1 + / - r_2$ | `if` ($!(isAddr(r_1) \&\& isAddr(r_2))$)<br>$SR(``r") \to nil$<br>`if` ($isAddr(r_1)$) $SR(``r") \to SR(``r_1")$;<br>`if` ($isAddr(r_2)$) $SR(``r") \to SR(``r_2")$; | L-LINEAR |
| $r := r_1 * / ... r_2$ | $SR(``r") \to nil$ | L-NON-LNR |
| `free`($r$) | $t = r$;<br>`while` ($accessible(t)$)<br>`if` ($SM(t)$) $SM(t) = SM(t) - \{t\}$;<br>$t++$; | L-FREE |

1. The occurrence "$r$" denotes the symbolic name of register $r$, the occurrence of $r$ denotes the value stored in $r$.
2. Operator "=" means set update, "$\to$" means pointer update.

Table 2: Memory Error Prevention and Recovery.

| Statement | Action | Rule |
|---|---|---|
| $r :=$ `malloc`($r_1$) | `for` ($i = r$ `to` $r + r_1 - 1$)<br>$accessible(i) = true$ | M-ALLOC |
| `free`($r$) | $t = r$;<br>`while` ($accessible(t)$)<br>$accessible(t) = false$<br>$t++$; | M-FREE |
| $r := \mathtt{R}(r_a)$ | `if` ($!accessible(r_a)$)<br>$recovery(r_a)$; | M-READ |
| $\mathtt{W}(r_a, r_v)$ | `if` ($!accessible(r_a)$)<br>$recovery(r_a)$; | M-WRITE |

Linear sets: the values stores in these addresses are linear correlated
SR: regs -> linear sets
SM: addresses -> linear sets

Tracking all memory operations is too expensive! It was reported that X-Force has 473 times runtime overhead over the native execution.

# Memory pre-planning scheme

- Allocating a memory region before running, instead of allocating new memory block on demand. That is
  - We allocate a memory region with fixed size.
  - For invalid memory reading e.g., ...=*p, we assign p an address located in pre-allocated memory.
- Two critical questions:
  - How to guarantee the address arithmetic will be safe?
  - How to guarantee the double dereference operation of the address will be safe?

To sum up, we need to guarantee the result of pointer operations will fail in the allocated memory again, we call this self-contained memory behavior .

# A simplest crafted memory region

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x0000 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x0010 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x0020 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

1. Suppose p is null(0), then *p, *(p+8), **p are both 0.

2. Note mapping address 0 to be valid in user address space should be ok when u can control the system kernel.

3. But such scheme will lead to many substantial wrong program dependencies as semantically unrelated memory operations through uninitialized/invalid pointer variables all end up accessing address 0x00.

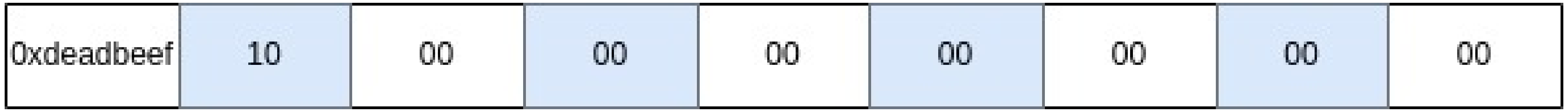
# A carefully crafted memory region

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x0000 | 80 | fe | 00 | 00 | 00 | 00 | 00 | 00 | 50 | 38 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x0010 | 48 | 74 | 00 | 00 | 00 | 00 | 00 | 00 | f8 | 04 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x0020 | d0 | ff | 00 | 00 | 00 | 00 | 00 | 00 | 08 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| | | | | | | | | | ...... | | | | | | | |
| 0xffd0 | 88 | 19 | 00 | 00 | 00 | 00 | 00 | 00 | 30 | 30 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0xffe0 | 40 | fc | 00 | 00 | 00 | 00 | 00 | 00 | 98 | 20 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0xfff0 | 20 | 50 | 00 | 00 | 00 | 00 | 00 | 00 | e8 | a7 | 00 | 00 | 00 | 00 | 00 | 00 |

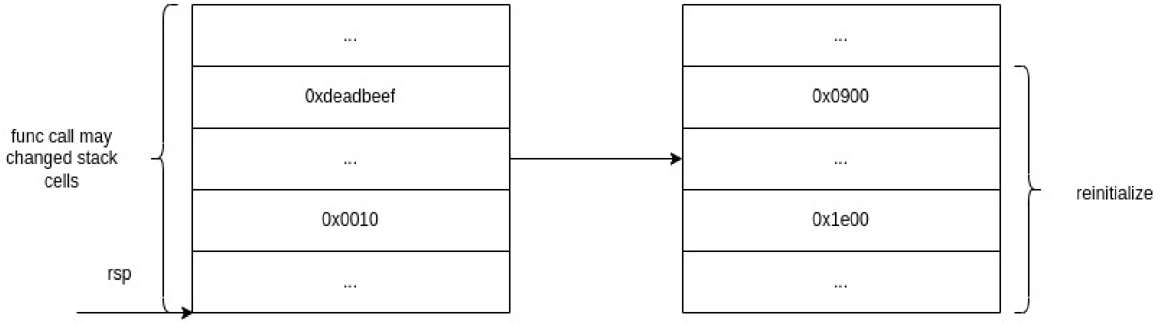
- Proper random values are filled in different bins.
- Initializes global, local variables, and heap regions allocated by original program logic with random values point to the pre-allocated memory.

# Initialization

- Global variable initialization
  - Initializing the .bss segment(uninitialized or zero-initialized global variables) with addresses in the pre-allocated memory region.
- Heap region initialization
  - Intercepting the memory allocation operation.  Let allocated region contain random word-aligned values that the addresses in pre-allocated memory region.

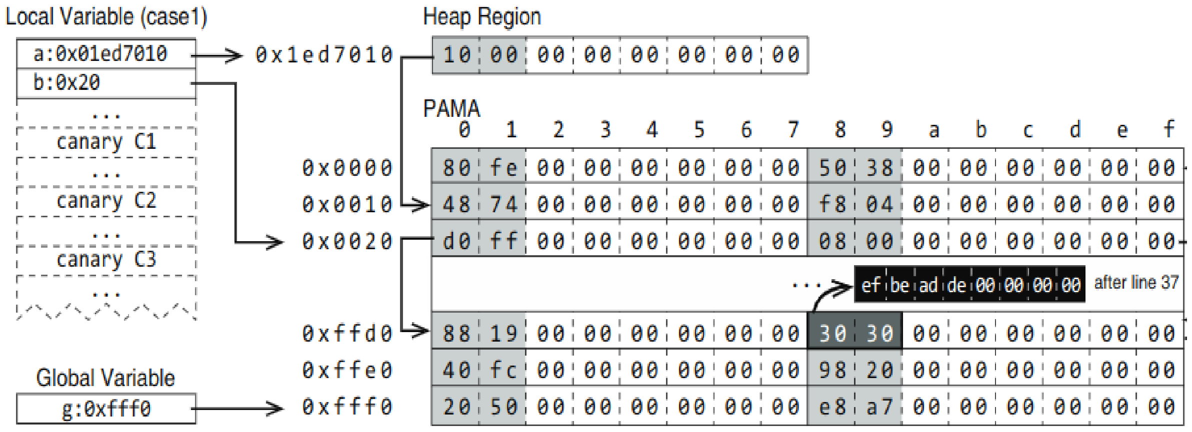



- Local region initialization
  - Initializing entire stack region be a heap region.
  - Need reinitialization for runtime function call.
  - Random testing for identifying stack frame



# A example with pre-allocated memory

```c
typedef struct{double *f1; long *f2;} T;
typedef struct{char f3; long *f4; long *f5;} G;
void case1() {
    long **a = malloc(...);
    T *b;
    if (cond1()) init(b);
    if (cond2()) {
        long *alias = b->f2;
        *(b->f2) = **a; // [0x0008] = [0x0010]
        *(b->f1) = 0.1; // [0xffd0] = 0.1
        long tmp = *alias;
    }
}
```




1. global variable g points to 0xfff0,  b is points to 0x20 and a is allocated memory contains 0x0010.
2. &(b->f1)  is 0x20 and &(b->f2) is 0x24.
3. b->f1 is point 0xffd0, b->f2 is point 0x0008
4. *a is point 0x0010,

# Two obvious problems

- Memory read and write may out of bound.
- Suppose pointer variable c points one address in pre-allocated memory, we say 0x1000.  Then *c = 0xdeadbeef,  and suppose there is unrelated pointer variable d also points to 0x1000. If the program run dereference  **d, the invalid memory read will occur for read from 0xdeadbeef.

The probability of such situations can be very low,  if we carefully construct the memory scheme.

# Probability Analysis

- Let $PA$ be the set of all possible address within pre-allocated memory region, and $WA$ be its word-aligned subset. Let $S$ be the size of pre-allocated memory region. So we have $S$ = $|PA|$ = $|WA|$ × $|8|$. Let $FV$ be a random subset of $WA$, whose elements are used as values to be filled in pre-allocated memory region. Let $d$ be the ratio $\frac{FV}{WA}$.

  Let $x$ be the a filling value selected from FV, $\alpha$ be an offset. The probability $P_{oob}$ of $x + \alpha$ being out of the bound of pre-allocated memory region is

$$P_{oob} = \Pr[(x + \alpha) \notin PA \mid x \in FV] = \frac{\alpha}{S - 8} \cdot \left(1 - \frac{8}{d \cdot S}\right)$$

  Let $x$ and $y$ be two filling values independently selected from $FV$. The probability $\mathrm{P}_{cac}$ of coincidental address collision, when $x$ and $y$ have the same value, is

$$\mathrm{P}_{cac} = \Pr[x = y \mid x \in FV, y \in FV] = \frac{1}{|\mathrm{FV}|} = \frac{8}{d \cdot S}$$

# Some “fatal errors”

- Target program may set semantically unrelated pointers to null after these pointers were initialized.
  - For example, local variables will be initialized with addresses in the pre-allocated memory region at the start of function call, if the function initialize these variables again, it will lead to program dependencies between these variables.
- Large pre-allocated memory region support.
  - Use the QEMU user-mode emulation mapping mechanism.
- Misaligned memory access.
  - No better way for this memory pre-planning scheme at the time.

```
foo(){
   int * p = NULL;
   int * q = NULL;
   ....
}
```

# Evaluation: experiment results

TABLE I: SPEC2000 Results

| Benchmark | PMP | | | | | | | | | X-Force | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | execution status | | | code coverage | | | memory dependence | | | execution status | | | code coverage | | | memory dependence | | |
| | time (s) | # run | # fail | # insn | # block | # func | # found | # correct | # mistyped | time (s) | # run | # fail | # insn | # block | # func | # found | # correct | # mistyped |
| 164.gzip | 24.6 | 382 (15.6/s) | 11 (3%) | 7,650 (100%) | 699 (99%) | 61 (100%) | 3,529 | 2,824 (80%) | 0 (0%) | 2,112 | 369 (0.17/s) | 10 (3%) | 7,420 (97%) | 669 (95%) | 61 (100%) | 3,662 | 2,343 (64%) | 28 (1%) |
| 175.vpr | 76.8 | 1,006 (13.1/s) | 82 (8%) | 26,783 (83%) | 2,007 (71%) | 226 (89%) | 13,418 | 8,983 (67%) | 333 (2%) | 9,436 | 1,000 (0.10/s) | 79 (8%) | 26,677 (83%) | 2,004 (70%) | 226 (89%) | 13,332 | 7,199 (57%) | 2,428 (18%) |
| 176.gcc | 3490.2 | 26,524 (7.6/s) | 822 (3%) | 186,310 (49%) | 16,104 (44%) | 1,239 (65%) | 573,375 | 384,161 (67%) | 11,467 (2%) | 347,014 | 26,647 (0.08/s) | 799 (3%) | 183,280 (48%) | 16,098 (43%) | 1,221 (64%) | 573,926 | 332,303 (58%) | 63,131 (11%) |
| 181.mcf | 8.6 | 144 (16.7/s) | 2 (1%) | 2,977 (100%) | 213 (100%) | 24 (100%) | 1,718 | 1,248 (73%) | 0 (0%) | 374 | 164 (0.43/s) | 2 (1%) | 2,947 (99%) | 213 (100%) | 24 (100%) | 1,487 | 1,011 (68%) | 130 (9%) |
| 186.crafty | 860.3 | 2,753 (3.2/s) | 15 (0.5%) | 40,404 (96%) | 4,237 (96%) | 104 (100%) | 22,437 | 14,300 (64%) | 20 (0.08%) | 99,764 | 2,830 (0.03/s) | 13 (0.4%) | 41,685 (99%) | 4,381 (99%) | 104 (100%) | 22,816 | 12,092 (53%) | 2,749 (12%) |
| 197.parser | 98.2 | 1,590 (16.2/s) | 68 (4%) | 22,093 (90%) | 2,688 (92%) | 279 (94%) | 9,958 | 6,664 (67%) | 887 (9%) | 6,340 | 1,685 (0.27/s) | 69 (4%) | 23,331 (95%) | 2,799 (96%) | 288 (97%) | 11,740 | 5,870 (50%) | 3,682 (31%) |
| 252.eon | 37.2 | 707 (19.0/s) | 27 (4%) | 28,600 (71%) | 5,560 (70%) | 502 (82%) | 9,521 | 4,457 (47%) | 142 (1%) | 4,020 | 659 (0.16/s) | 26 (4%) | 27,622 (69%) | 5,413 (68%) | 501 (81%) | 9,121 | 3,557 (39%) | 5,669 (62%) |
| 253.perlbmk | 1,189 | 10,318 (8.7/s) | 508 (5%) | 118,135 (88%) | 11,600 (90%) | 692 (97%) | 66,726 | 28,394 (43%) | 4,001 (6%) | 176,096 | 10,400 (0.06/s) | 502 (4%) | 119,467 (89%) | 11,676 (90%) | 696 (97%) | 70,611 | 24,713 (35%) | 18,866 (27%) |
| 254.gap | 1,054 | 7,754 (7.3/s) | 310 (4%) | 49,869 (54%) | 4,519 (50%) | 401 (88%) | 38,243 | 20651 (54%) | 3,059 (8%) | 103,458 | 7,461 (0.07/s) | 298 (4%) | 49,920 (54%) | 4,521 (50%) | 401 (88%) | 38,784 | 18228 (47%) | 6,593 (17%) |
| 255.vortex | 487.0 | 7,232 (14.9/s) | 157 (2%) | 100,718 (92%) | 15,513 (91%) | 577 (92%) | 55,205 | 19,939 (36%) | 630 (1%) | 58,646 | 7,223 (0.12/s) | 132 (2%) | 100,652 (92%) | 15,489 (91%) | 577 (92%) | 54,977 | 15,393 (28%) | 14,072 (26%) |
| 256.bzip2 | 16.0 | 249 (15.6/s) | 13 (5%) | 6,338 (92%) | 545 (94%) | 60 (95%) | 2,755 | 2,375 (86%) | 0 (0%) | 842 | 258 (0.19/s) | 11 (4%) | 5,179 (76%) | 471 (82%) | 53 (84%) | 2,434 | 1,849 (76%) | 215 (9%) |
| 300.twolf | 221.4 | 2,972 (13.4/s) | 97 (3%) | 52,351 (91%) | 3,682 (86%) | 165 (99%) | 24,032 | 10,333 (43%) | 528 (2%) | 21,308 | 2,997 (0.14/s) | 90 (3%) | 52,831 (92%) | 3,749 (88%) | 165 (99%) | 25,664 | 8,212 (32%) | 3,132 (12%) |
| Average | - | 12.6/s | 3.5% | 83.8% | 79.1% | 91.8% | - | 60.6% | 2.6% | - | 0.15/s | 3.4% | 82.7% | 81.0% | 90.9% | - | 50.6% | 19.6% |

# Discussion

- ????