BINARY OBFUSCATION THAT DOESN'T KILL LTO:

MAKING FUNCTION LAYOUT RANDOMIZATION POSSIBLE FOR THE SWITCH

PRESENTED BY: FARZON LOTFI

MANDATORY ADMINISTRATE

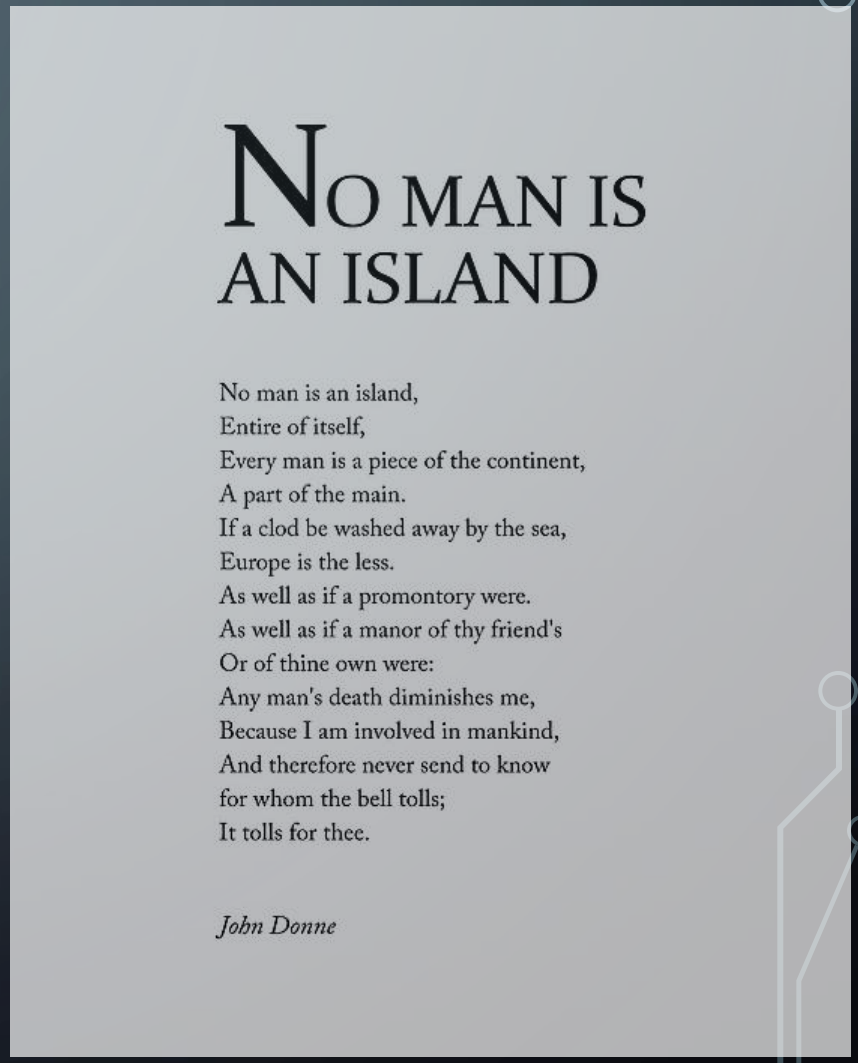
- Any opinions expressed in this talk are my own and do not represent the opinions of my employer.
- If you want bootleg ninja cat and master chief stickers find me after this talk.
- If you want printed photo stickers also find me.

BACKGROUND

- Compiler engineer with a bachelor's & master's from Georgia Tech.
- At Microsoft focused on browser security, GPU compilers, and language design.
- Formerly at Blizzard, working on large-scale anti-cheat systems.
- Passionate about the intersection of security and performance.
- @farzonl on github @noztol most everywhere else.

MOTIVATIONS

- I really like teaching the things I learn
- Moreover, I want you all to know with the right mindset, lexicon, and tools you can do this too.
- Finally, what better way to demonstrate that “No Man is an Island” I teach “you” so “you” can teach others.
- If this talk inspires “you” to do something better, then it did its job.

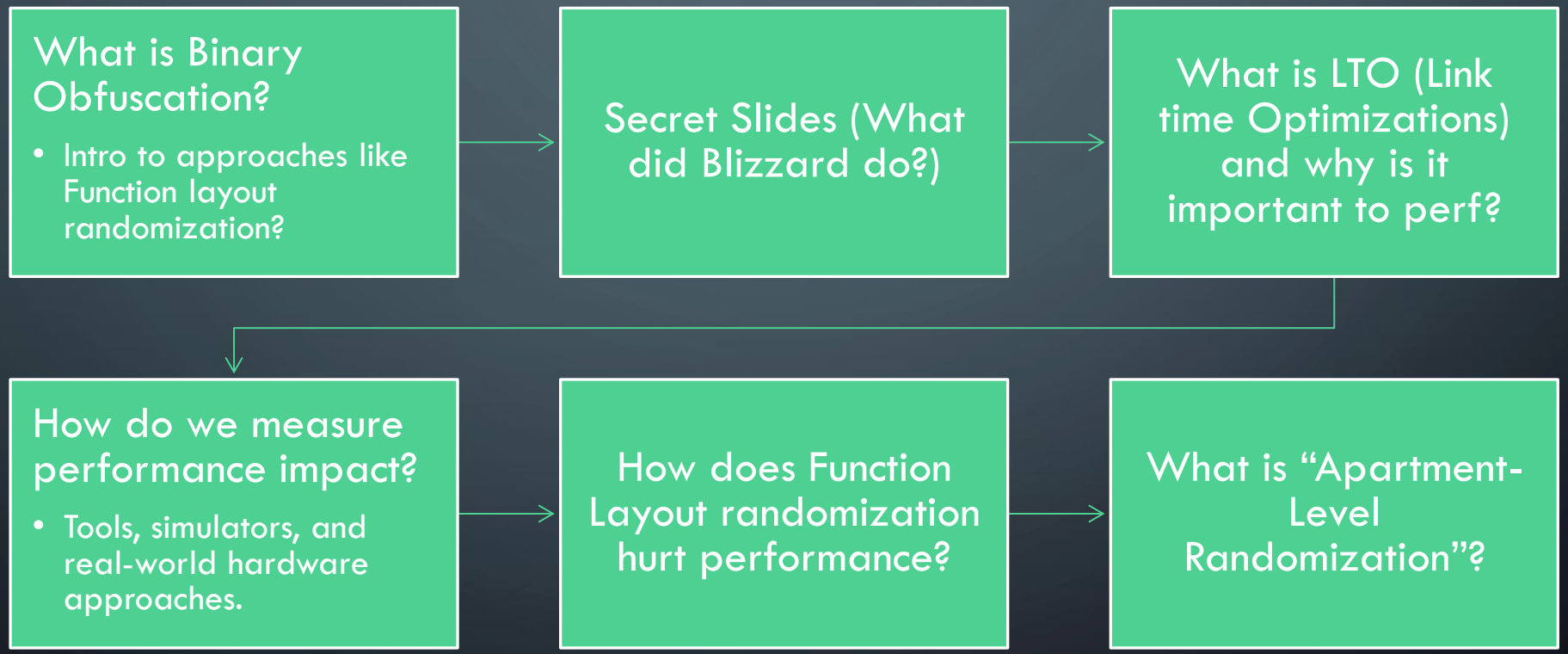


No man is an island

No man is an island,
Entire of itself,
Every man is a piece of the continent,
A part of the main.
If a clod be washed away by the sea,
Europe is the less.
As well as if a promontory were.
As well as if a manor of thy friend's
Or of thine own were:
Any man's death diminishes me,
Because I am involved in mankind,
And therefore never send to know
for whom the bell tolls;
It tolls for thee.

John Donne

OUTLINE (QUESTIONS THIS TALK ANSWERS)



ACRONYMS

- LLVM - Low Level Virtual Machine (A Framework for Compiler CodeGen)
- CodeGen – Code Generation
- LTO - Link Time Optimization
- CLANG – C Language (A C\C++ compiler that uses LLVM for CodeGen)
- IR – Intermediate Representation
- VEX IR - Valgrind EXpression Intermediate Representation
- Basic Block- a straight-line of code instructions with no branches
- Ida – A commercial disassembler

WHAT IS BINARY OBFUSCATION (1/2)?

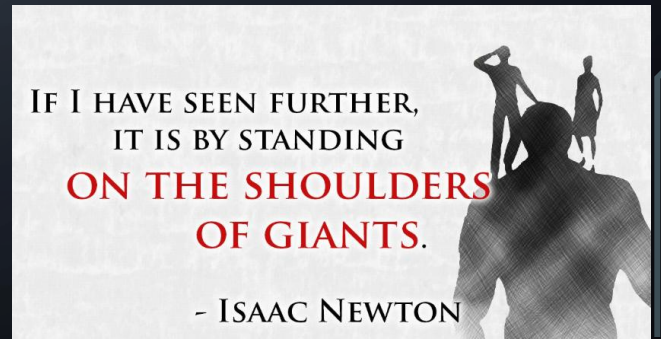
- Transforming compiled binaries to make reverse engineering and static analysis harder.
- Examples
  - Anti-Disassembly: Software will puke with unexpected inputs hardware tends to treat as no-ops.
    - Easiest with x86 because of variable length instruction set and alignment requirements.
    - Goal is to confuse or break disassemblers with illegal or undefined instructions or overlapping instructions.
  - Anti-VM: Detect if you are in a VM to evade analysis by cheat engines or gold miners.
    - Most VM software does not hide this fact. VM finger printing can be done via:
      - BIOS strings, CPUID Instruction, Process and File Detection, CPU vendor strings, or Virtual devices
  - Anti-Debug: Detect, Mislead, and break debuggers.
    - Detection: API-Based or Debug register based. Mislead: trigger exceptions code paths, Break: Anti-Breakpoints or self debug (attach before)

WHAT IS BINARY OBFUSCATION (2/2)?

- Transforming compiled binaries to make reverse engineering and static analysis harder
- Examples
  - Dataflow obfuscation: Make it difficult to statically analyze movement of data.
    - Examples: Encoding and Decoding Values (x = (a + b) ^ 0xA5;), Dead or redundant assignment, or Intermediate Computation Chains
  - Control flow obfuscation: Conceal the logical structure of a program in order to complicate discovery of high-level logic via disassemblers, decompilers, or symbolic execution. Techniques include branch flattening, function inlining/outlining, indirect brancing (ie gotos, switch tables), **function layout randomization**.
  - Packers techniques used to compress, encrypt, or transform an executable to hide its original code and data until runtime.

LOOKING TO DIP YOUR TOE?: READING LIST

- If these techniques interest you look up:
- Write Obfuscation:
  - https://www.apriorit.com/dev-blog/687-reverse-engineering-llvm-obfuscation (llvm: During compilation)
  - https://www.praetorian.com/blog/extending-llvm-for-code-obfuscation-part-1/
  - https://synthesis.to/2021/03/03/flattening_detection.html (VmProtect: post compilation)
- Reverse Engineer Obfuscation
  - http://www.oklabs.net/skype-reverse-engineering-the-long-journey/ (skype reverse engineering)
  - https://synthesis.to/2021/03/03/flattening_detection.html
  - https://napongizero.github.io/blog/Defeating-Code-Obfuscation-with-Angr (A concrete Angr tutorial)
- Write a Game cheat
  - Cheat Engine
    - https://www.cheatengine.org/ (software)
    - https://www.youtube.com/watch?v=Emiuht3YSXA (tutorial series by Stephen Chapman)
  - https://lonami.dev/blog/woce-1/ (8 part series)



DOES BINARY OBFUSCATION ALONE PREVENT GAME CHEATS?

- **NO!** It just makes it harder to pull off.
- It is one of many tools a Game Security Engineer will use to reducing cheating.
- ML is used actively these days in live FPS games to track head shot accuracy for auto bans.
- Some companies use kernel-based detections for process hijacking and automations click (gold) farming detections.

WHAT DID BLIZZARD USE?

- At Blizzard we wrote an internal tool called Aegis 2.0 (Wasn't there for 1.0)
- For reading and writing Binary formats like, PE, ELF, and Mach-O we used LIEF: https://github.com/lief-project/LIEF

- For x86/x86_64 disassembly we used XED https://intelxed.github.io/
- For ARM/ARM64 we used Capstone https://www.capstone-engine.org/

- Aegis 2.0 obfuscation passes took inspiration from LLVM's pass manager. We would lift assembly to an internal IR we could transform and write back.
- We tested our transformations against the clang unit test suite. If the unit tests still passed it was high confidence we didn't introduce codgen that would break games
- We never got exceptions transformations working
- Aegis 2.0 could do everything described in the last two slides.

WHY TALK ABOUT AEGIS NOW?

- Aegis is now deprecated.
  - These techniques will be lost to time unless we talk about them.
- Blizzard Became a subsidiary of my current employer last year
  - MSHR said talking about algorithms and approaches is fine.
  - The team that built Aegis is restructured (most left)

WHAT IS LTO?

- Stands for Link time optimization
- Optimization performed after all compilation units are linked
  - One of the few optimizations that sees the whole program
  - Enables cross-module inlining, Global dead code elimination, and Global constant folding
  - Improves performance significantly by:
    - Reduced binary size, eliminated unused functions.
    - Most importantly to this talk improves function layout to optimize for jump distance and cache locality
      - Especially important for tight loops or hot code paths where jumps between functions happen frequently.

LTO: FUNCTION LAYOUT OPTIMIZATIONS

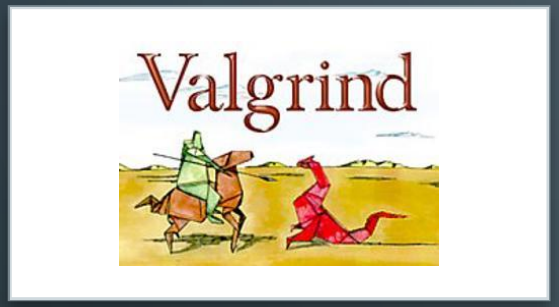
- Goal: Place frequently-called functions closer together
- Reduces:
  - Jump distance
  - Instruction cache misses
  - Page faults
- Maximizes performance on platforms with:
  - Limited cache
  - Strict memory budgets
  - Essential for consoles like the Nintendo Switch
- Function Layout Randomization obfuscations break this by undoing the optimization

HOW CAN WE DIAGNOSE PERF ISSUE?

- Measure What Matters
  - CPU cycles per function
  - Cache miss rates (i-cache, d-cache)
  - Branch mispredictions
  - Page faults
- These are all done via Dynamic Analysis

WHAT IS DYNAMIC ANALYSIS?

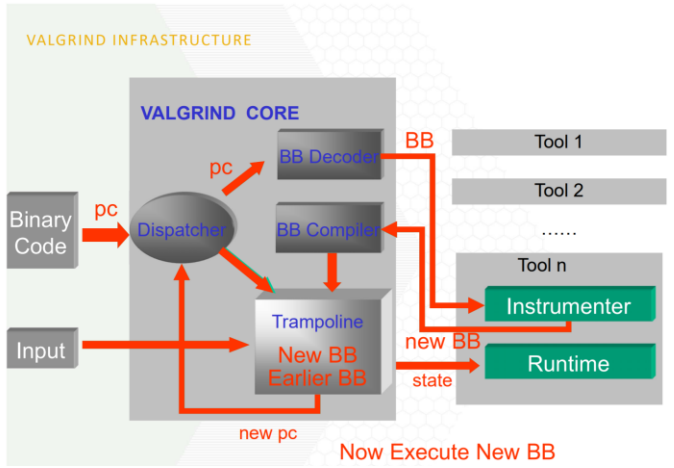
- Dynamic Binary Instrumentation: Given an original binary executable and an input, start executing the binary with the input, and during execution add instrumentation to the binary on the fly
- Instrument dynamically = during runtime
- Advantages for dynamic instrumentation:
  - No need to recompile or relink
  - Discover code at runtime
  - Handle dynamically-generated code
  - Attach to running processes

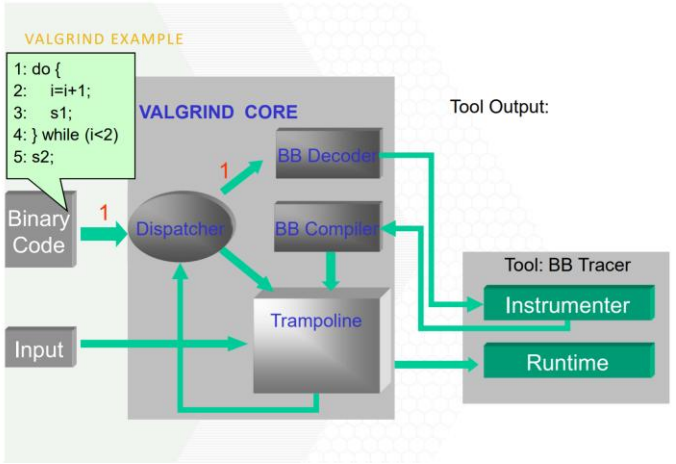


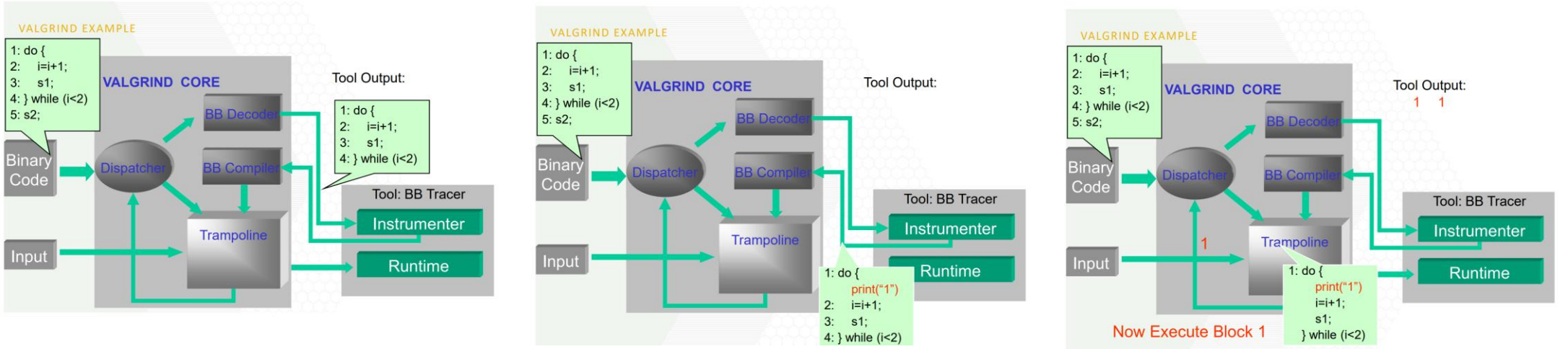
WHAT ARE THE TOOLS OF THE TRADE?

- We will look at:
  - Platform-specific:
    - instrumentation targets the exact instructions that are executing on the CPU
  - Emulation-based
    - instrumentation targets some intermediate language

VALGRIND A GIFT FROM THE GODS





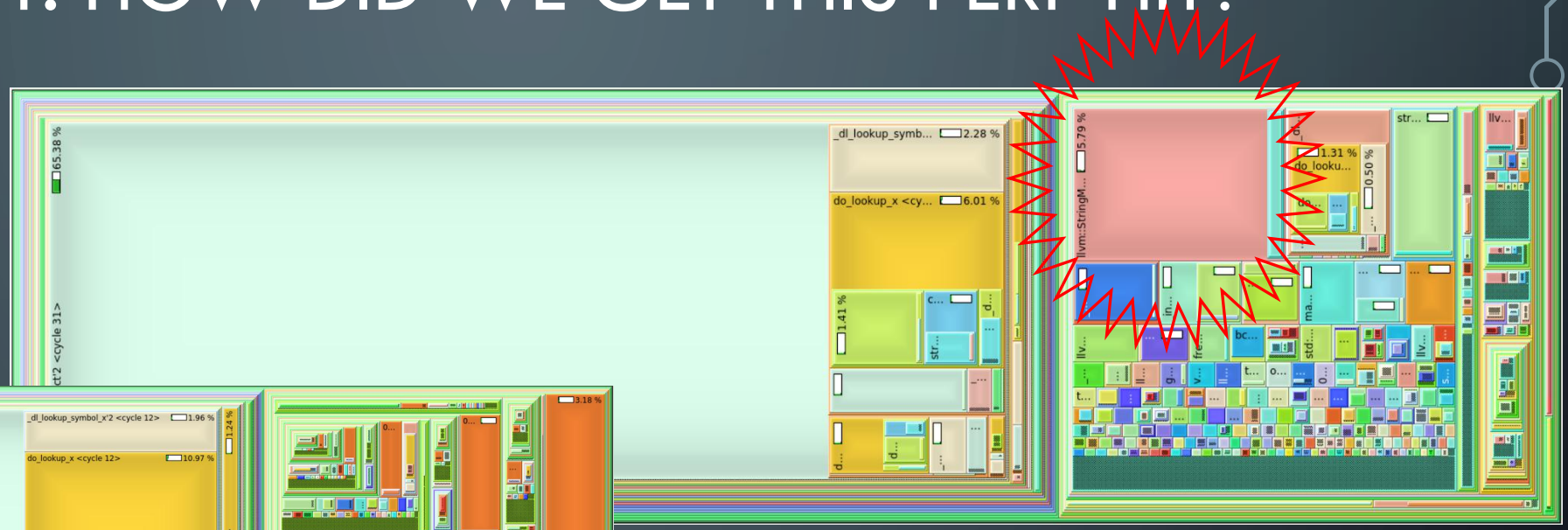


VALGRIND: EXAMPLE CONTINUED

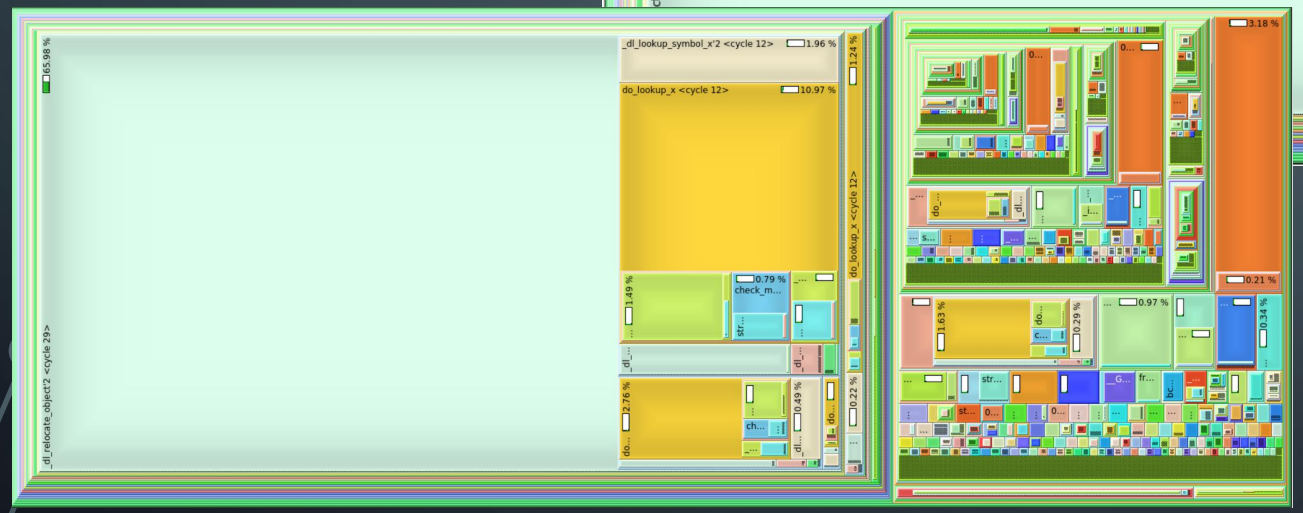
CASE STUDY DETECTING ALIGNMENT BUGS WITH VALGRIND

- For processing Valgrind output files I use as standard ubuntu image with
- The software I run is called Kcachegrind it has dependencies on the kde sdk and graphviz
  - apt install kcachegrind graphviz kdesdk
- The Valgrind tools I run are cachegrind & callgrind and it needs to run on an arm device for proper Switch comparisons.
- We used an AWS ARM64 machine to run valgrind like so
  - valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes --collect-jumps=yes ./LexTests

CASE STUDY: HOW DID WE GET THIS PERF HIT?



Function shuffle File Scope

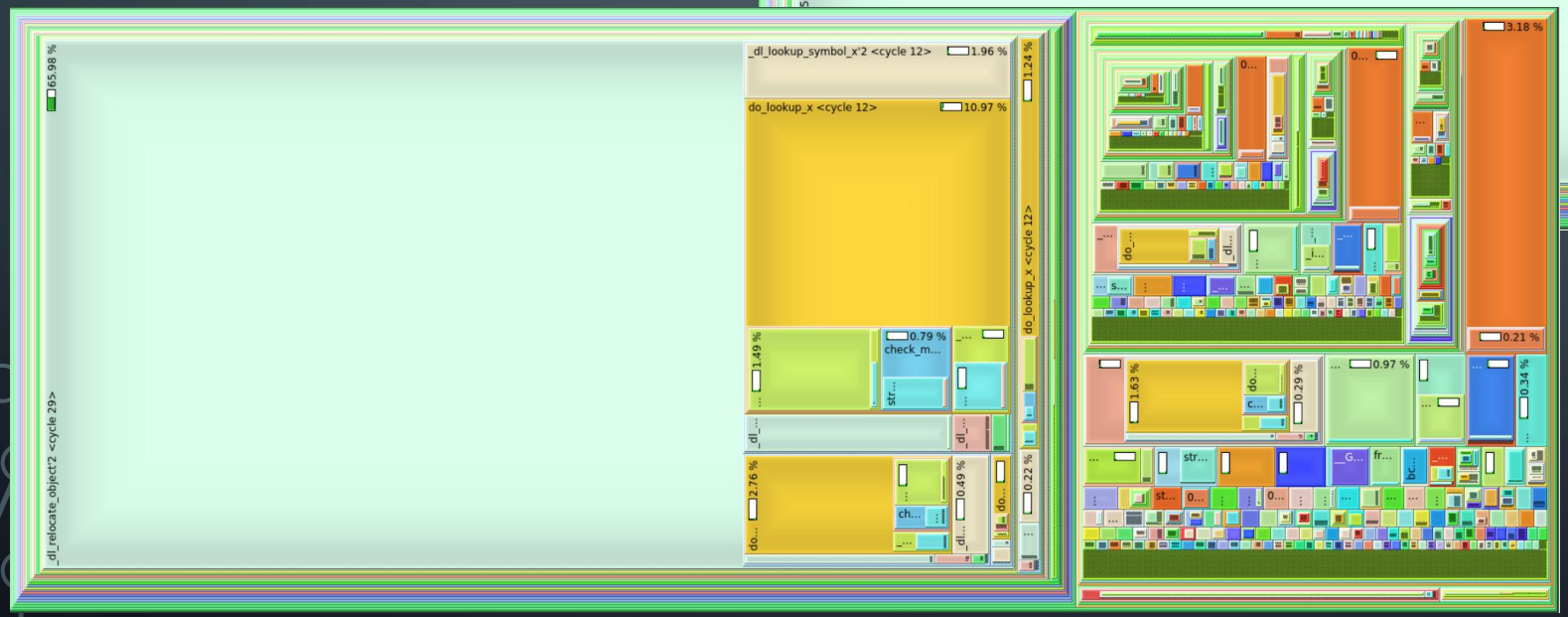


No Transformations

CASE STUDY: THE PERF HIT IS THE SAME ACROSS TRANSFORMATIONS?

- Why do these string related functions take longer after any Aegis2-post transformation?
  - ie function shuffle and basic block shuffle have the same perf problem

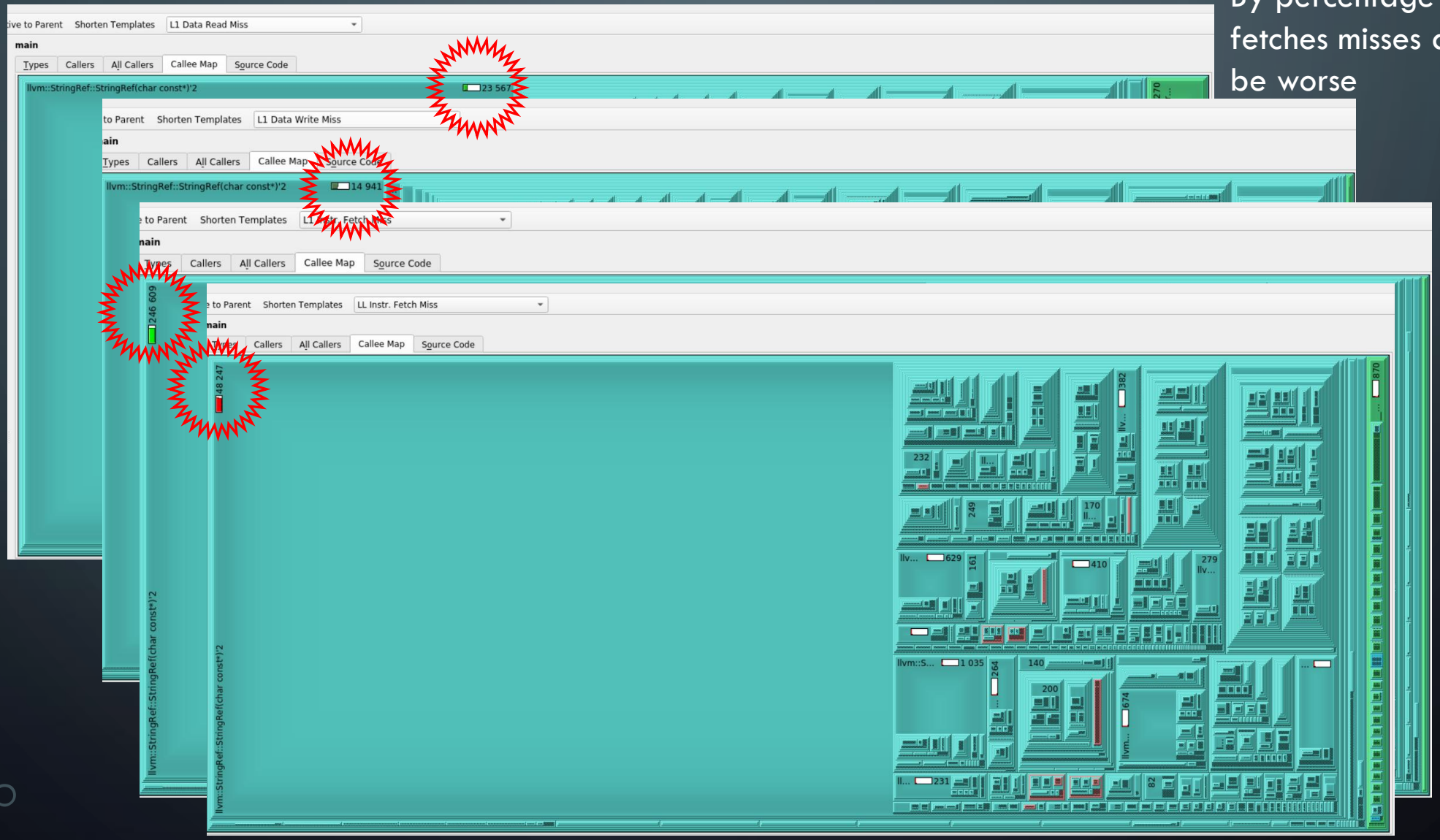
Basic Block shuffle



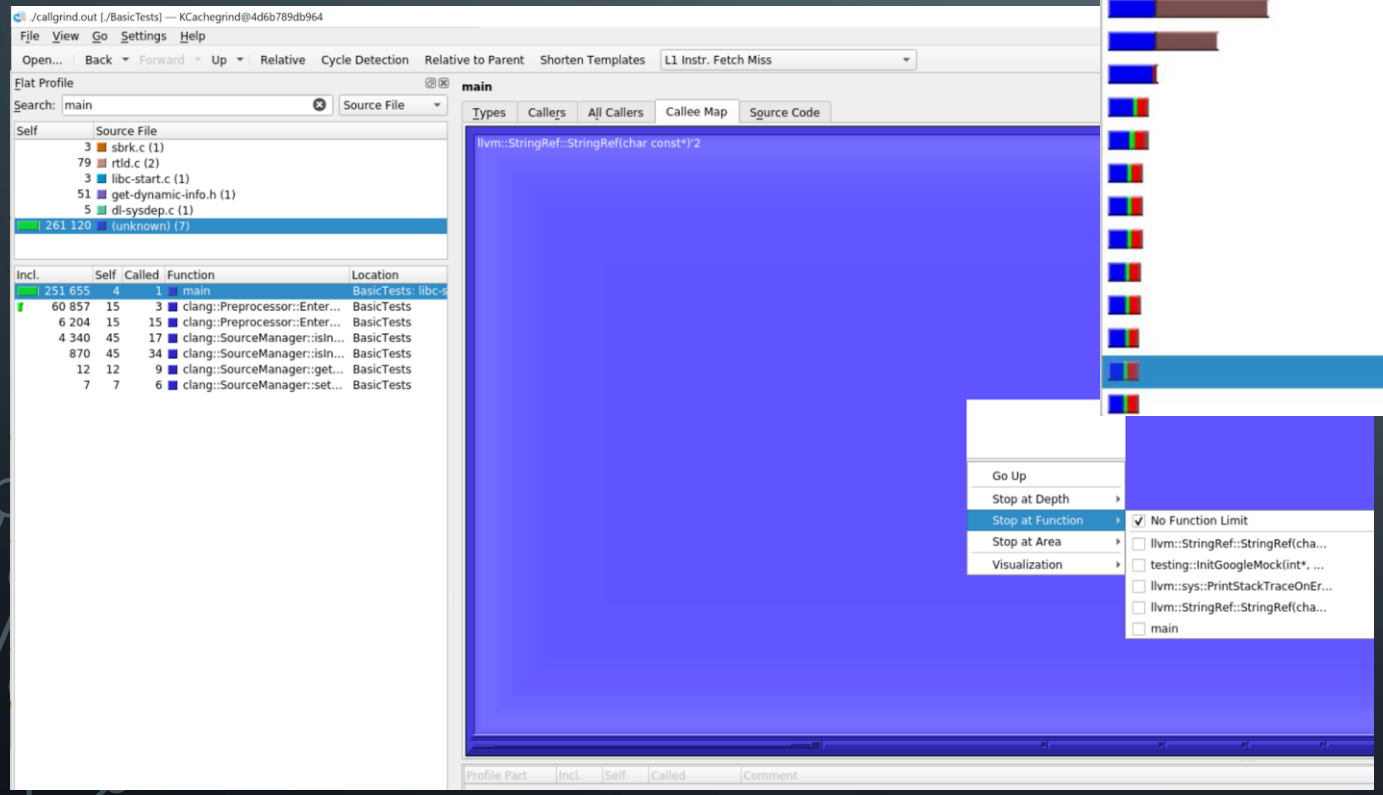
No Transformations

CASE STUDY: ZOOM IN ON MAIN

Clearly cache misses are an issue with strings now. By percentage Instructions fetches misses appear to be worse



CASE STUDY: MORE PERF DATA



Incl.	Self	Called	Function	Location
265 190 301	252 733	7 158	llvm::StringRef::StringRef(c...	BasicTests
184 839 801	1 909	14	testing::internal::UnitTestI...	BasicTests
182 081 383	4 297	58	testing::TestCase::Run()'2	BasicTests
156 782 913	62 106	1 611	AddKeyword(llvm::StringRe...	BasicTests
121 979 093	30 785	2 051	clang::targets::X86TargetIn...	BasicTests
119 691 999	47 276	3 642	bool std::__1::operator==<...	BasicTests
86 480 935	8 330	47	testing::TestInfo::Run()'2	BasicTests
37 290 445	29 968	567	llvm::StringMap<clang::Op...	BasicTests
30 208 750	2 895	417	0x0000000000745640'2	BasicTests
30 022 103	5 427	106	testing::Test::HasFatalFailu...	BasicTests
25 509 942	143	1	_start	BasicTests
25 509 799	131	1	0x00000000007455a0	BasicTests
25 507 228	825	1	__libc_csu_init	BasicTests:
23 497 584	4 784	62	llvm::StringRef::StringRef(c...	BasicTests
23 397 566	478	1	main	BasicTests:
23 256 566	1 207	39	std::__1::basic_string<char...	BasicTests
23 066 487	360	1	llvm::sys::PrintStackTraceO...	BasicTests
23 038 448	247	1	testing::InitGoogleMock(int...	BasicTests

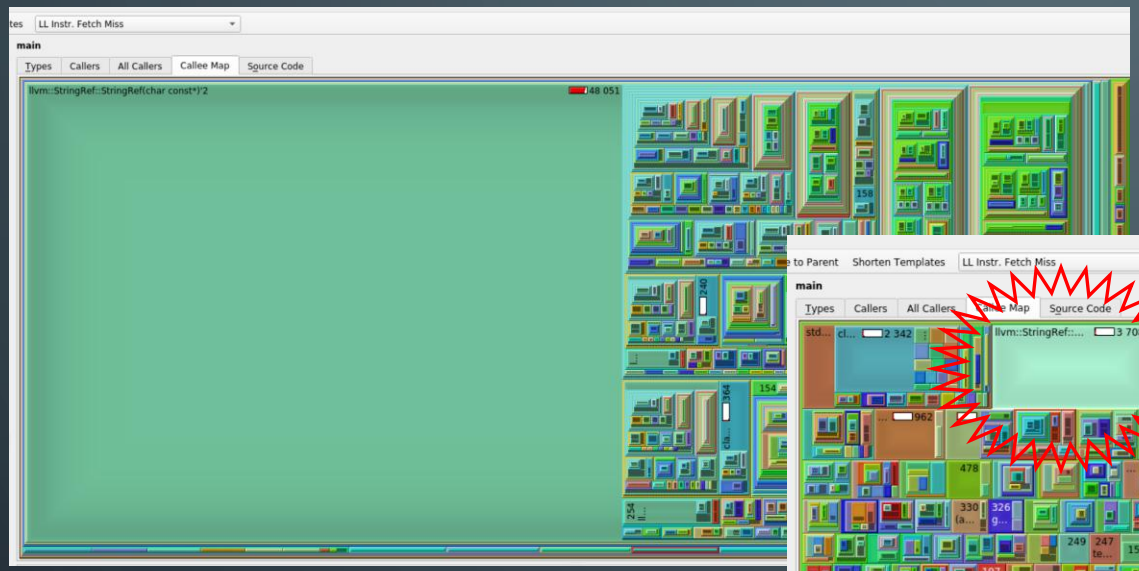
CASE STUDY: IDENTIFYING THE BUG

- We identified the perf bug the next step was to experiment with fixes
- Ida was used for this to diff emitted assembly.
  - We found two potential issues
    - NOPS between common jump patterns
    - Alignment added to functions that did not need them.

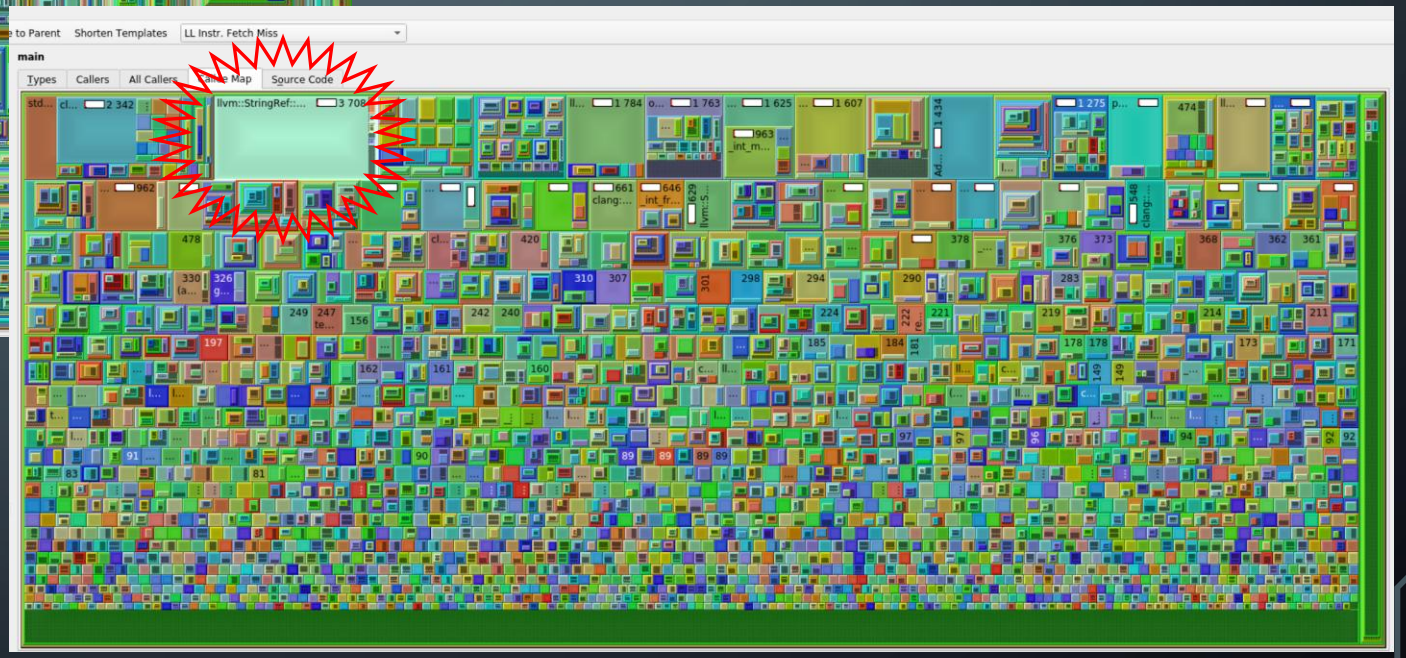
```
BL              _ZNKSt3__112basic_string
TBNZ            W0, #0, loc_199CC
NOP
B               loc_199DC
```

```
.text:00000000004016D0 ; End of function orderValue(llvm::Value const*,`anonymous namespace'::OrderMap &)
.text:00000000004016D0
.text:00000000004016D0 ; ---------------------------------------------------------------------------
.text:00000000004016D4                 ALIGN 0x20
.text:00000000004016E0
.text:00000000004016E0 ; =============== S U B R O U T I N E =======================================
```

CASE STUDY: ALIGNMENT CHANGE

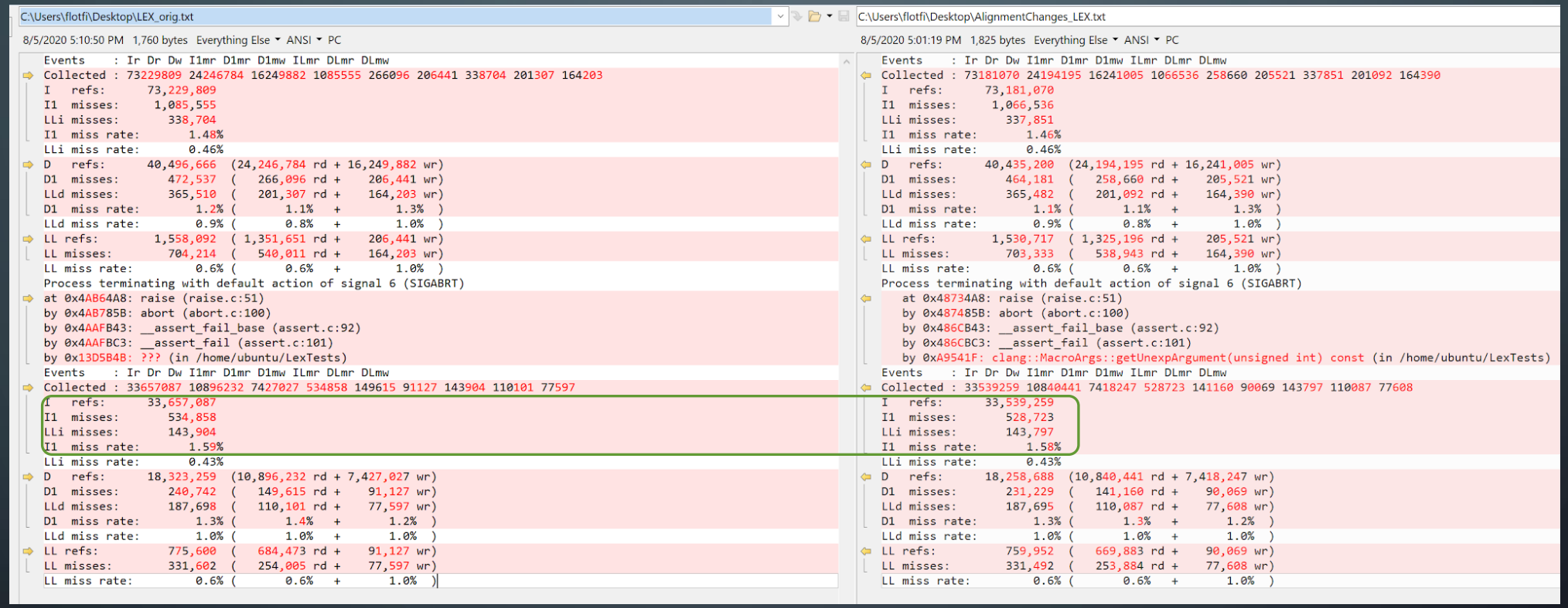


Before



After

CASE STUDY: HIGH LEVEL DIFF OF ALIGNMENT 1/2



C:\Users\flotfi\Desktop\LEX_orig.txt

8/5/2020 5:10:50 PM 1,760 bytes Everything Else ANSI PC

```
Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
Collected : 73229809 24246784 16249882 1085555 266096 206441 338704 201307 164203
I   refs:      73,229,809
I1  misses:     1,085,555
LLi misses:       338,704
I1  miss rate:       1.48%
LLi miss rate:       0.46%
D   refs:      40,496,666  (24,246,784 rd + 16,249,882 wr)
D1  misses:       472,537  (   266,096 rd +    206,441 wr)
LLd misses:       365,510  (   201,307 rd +    164,203 wr)
D1  miss rate:        1.2% (       1.1%   +        1.3%  )
LLd miss rate:        0.9% (       0.8%   +        1.0%  )
LL refs:        1,558,092  ( 1,351,651 rd +    206,441 wr)
LL misses:        704,214  (   540,011 rd +    164,203 wr)
LL miss rate:         0.6% (       0.6%   +        1.0%  )
Process terminating with default action of signal 6 (SIGABRT)
at 0x4AB64A8: raise (raise.c:51)
by 0x4AB785B: abort (abort.c:100)
by 0x4AAFB43: __assert_fail_base (assert.c:92)
by 0x4AAFBC3: __assert_fail (assert.c:101)
by 0x13D5B4B: ??? (in /home/ubuntu/LexTests)
Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
Collected : 33657087 10896232 7427027 534858 149615 91127 143904 110101 77597
I   refs:      33,657,087
I1  misses:       534,858
LLi misses:       143,904
I1  miss rate:       1.59%
LLi miss rate:       0.43%
D   refs:      18,323,259  (10,896,232 rd + 7,427,027 wr)
D1  misses:       240,742  (   149,615 rd +    91,127 wr)
LLd misses:       187,698  (   110,101 rd +    77,597 wr)
D1  miss rate:        1.3% (       1.4%   +        1.2%  )
LLd miss rate:        1.0% (       1.0%   +        1.0%  )
LL refs:          775,600  (   684,473 rd +    91,127 wr)
LL misses:        331,602  (   254,005 rd +    77,597 wr)
LL miss rate:         0.6% (       0.6%   +        1.0%  )
```

C:\Users\flotfi\Desktop\AlignmentChanges_LEX.txt

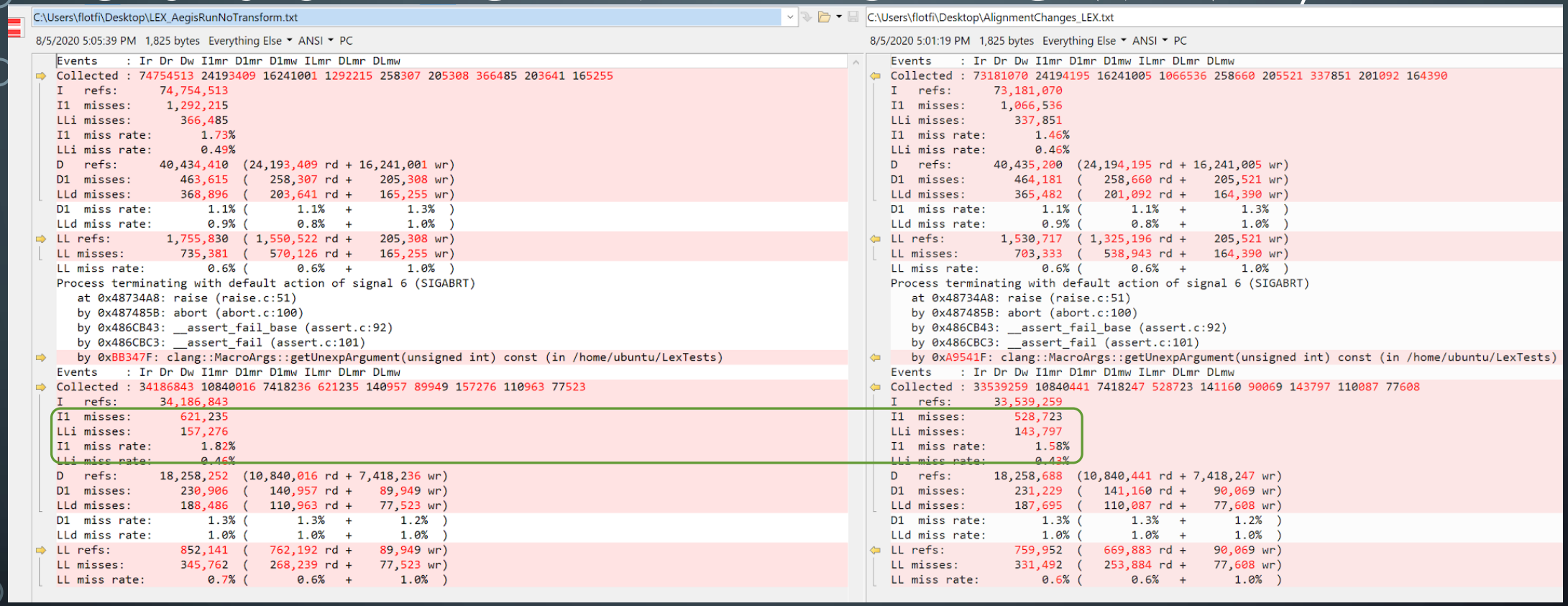
8/5/2020 5:01:19 PM 1,825 bytes Everything Else ANSI PC

```
Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
Collected : 73181070 24194195 16241005 1066536 258660 205521 337851 201092 164390
I   refs:      73,181,070
I1  misses:     1,066,536
LLi misses:       337,851
I1  miss rate:       1.46%
LLi miss rate:       0.46%
D   refs:      40,435,200  (24,194,195 rd + 16,241,005 wr)
D1  misses:       464,181  (   258,660 rd +    205,521 wr)
LLd misses:       365,482  (   201,092 rd +    164,390 wr)
D1  miss rate:        1.1% (       1.1%   +        1.3%  )
LLd miss rate:        0.9% (       0.8%   +        1.0%  )
LL refs:        1,530,717  ( 1,325,196 rd +    205,521 wr)
LL misses:        703,333  (   538,943 rd +    164,390 wr)
LL miss rate:         0.6% (       0.6%   +        1.0%  )
Process terminating with default action of signal 6 (SIGABRT)
   at 0x48734A8: raise (raise.c:51)
   by 0x487485B: abort (abort.c:100)
   by 0x486CB43: __assert_fail_base (assert.c:92)
   by 0x486CBC3: __assert_fail (assert.c:101)
   by 0xA9541F: clang::MacroArgs::getUnexpArgument(unsigned int) const (in /home/ubuntu/LexTests)
Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
Collected : 33539259 10840441 7418247 528723 141160 90069 143797 110087 77608
I   refs:      33,539,259
I1  misses:       528,723
LLi misses:       143,797
I1  miss rate:       1.58%
LLi miss rate:       0.43%
D   refs:      18,258,688  (10,840,441 rd + 7,418,247 wr)
D1  misses:       231,229  (   141,160 rd +    90,069 wr)
LLd misses:       187,695  (   110,087 rd +    77,608 wr)
D1  miss rate:        1.3% (       1.3%   +        1.2%  )
LLd miss rate:        1.0% (       1.0%   +        1.0%  )
LL refs:          759,952  (   669,883 rd +    90,069 wr)
LL misses:        331,492  (   253,884 rd +    77,608 wr)
LL miss rate:         0.6% (       0.6%   +        1.0%  )
```

No Transformation

Transformation With Alignment fix

CASE STUDY: HIGH LEVEL DIFF OF ALIGNMENT 2/2



C:\Users\flotfi\Desktop\LEX_AegisRunNoTransform.txt

8/5/2020 5:05:39 PM 1,825 bytes Everything Else ANSI PC

```
Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
Collected : 74754513 24193409 16241001 1292215 258307 205308 366485 203641 165255
I   refs:      74,754,513
I1  misses:     1,292,215
LLi misses:       366,485
I1  miss rate:       1.73%
LLi miss rate:       0.49%
D   refs:      40,434,410  (24,193,409 rd + 16,241,001 wr)
D1  misses:       463,615  (   258,307 rd +    205,308 wr)
LLd misses:       368,896  (   203,641 rd +    165,255 wr)
D1  miss rate:        1.1% (       1.1%   +        1.3%  )
LLd miss rate:        0.9% (       0.8%   +        1.0%  )
LL refs:        1,755,830  ( 1,550,522 rd +    205,308 wr)
LL misses:        735,381  (   570,126 rd +    165,255 wr)
LL miss rate:         0.6% (       0.6%   +        1.0%  )
Process terminating with default action of signal 6 (SIGABRT)
   at 0x48734A8: raise (raise.c:51)
   by 0x487485B: abort (abort.c:100)
   by 0x486CB43: __assert_fail_base (assert.c:92)
   by 0x486CBC3: __assert_fail (assert.c:101)
   by 0xBB347F: clang::MacroArgs::getUnexpArgument(unsigned int) const (in /home/ubuntu/LexTests)
Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
Collected : 34186843 10840016 7418236 621235 140957 89949 157276 110963 77523
I   refs:      34,186,843
I1  misses:       621,235
LLi misses:       157,276
I1  miss rate:       1.82%
LLi miss rate:       0.46%
D   refs:      18,258,252  (10,840,016 rd + 7,418,236 wr)
D1  misses:       230,906  (   140,957 rd +    89,949 wr)
LLd misses:       188,486  (   110,963 rd +    77,523 wr)
D1  miss rate:        1.3% (       1.3%   +        1.2%  )
LLd miss rate:        1.0% (       1.0%   +        1.0%  )
LL refs:          852,141  (   762,192 rd +    89,949 wr)
LL misses:        345,762  (   268,239 rd +    77,523 wr)
LL miss rate:         0.7% (       0.6%   +        1.0%  )
```

C:\Users\flotfi\Desktop\AlignmentChanges_LEX.txt

8/5/2020 5:01:19 PM 1,825 bytes Everything Else ANSI PC

```
Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
Collected : 73181070 24194195 16241005 1066536 258660 205521 337851 201092 164390
I   refs:      73,181,070
I1  misses:     1,066,536
LLi misses:       337,851
I1  miss rate:       1.46%
LLi miss rate:       0.46%
D   refs:      40,435,200  (24,194,195 rd + 16,241,005 wr)
D1  misses:       464,181  (   258,660 rd +    205,521 wr)
LLd misses:       365,482  (   201,092 rd +    164,390 wr)
D1  miss rate:        1.1% (       1.1%   +        1.3%  )
LLd miss rate:        0.9% (       0.8%   +        1.0%  )
LL refs:        1,530,717  ( 1,325,196 rd +    205,521 wr)
LL misses:        703,333  (   538,943 rd +    164,390 wr)
LL miss rate:         0.6% (       0.6%   +        1.0%  )
Process terminating with default action of signal 6 (SIGABRT)
   at 0x48734A8: raise (raise.c:51)
   by 0x487485B: abort (abort.c:100)
   by 0x486CB43: __assert_fail_base (assert.c:92)
   by 0x486CBC3: __assert_fail (assert.c:101)
   by 0xA9541F: clang::MacroArgs::getUnexpArgument(unsigned int) const (in /home/ubuntu/LexTests)
Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
Collected : 33539259 10840441 7418247 528723 141160 90069 143797 110087 77608
I   refs:      33,539,259
I1  misses:       528,723
LLi misses:       143,797
I1  miss rate:       1.58%
LLi miss rate:       0.43%
D   refs:      18,258,688  (10,840,441 rd + 7,418,247 wr)
D1  misses:       231,229  (   141,160 rd +    90,069 wr)
LLd misses:       187,695  (   110,087 rd +    77,608 wr)
D1  miss rate:        1.3% (       1.3%   +        1.2%  )
LLd miss rate:        1.0% (       1.0%   +        1.0%  )
LL refs:          759,952  (   669,883 rd +    90,069 wr)
LL misses:        331,492  (   253,884 rd +    77,608 wr)
LL miss rate:         0.6% (       0.6%   +        1.0%  )
```

Transformation

Transformation With Alignment fix

256mb Binary

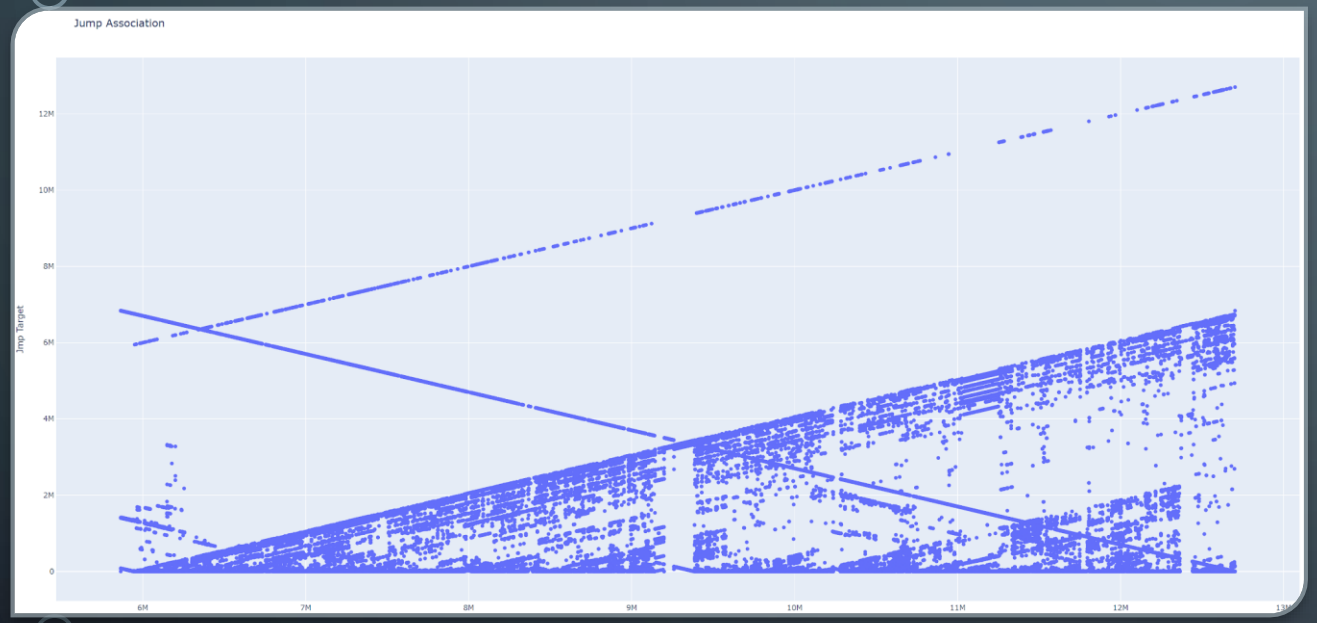
When we compare against our patch to pre patch we reduce cache misses by .24 which is about 200k fewer misses

New_Value = 1.58; Original_Value = 1.82; Percentage_Increase = Math.abs(((New_Value - Original_Value) / Original_Value)) * 100 = 13% fewer misses

DATA VIZ TO FIND PERF ISSUES

- Should be clear from the case study we can use Data Viz to find perf bugs
- We can turn Ida, Angr, pin, qemu into Data collecting tools to visualize perf problems
  - For my case I started with Ida.
  - Wanted to know why Function shuffling was more expensive.
  - So I graphed the call distances

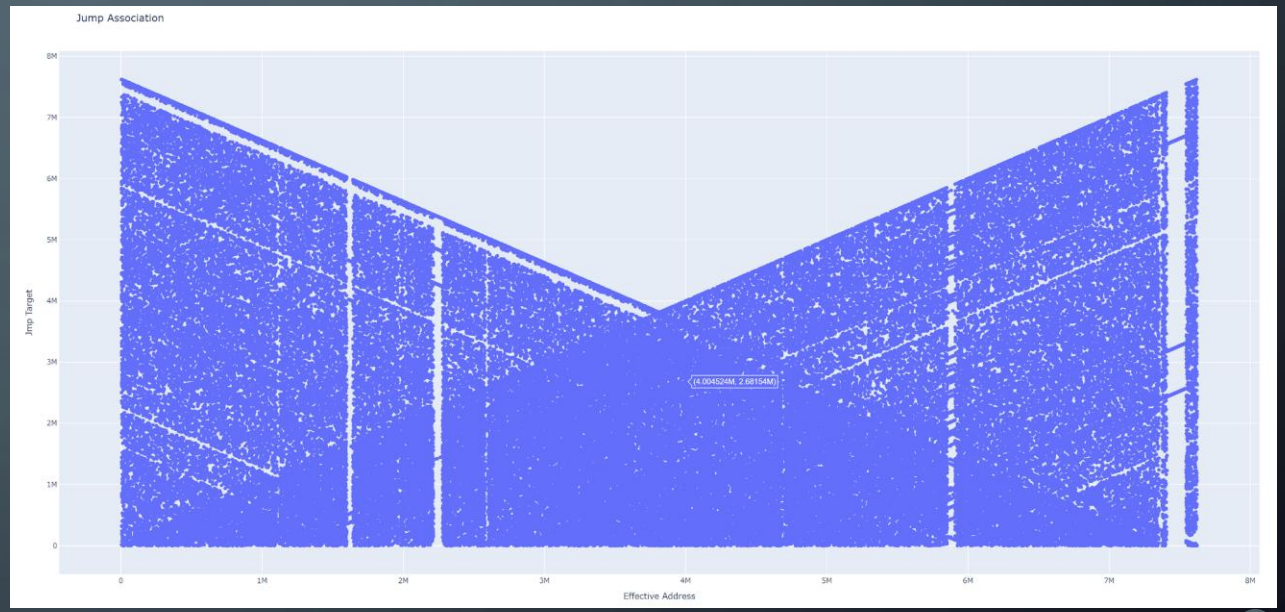
DATA VIZ: FUNCTION CALL DISTANCE



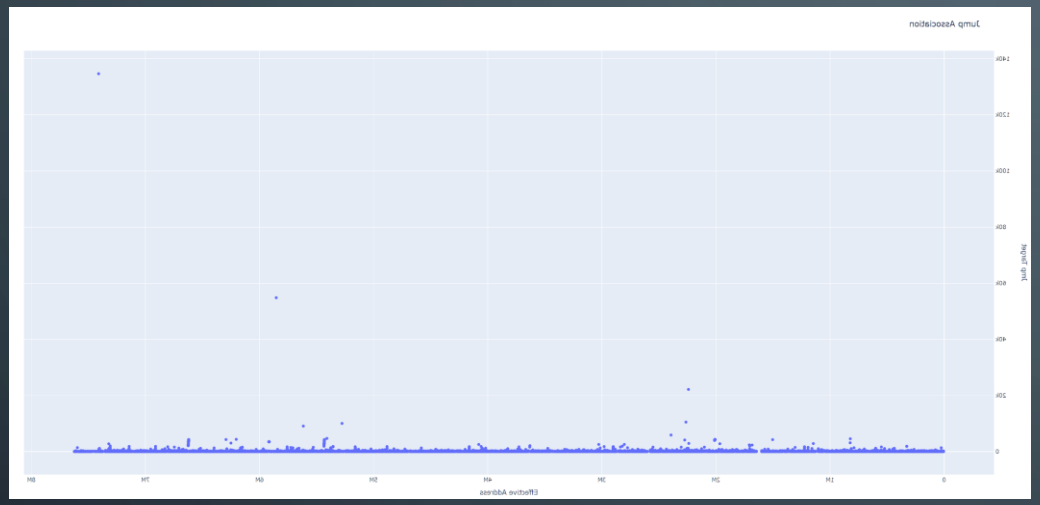
- Here we have Clang's BasicTest. A 10mb Binary with no function shuffling
  - Y-axis is Jump Target.
  - X-axis is Effective Address
- One thing is clear in the data, there is an ordering, functions are either localized or are far away, very few in between.

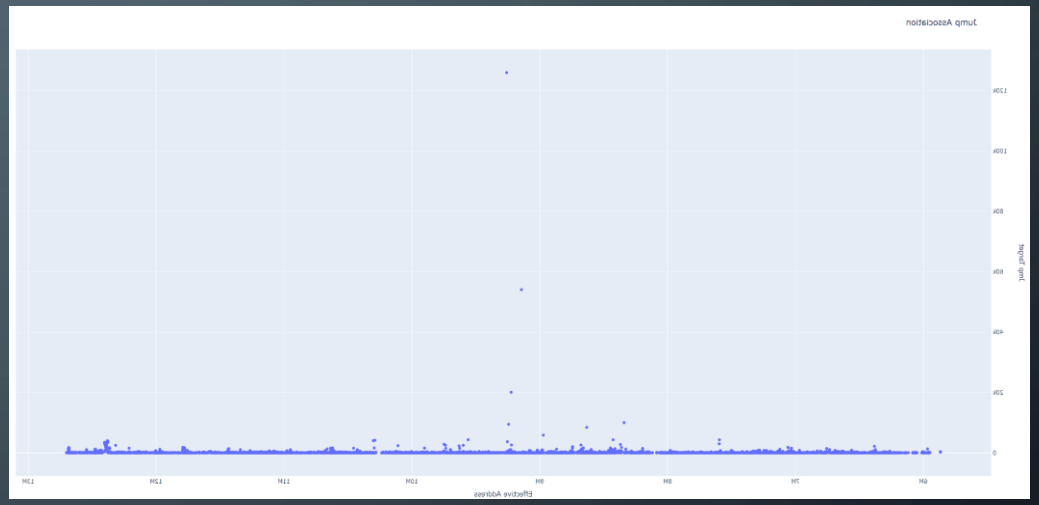
DATA VIZ: FUNCTION CALL DISTANCE AFTER SHUFFLING

- Here we have a 10mb BasicTest Binary with function shuffling
- One thing is clear in the data, there is no ordering we have a uniform distribution, things are random and we have broken the ordering caused by LTO.
- This would be fine if the perf was negligible, but its not.



DATA VIZ: SANITY CHECK (BB JMP DST THE SAME)





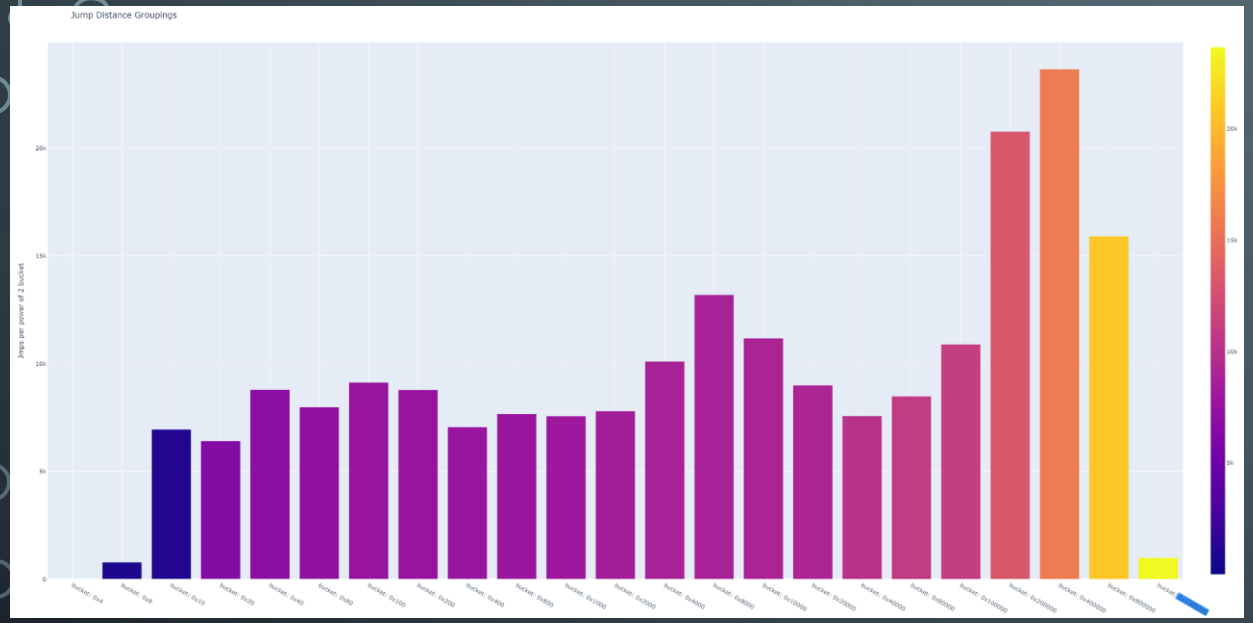
- If we just map the branches no discernible changes between the original binary (left) and the function shuffled binary (right). Only change would be function shuffle changes placement of functions starting effective address.
- TLDR the perf hit we see is isolated to call distance

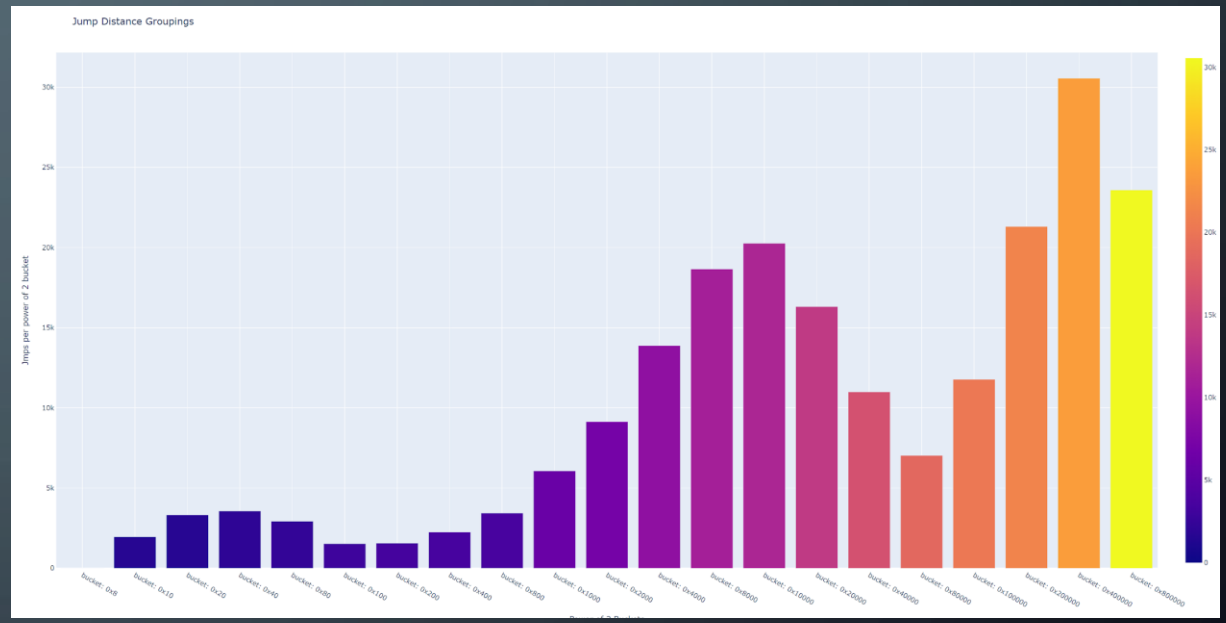
HOW DOES FUNCTION LAYOUT RANDOMIZATION HURT PERFORMANCE?

- Given the DataVis and Alignment Layout Bug case study you all can answer this.

- CACHE MISSES and PAGE FAULTS

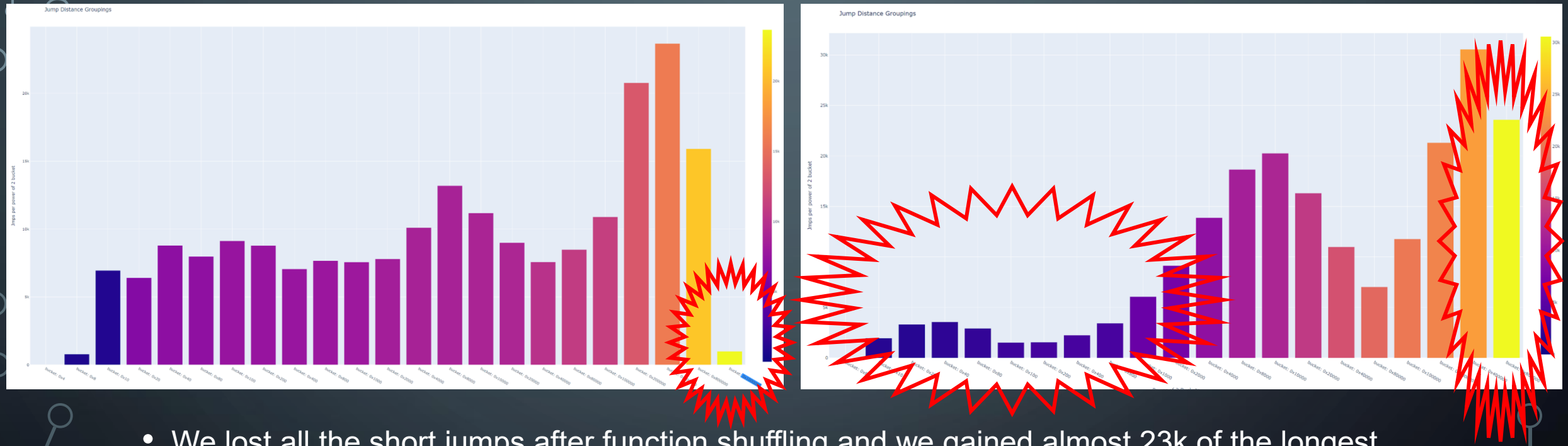
IDEA: CAN WE GROUP THE JMP DISTANCES?





- Comparisons of the scatter plot makes it look like entropy ate the universe.
- Best to give up now? NO.
- Grouping the Data will give us a better idea of what functions moved.
- Left: No transformation, right: filescope function shuffle

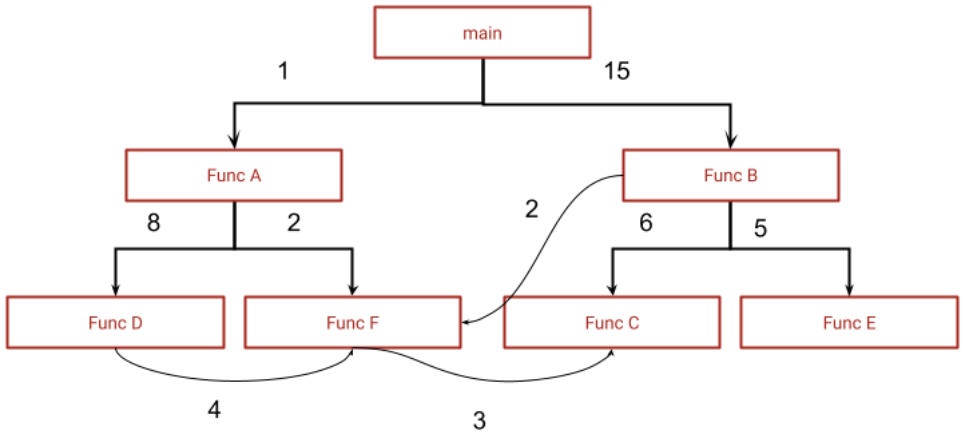
IDEA: CAN WE GROUP THE JMP DISTANCES?



- We lost all the short jumps after function shuffling and we gained almost 23k of the longest jumps

WHAT IS "APARTMENT-LEVEL RANDOMIZATION"?

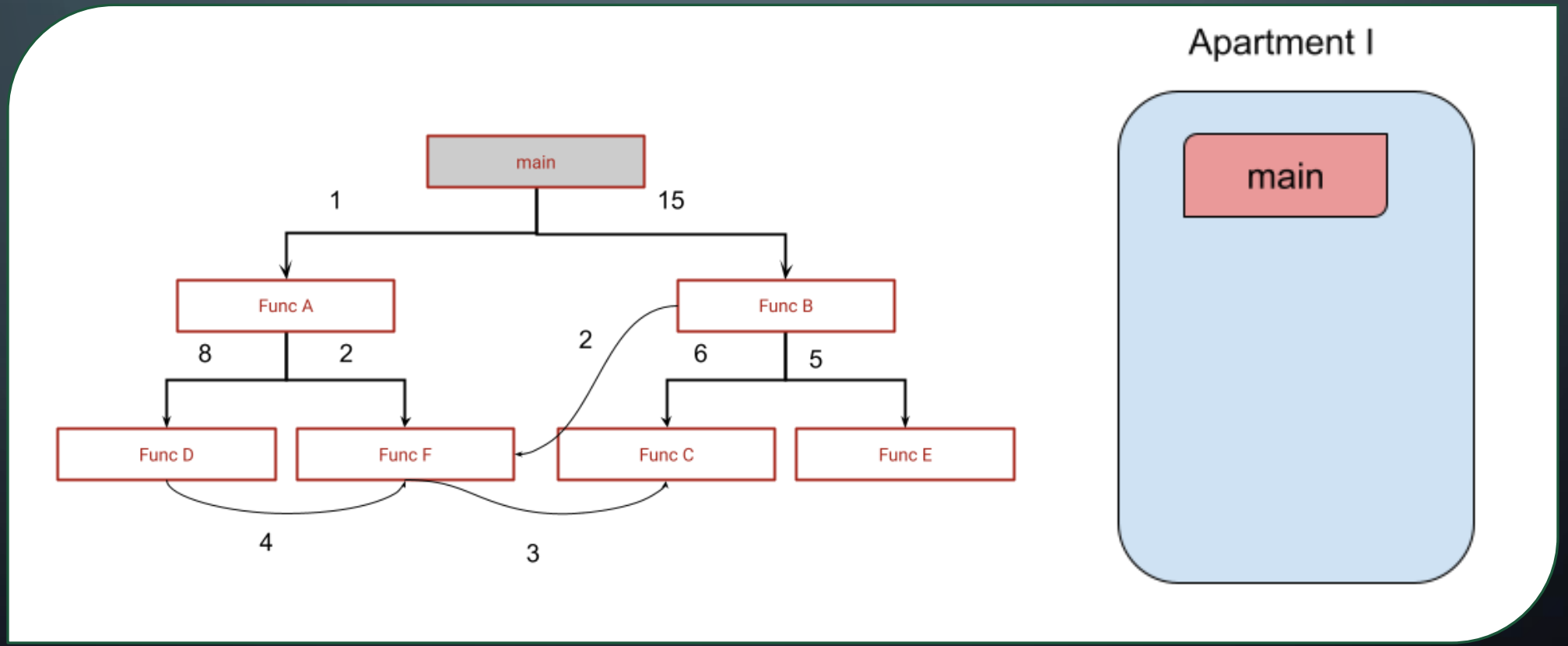
- It is a term I'm coining so you won't find any literature on it.
- Essentially, we want to use the original call distances as a heuristic, so we don't break LTO as badly.
- The call distance will be a heuristic for a K nearest neighbor like algorithm for bucketing functions, these buckets are "Apartments".
- Instead of shuffling functions we shuffle apartments. We can also shuffle within an apartment with minimal impact.



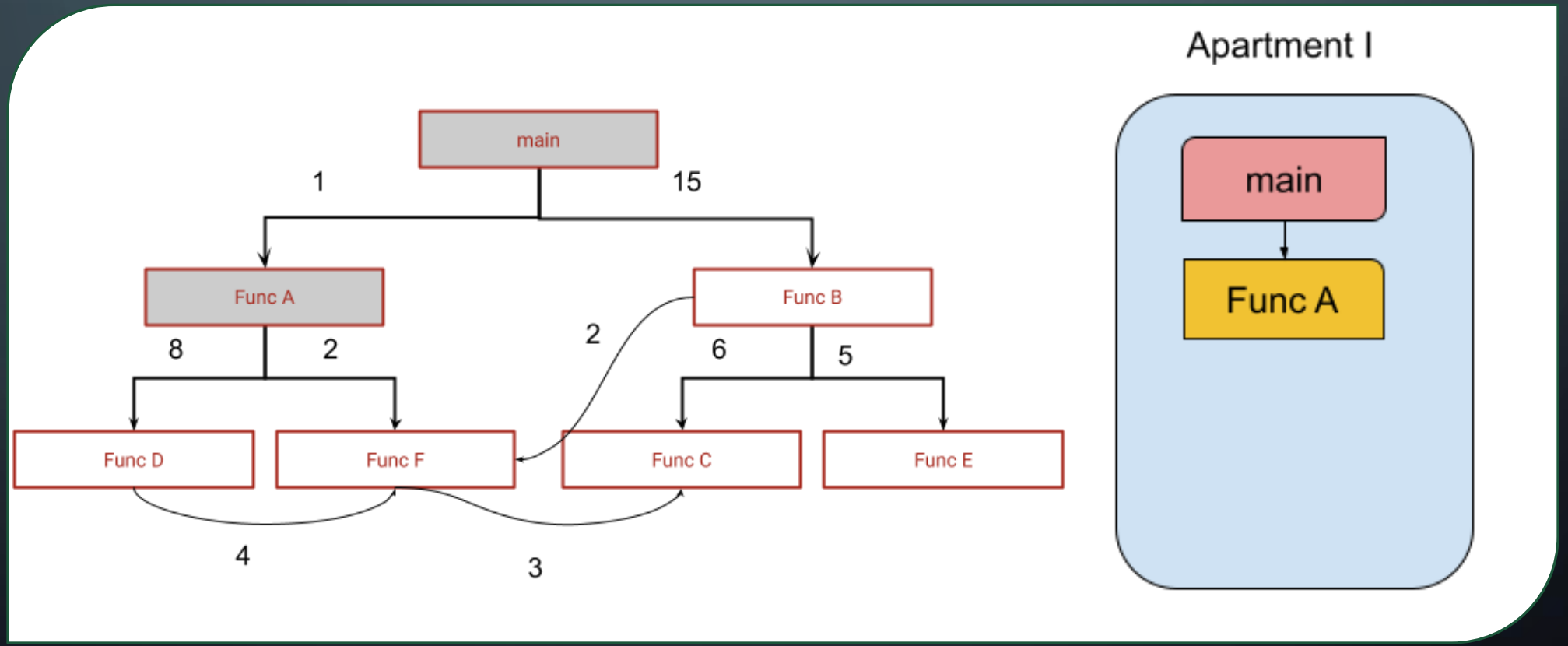
EXAMPLE: APARTMENT LEVEL RANDOMIZATION

Let K = 6. Perform any graph search (ex. BFS/DFS) on the call graph.
If distance is less than K add to existing apartment. Else create a new apartment.
To do this correctly we need a map of functions to a sorted list of callers by jmp distance.

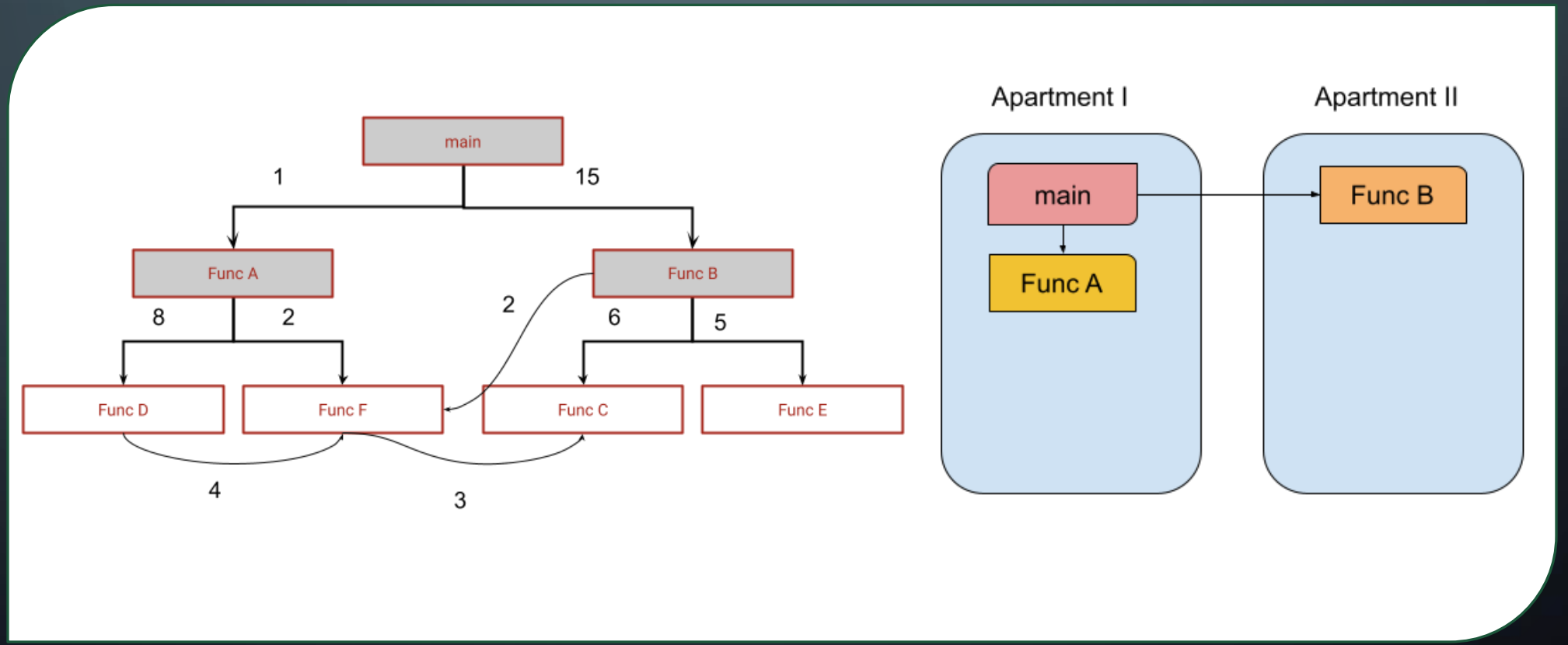
EXAMPLE: APARTMENT LEVEL RANDOMIZATION



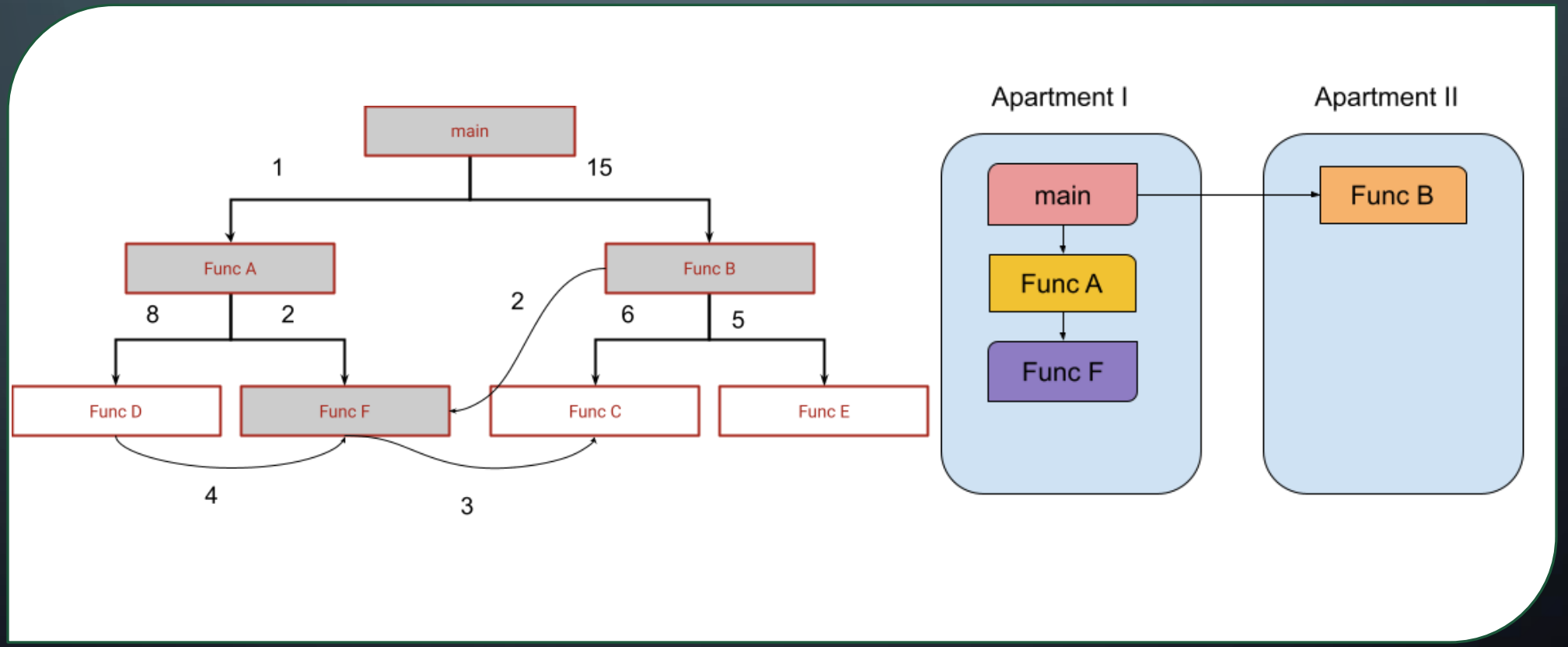
EXAMPLE: APARTMENT LEVEL RANDOMIZATION



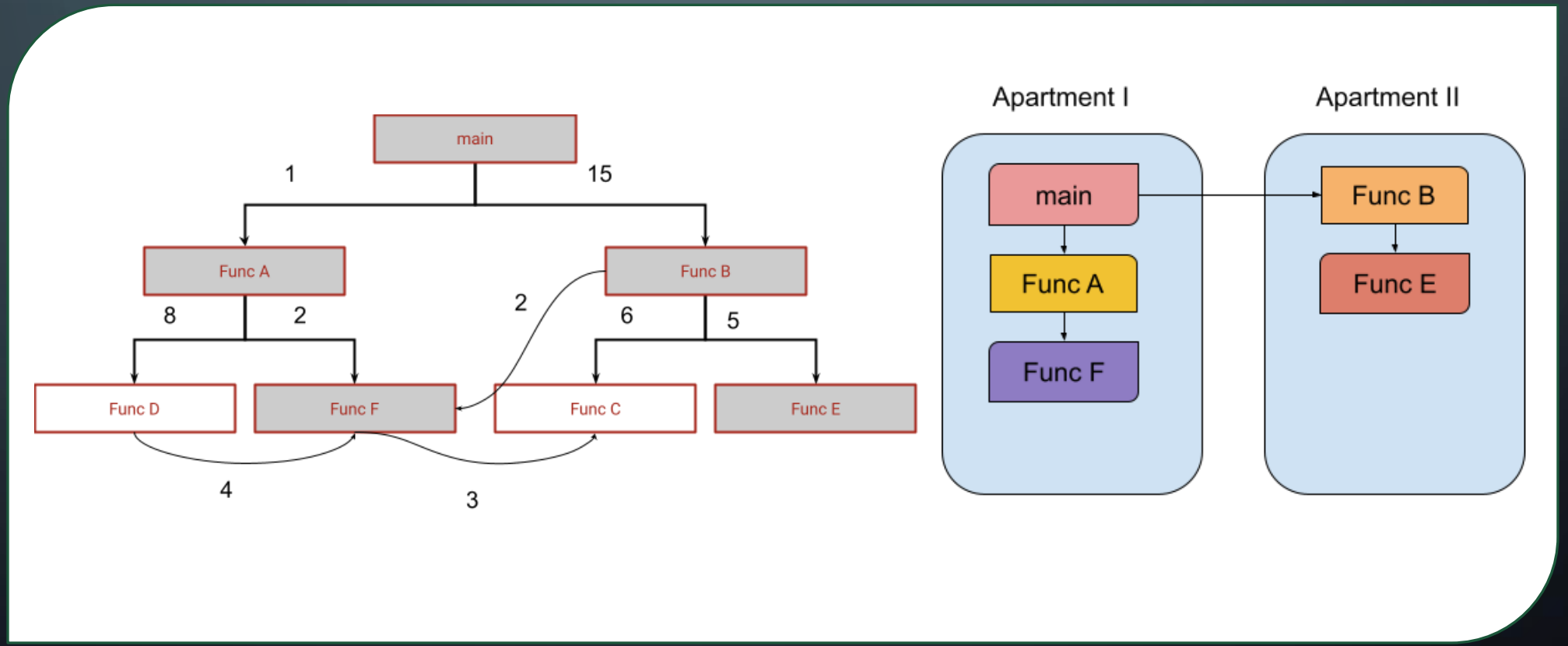
EXAMPLE: APARTMENT LEVEL RANDOMIZATION



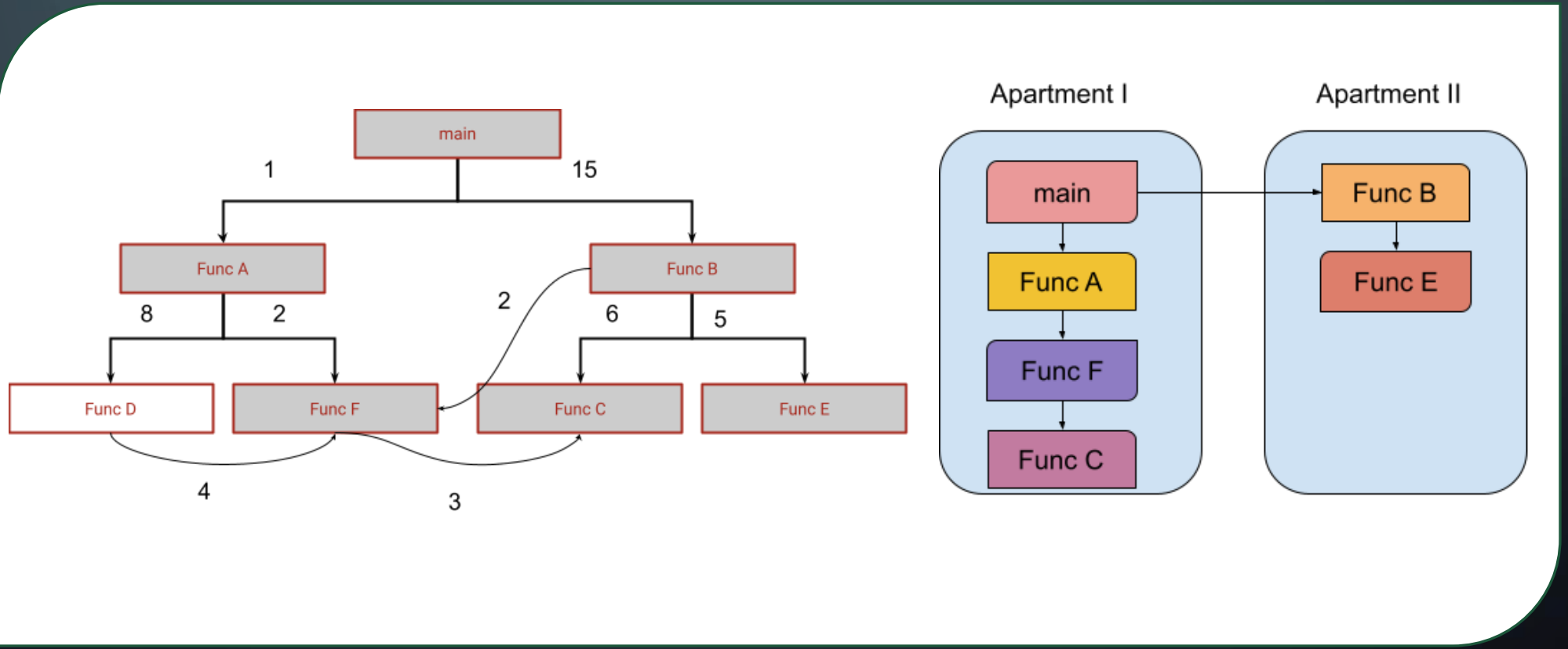
EXAMPLE: APARTMENT LEVEL RANDOMIZATION



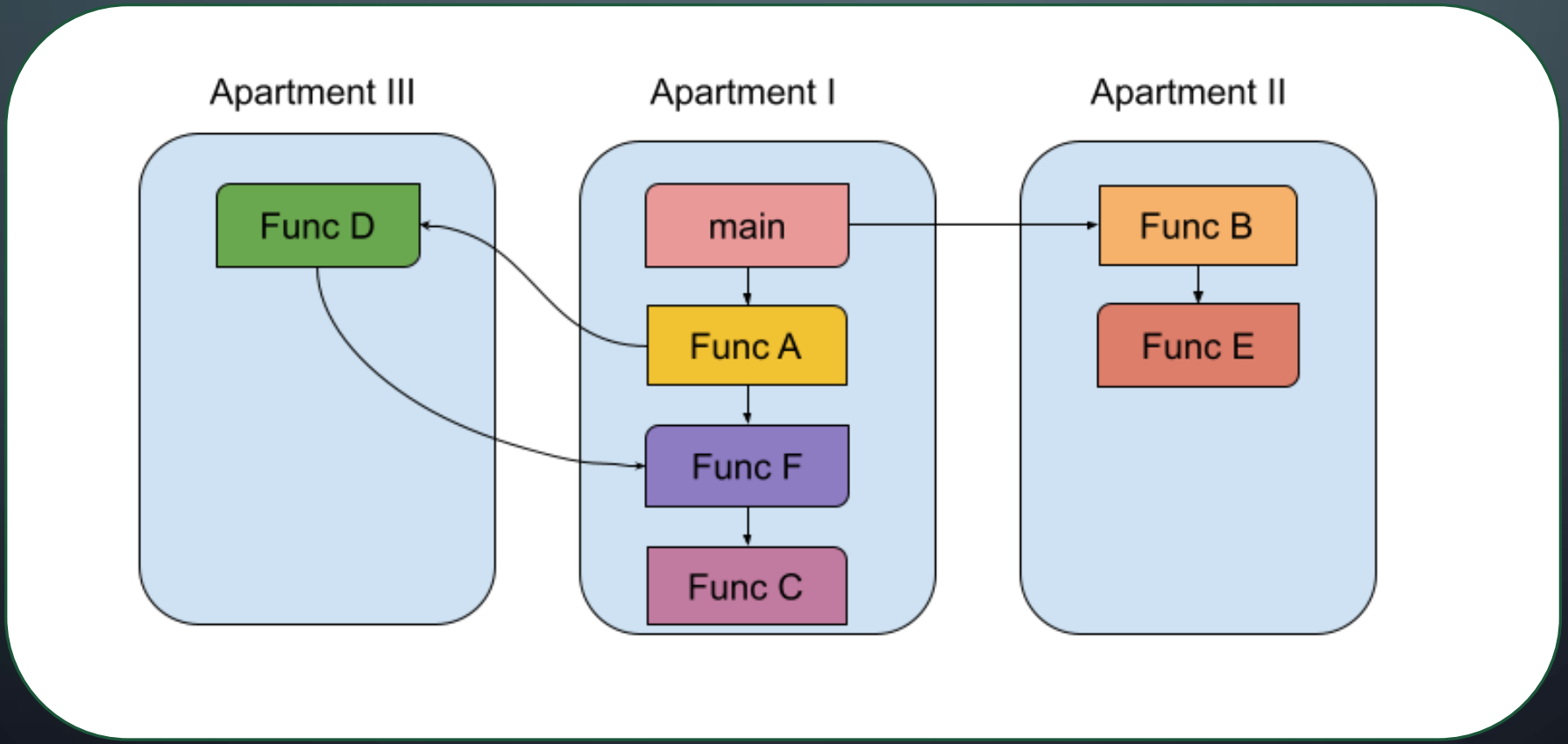
EXAMPLE: APARTMENT LEVEL RANDOMIZATION



EXAMPLE: APARTMENT LEVEL RANDOMIZATION



EXAMPLE: WALK CALL GRAPH & PUT THINGS IN APARTMENTS



- In this example we set k to 6. I.e. if the call distance is greater than 6 we don't add it to an existing apartment when we see it.
- The long jumps remain long and can be randomized.
- The short jumps remain grouped together

SO WHY DOES THIS APPROACH WORK AND BY HOW MUCH?

- We maintain Locality of References (specifically spatial locality)
- Long calls remain long or get longer
  - making them longer usually doesn't hurt perf because a long call was already going to cause a page fault or a cache miss.
- Short calls remain short
- Perf restored to 99% of untransformed binaries on AWS Graviton (Gen1).

WAIT I THOUGHT YOU SAID MOST PERF RESTORED BUT THIS IS ALMOST ALL?

- Yes! With the right heuristics you can get almost all the perf back.
- If we are willing to tolerate a bigger perf hit we can also randomize the function layout within an apartment.
- This get you to within 85% to 90% of original perf of an untransformed binary.

WAIT I THOUGHT THIS WAS ABOUT THE SWITCH?

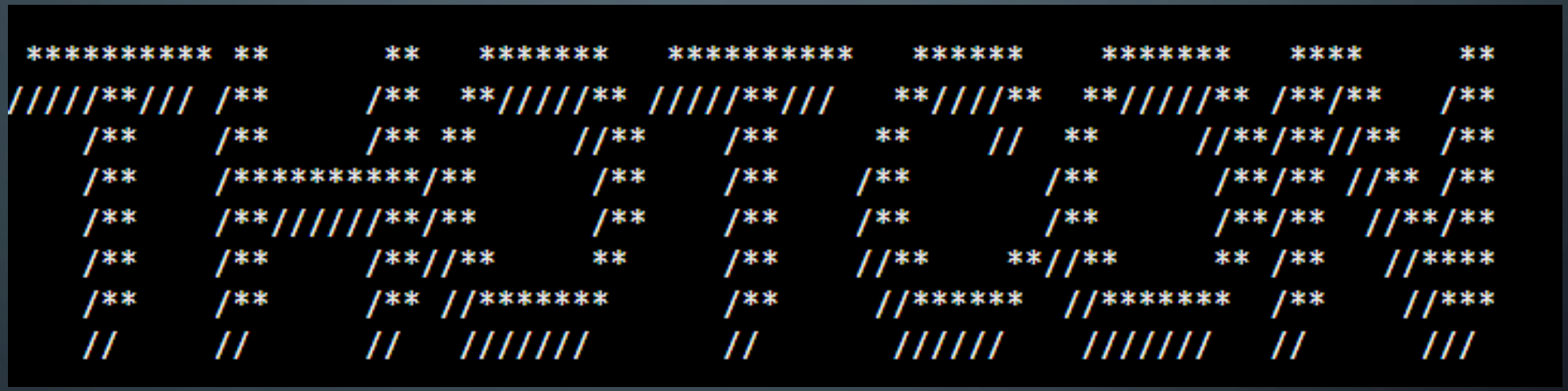
- Experiments and testing were done on Graviton (Gen1) CPU and an RPI4 both with stock ARM Cortex A72s.
- The Switch uses a Nvidia Tegra X1 with a stock ARM Cortex-A57.
  - With tools like Cachegrind we can simulate correct cache size differences.
  - Stock ARM CPUs have similar performance characteristics.
- Its easier to spin up a VM and test with OSS unit tests and benchmarks than to get a signed game binary on to a switch.

Product	AWS VM (A1 EC2)	Nintendo Switch	RPI4
CPU Vendor	Amazon	Nvidia	Broadom
CPU Name	Graviton	Tegra X1	BCM2711
ISA	ARMv8-A	ARMv8-A	ARMv8-A
ARM Version	Cortex A72	Coretx A57	Cortex A72
Clock Rate	2.3 GHz	1020–1785 MHz	1.8 Gz
L1 Cache	80 KB per core	80 KB per core	80 KB per core
L2 Cache	8 MB	2 MB	1 MB

FUTURE WORK (IE WORK I WISH THEY LET ME DO)

- Right now heuristics for apartment cut offs is set to a good enough default that can be changed with a flag.
- Finding the right heuristic per game would be a manual process:
  - Use the apartment jump distance less than or equal to flag & Recompile the binary
  - Run in the binary through cachegrind track perf hit use a smaller jmp distance if perf hit is larger than 5%
  - The same jmp distance won't have the same performance characteristics across games
- This an optimization problem.
  - We could use ci to find the right heuristic per game.

THANK YOU




QUESTIONS?

CREDITS

- https://github.com/HearthSim/hsdata/blob/b4a35db050f92c681f2932d002a4768b7f69e10a/Strings/thTH/CREDITS_2020.txt#L651