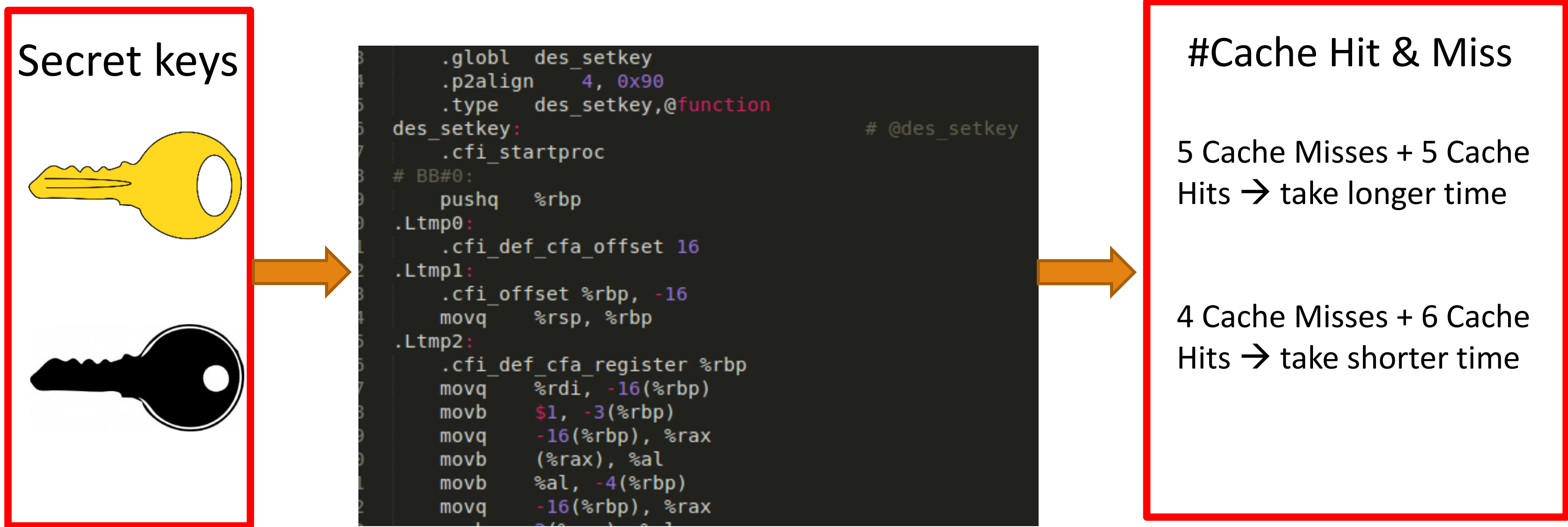
Program Repair for Elimilating Timing Side-channel Leaks

Meng Wu, Shengjian Guo,Patrick Schaumont, Chao Wang

Cache-based Timing Channel Attack



By observing the number of cache misses or the memory access timing variations, attackers can infer which secret key is fed to the program.

Existing Mitigation Techniques

Partitioned Cache
Randomized Cache
Line-locking Cache

Secret Independent
Control Flow
Manual Modification

Contribution

- Statically detect Cache-based timing leakage in crypto software.
- Automatically mitigate Cache-based timing leakage.
- Use GEM5 simulator to confirm mitigated code is leakage free.

Motivating Example

```
//Suppose: a is secret data
uint8_t foo(uint8_t a){
  uint8_t b;
  for (uint8_t i = 0; i < 8; i++){
    b = a & 0x80; // b is also secret
    a <<= 1;
    if (b == 0x80) // sensitive branch
      a ^= 0x1b;
  }
  return a;
}
```

```
uint8_t foo(uint8_t a){
  uint8_t b, a_1=0;
  for (uint8_t i = 0; i < 8; i++){
    b = a & 0x80;
    a <<= 1;
    a = (b == 0x80)? a^0x1b : a;
  }
  return a;
}
```

Leak due to Secret Dependent Control Flow

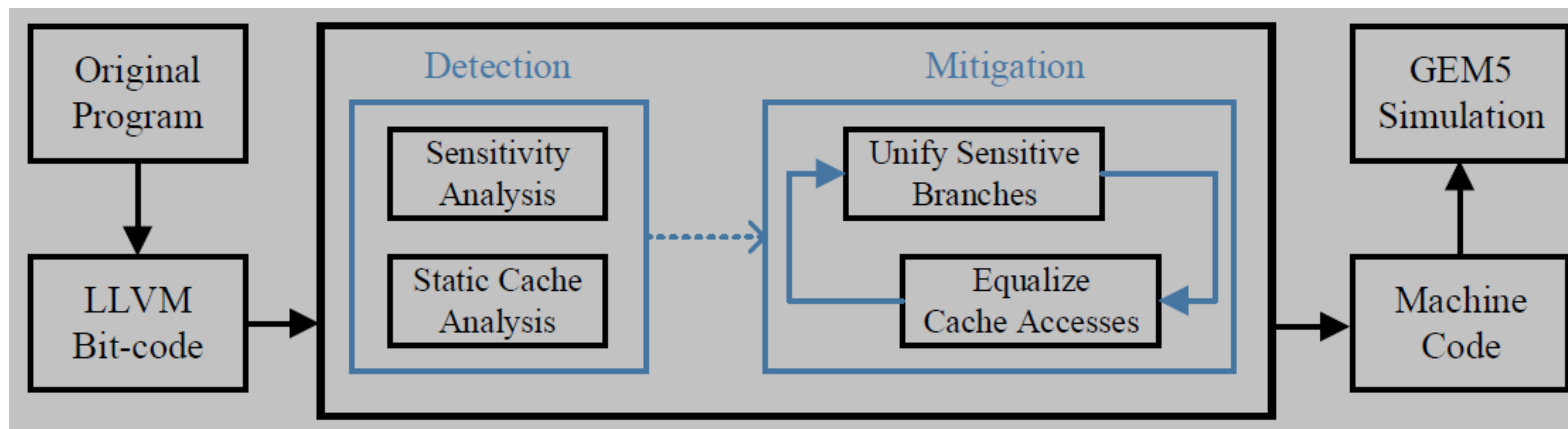
Motivating Example

```
const uint8_t sbox[256] = {
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
  0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
  ...};
void subBytes(uint8_t *block){
  uint8_t i;
  for (i = 0; i < 16; ++i)
    block[i] = sbox[block[i]];
}
```

Leak due to secret dependent addressing

```
void subBytes(uint8_t *block){
  uint8_t temp, i, j;
  for (i = 0; i < 16; ++i){
    j=255;
    do{
      temp = sbox[j];
      block[i] = (block[i] == j)? temp : block[i];
    }while(j--)
  }
}
```

Overview of Our Approach



- Sensitivity Analysis: only require user specified sensitive input as sensitivity seed, automatically propagate secret information.
- Static Cache Analysis: to optimize out redundant mitigation.
- GEM5 simulator to further confirm the effectiveness of our method.

Sensitivity Analysis for Detecting Timing Variations

Static sensitivity analysis to capture instructions that may cause timing variations.

Sensitivity propagation rules:

%des = **load** %src or **store** %src %des

%des **= op** %op1, %op2

%des = **trunc** i64 %src to i32

br i1 %cmp, label %bb1, label %bb2

%addr = **getelementptr** @sbox, %index

Sensitivity Analysis for Detecting Timing Variations

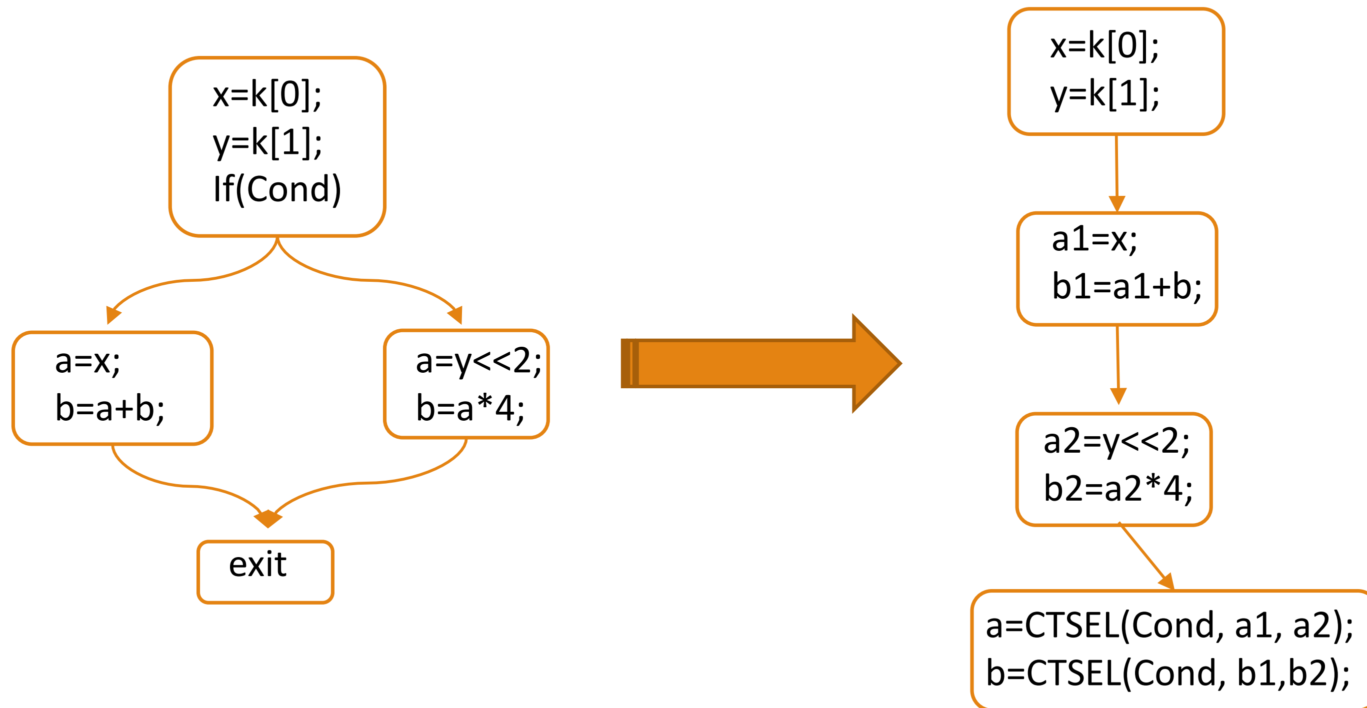
Static sensitivity analysis to capture instructions that may have leakage.

Detection rules (over-approximation):

```
br i1 %cmp, label %bb1, label %bb2
```

```
%addr = getelementptr @sbox, %index
```

Mitigate Sensitive Branches



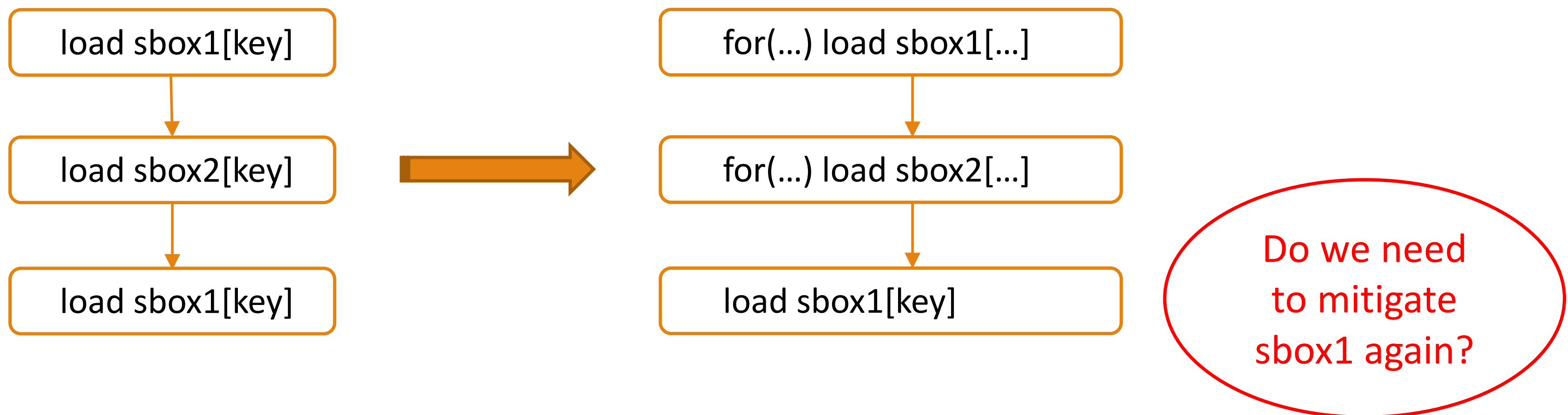
- Traverse the CFG to mitigate Sensitive Branches
- Create Dummy Assignments to replace both branches.
- Use CTSEL intrinsic to assign real value.

-O1: cache line oblivious optimization

```
void subBytes(uint8_t *block){
  uint8_t i, j, val;
  for (i = 0; i < 16; ++i){
    uint8_t *index = block+i;
    uint8_t pos = *index % CS;
    for (j = 0; j < CL; ++j){
      val = sbox[pos];
      *index = (*index == pos) ? val : *index;
    }
  }
}
```

- Access all elements in the array is too expensive!
- Cache are always accessed line by line, so no need to load every element.
- Only need to make sure to touch every cache line to remove redundant access.

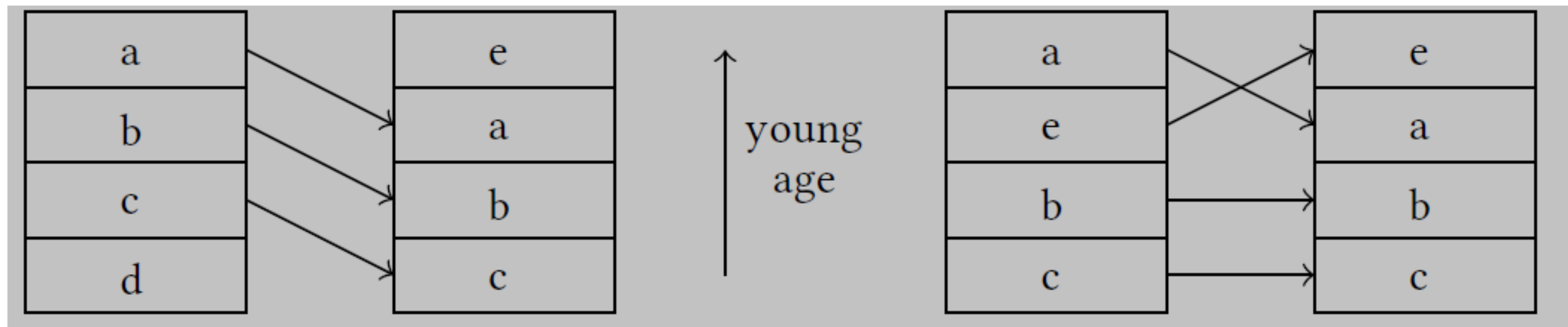
-O2: static cache analysis optimization



- Leverage a conservative static cache analysis to remove more redundant mitigation

-O2: static cache analysis optimization

LRU Must-Analysis: variable definitely cached

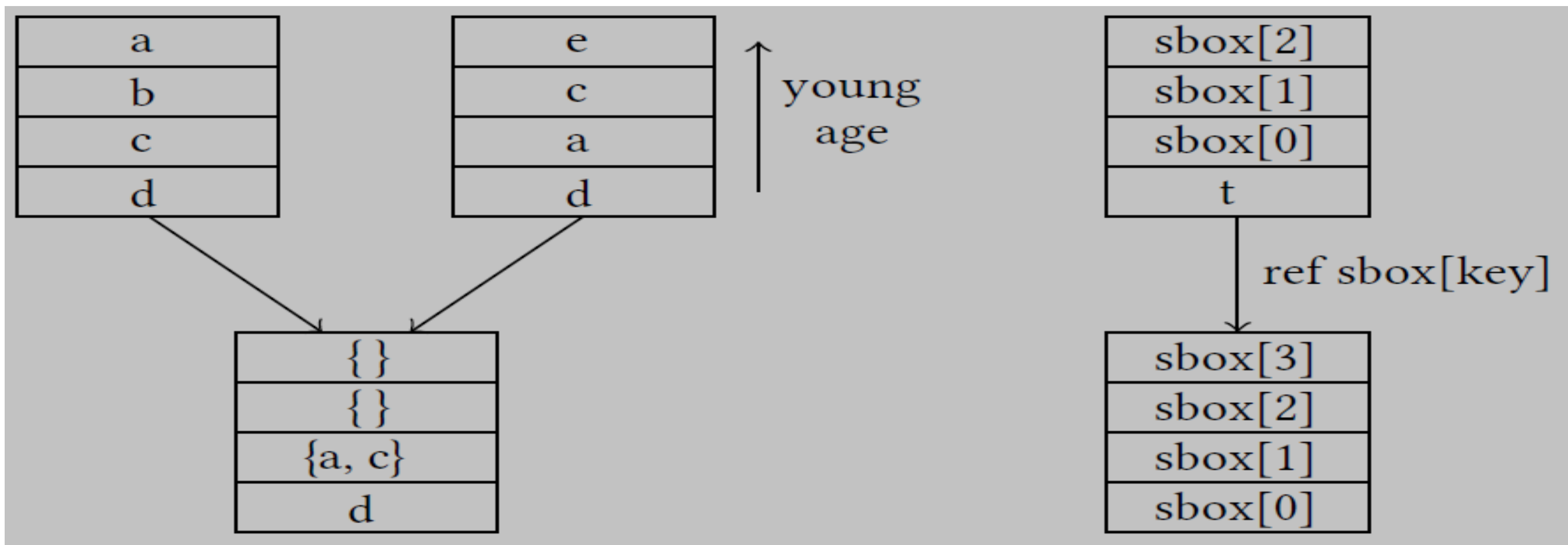


Update of cache model with LRU cache policy

- An under-approximation on cache content.
- Implemented as an LLVM pass.

-O2: static cache analysis optimization

LRU Must-Analysis: variable definitely cached



Merge of program paths and conservative update undecidable reference.

- An under-approximation on cache content.
- Implemented as an LLVM pass.

Effectiveness of Optimizations

Method	Total access	#cache miss	#cache hit
Original	K	[M, 1]	[K-M, K-1]
Naive	K*N	M	K*N-M
Cache Line Oblivious	K*M	M	M(K-1)
Multi-access	M+K-1	M	K-1

K -- #accesses of array in the original program
N -- array length
L -- Cache line size
M – [N/L], the number of cache lines to store the array

Benchmarks

Program	Source	Desc	LOC	LUT Size(byte)	Branch
aes	Chronos	AES Algorithm	1,373	16,424	0
des	Chronos	DES Algorithm	823	6,656	0
des3	Chronos	DES-EDE3 Algorithm	841	6,656	0
anubis	Chronos	Anubis Algorithm	646	6,220	0
cast5	Chronos	Cast5 cipher algorithm (rfc2144)	780	8,192	0
cast6	Chronos	Cast6 cipher algorithm (rfc2612)	533	4,896	0
fcrypt	Chronos	FCrypt encryption algorithm	375	4,096	0
khazad	Chronos	Khazad Algorithm	865	16,456	0
LBlock	FELICS	L-block cipher	500	160	0
Piccolo	FELICS	Piccolo primitive cipher	237	148	0
PRESENT	FELICS	PRESENT Cipher	178	2064	0
TWINE	FELICS	TWINE cipher	219	67	0
aes	SuperCop	AES Algorithm	1,178	8,488	0
cast	SuperCop	CAST algorithm	897	16,384	0
…					

- Widely used cipher implementation from reputable source.
- 19,708 lines of C/C++ code in total
- Look-up Table size from 67 bytes to 16k+ bytes.

Result: Detection

Program	Total			Sensitivity		
	#IF	#LUT	#LUT-access	#IF	#LUT	#LUT-access
aes	3	15	424	0	4	416
des	2	11	640	0	11	640
des3	2	11	1152	0	11	1152
anubis	1	7	871	0	6	868
cast5	0	8	448	0	8	448
cast6	0	6	448	0	4	384
fcrypt	0	4	128	0	4	128
khazad	0	9	240	0	8	248
3way	10	0	0	3	0	0
des	16	14	456	2	8	128
Loki91	10	1	512	4	0	0

Result: Mitigation

Program	Mitigation w/o –O2				Mitigation w/ -O2			
	#LUT-a	Time(s)	Prog-size	Ex-time	#LUT-a	Time(s)	Prog-size	Ex-time
aes	416	0.61	5.40x	2.70x	20	0.28	1.22x	1.11x
des	640	1.17	19.50x	5.68x	22	0.13	1.23x	1.07x
des3	1,152	1.8	12.90x	12.40x	22	0.46	1.13x	1.07x
anubis	868	3.12	9.08x	6.90x	10	0.75	1.10x	1.07x
cast5	448	0.79	7.24x	3.84x	12	0.22	1.18x	1.07x
cast6	384	0.72	7.35x	3.48x	12	0.25	1.16x	1.08x
fcrypt	128	0.07	5.70x	1.59x	8	0.03	1.34x	1.05x
khazad	248	0.45	8.60x	4.94x	16	0.07	1.49x	1.35x
3way	0	0.01	1.01x	1.03x	0	0.01	1.01x	1.03x
des	128	0.05	2.21x	1.22x	8	0.03	1.09x	1.11x
Loki91	0	0.01	1.01x	2.83x	0	0.01	1.01x	2.83x

- General cache configuration.
- Improvement of Optimization.
- Acceptable overhead on code size and execution time

Result: Simulation

Program	Before Mitigation				Mitigation w/o Opt		Mitigation w/ Opt	
	#CPU Cycles(in1, in2)		#Miss(in1, in2)		#CPU Cycles	#Miss	#CPU Cycles	#Miss
aes	100,554	101,496	261	269	204,260	303	112,004	303
des	95,630	90,394	254	211	346,170	280	100,694	280
des3	118,362	111,610	271	211	865,656	280	124,176	280
anubis	128,602	127,514	276	275	512,452	276	134,606	276
cast5	102,426	102,070	282	279	266,156	304	108,068	304
cast6	96,992	97,492	238	245	233,774	245	100,914	245
fcrypt	84,616	83,198	224	218	114,576	240	88,236	240
khazad	101,844	100,724	332	322	366,756	432	130,682	432
3way	87,834	87,444	181	181	90,844	182	90,844	182
des	152,808	147,344	224	222	181,074	225	168,938	225
Loki91	768,064	768,296	181	181	2,170,626	181	2,170,626	181

- Simulated with GEM5 on **cycle-accurate CPU** model
- After mitigation: Identical number of CPU cycles and cache misses

Questions