Adopting Razor for Post-deployment Software Debloating

Chenxiong Qian, Kevin Koo, Hong Hu,
Taesoo Kim, Wenke Lee

CREATING THE NEXT®

Functionalities Wanted

Functionalities that Come with Installation

Functionalities that Come with Installation

Well-investigated

Functionalities that Come with Installation

Well-investigated

Unknown

Functionalities that Come with Installation

Well-investigated

Unknown

Bloated Code

Bloated Code Increases an Attack Surface

- TLS heartbeat extension.
- Not used by most users.
- Enabled by default.

Software Debloating

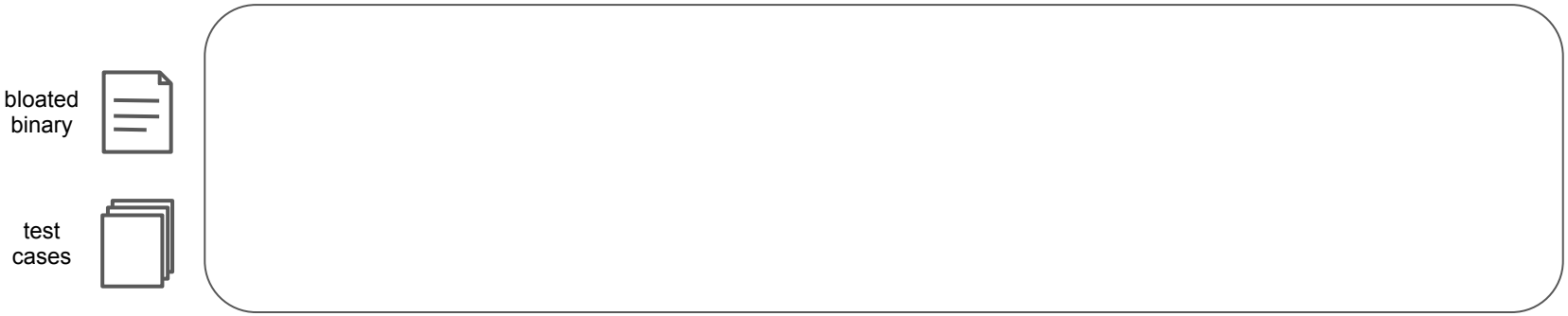
- Most existing software debloating systems have the following limitations:
  - Require source code.
    - Source code is not always accessible to users.
    - It’s challenging and time-consuming to recompile source code.
  - Assume test cases are complete.
    - This assumption mostly fails in real world.
    - Causes over-debloating problem.

Razor

➢ Performs code reduction for deployed **binaries**.

➢ Uses **heuristics** to infer related code for given test cases.

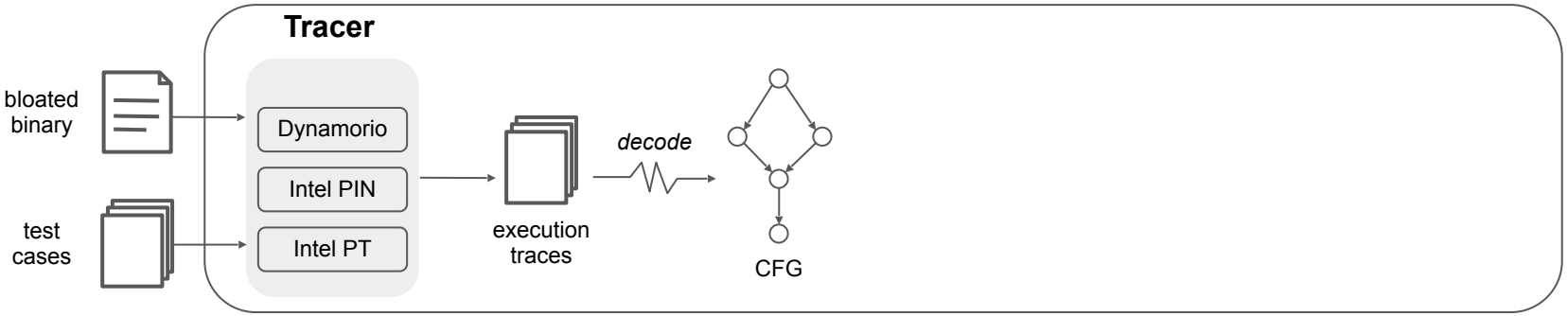
Overview



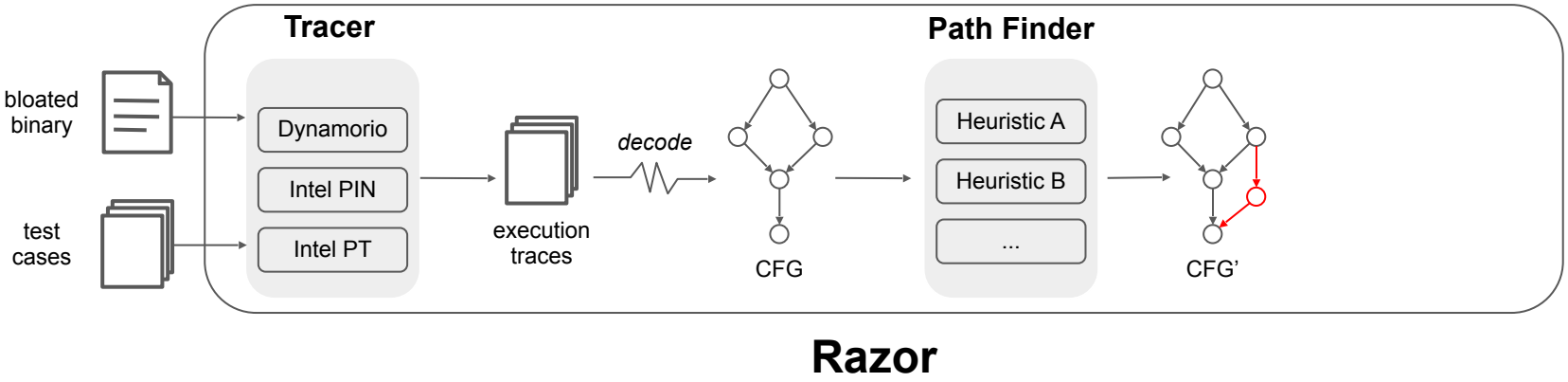
Overview


Tracer
bloated binary
test cases
Dynamorio
Intel PIN
Intel PT
Razor

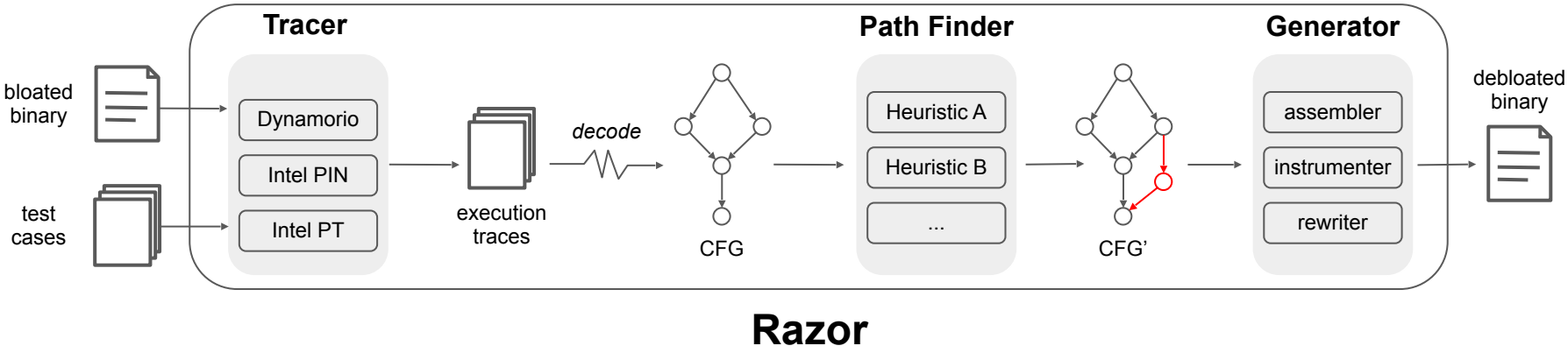
Overview


Tracer
bloated binary
test cases
Dynamorio
Intel PIN
Intel PT
execution traces
decode
CFG
Razor

Overview


Tracer
Path Finder
bloated binary
test cases
Dynamorio
Intel PIN
Intel PT
execution traces
decode
CFG
Heuristic A
Heuristic B
...
CFG'
Razor

Overview


bloated binary
test cases
Tracer
Dynamorio
Intel PIN
Intel PT
execution traces
decode
CFG
Path Finder
Heuristic A
Heuristic B
...
CFG'
Generator
assembler
instrumenter
rewriter
debloated binary
Razor

Tracer

- Multiple tracers
  - Software-based tracers (Dynamorio, Intel PIN)
    - Complete trace
    - Significant overhead
  - Hardware-based tracer (Intel PT)
    - Small overhead
    - Incomplete trace

Tracer

- Multiple tracers
  - Software-based tracers (Dynamorio, Intel PIN)
    - Complete trace
    - Significant overhead
  - Hardware-based tracer (Intel PT)
    - Small overhead
    - Incomplete trace

- The collected trace contains three parts:

Executed Blocks
[0x4005c0, 0x4005f2]
[0x400596, 0x4005ae]
...

Conditional Branches
[0x4004e3: true]
[0x4004ee: false]
[0x400614: true, false]
...

Indirect Calls/Jumps
[0x400677, 0x4005e6#18, 0x4005f6#6
...

Path Finder

- Four Heuristics
  - zCode (zero code)
  - zCall (zero call)
  - zLib (zero library call)
  - zFunc (zero functionality)

Heuristic **zCode**

- Find a path ***p*** not in the executed CFG, add ***p*** if all code on ***p*** is executed.

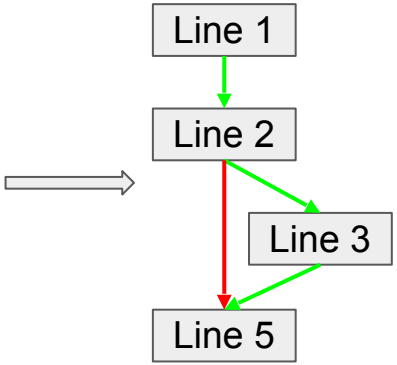
```
1 char *input = read_input();
2 if (everything_is_ok(input)) {
3     //do the job
4 }
5 return;
6
```

The traces only cover the “true” branch.

Heuristic **zCode**

- Find a path ***p*** not in the executed CFG, add ***p*** if all code on ***p*** is executed.

```
1 char *input = read_input();
2 if (everything_is_ok(input)) {
3     //do the job
4 }
5 return;
6
```



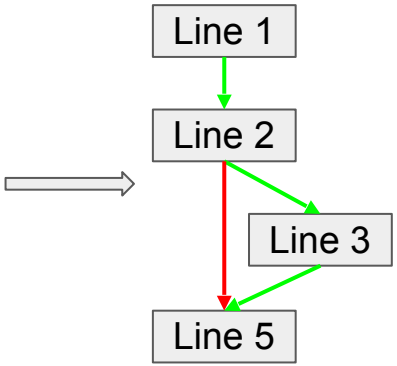
The traces only cover the “true” branch.

Line 2 → Line 5
is not allowed.

Heuristic **zCode**

- Find a path ***p*** not in the executed CFG, add ***p*** if all code on ***p*** is executed.

```
1 char *input = read_input();
2 if (everything_is_ok(input)) {
3     //do the job
4 }
5 return;
6
```



The traces only cover the “true” branch.

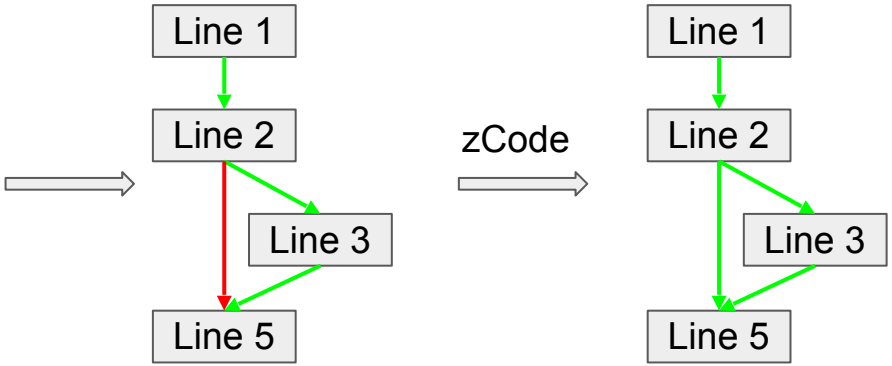
Line 2 → Line 5
is not allowed.

The path (*Line2 → Line 5*): **all code is executed.**

Heuristic **zCode**

➢ Find a path ***p*** not in the executed CFG, add ***p*** if all code on ***p*** is executed.

```
1 char *input = read_input();
2 if (everything_is_ok(input)) {
3     //do the job
4 }
5 return;
6
```



The traces only cover the “true” branch.

Line 2 → Line 5
is not allowed.

Line 2 → Line 5
is enabled.

The path (*Line2 → Line 5*): **all code is executed.**

Heuristic **zCall**

- Find a path ***p*** not in the executed CFG, add ***p*** if there are no call instructions on ***p***.

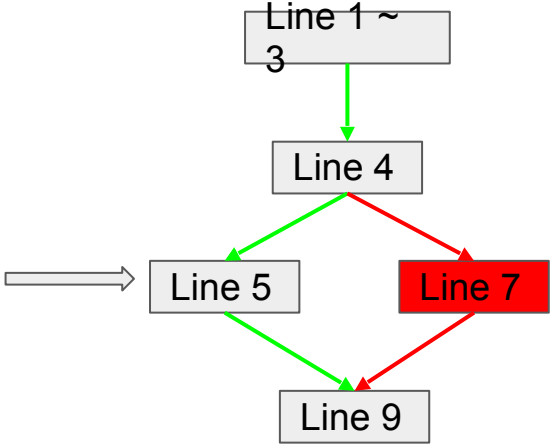
```
 1 int a = read_a();
 2 int b = read_b();
 3 unsigned int abs = 0;
 4 if (a > b) {
 5     abs = a - b;
 6 } else {
 7     abs = b - a;
 8 }
 9 return abs;
10
```

The traces only cover the “true” branch.

Heuristic **zCall**

- Find a path ***p*** not in the executed CFG, add ***p*** if there are no call instructions on ***p***.

```
 1 int a = read_a();
 2 int b = read_b();
 3 unsigned int abs = 0;
 4 if (a > b) {
 5     abs = a - b;
 6 } else {
 7     abs = b - a;
 8 }
 9 return abs;
10
```



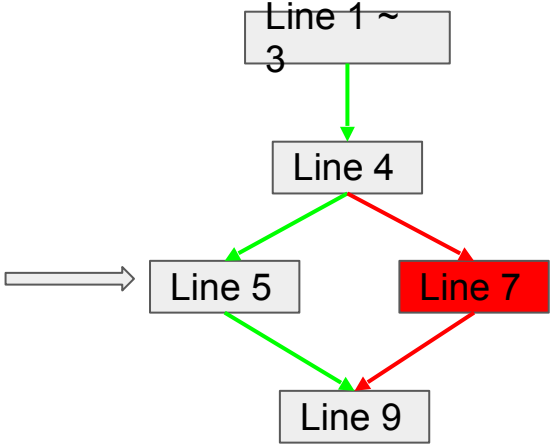
The traces only cover the “true” branch.

Line 7 is not executed.

Heuristic **zCall**

➢ Find a path ***p*** not in the executed CFG, add ***p*** if there are no call instructions on ***p***.

```
 1 int a = read_a();
 2 int b = read_b();
 3 unsigned int abs = 0;
 4 if (a > b) {
 5     abs = a - b;
 6 } else {
 7     abs = b - a;
 8 }
 9 return abs;
10
```



The traces only cover the “true” branch.

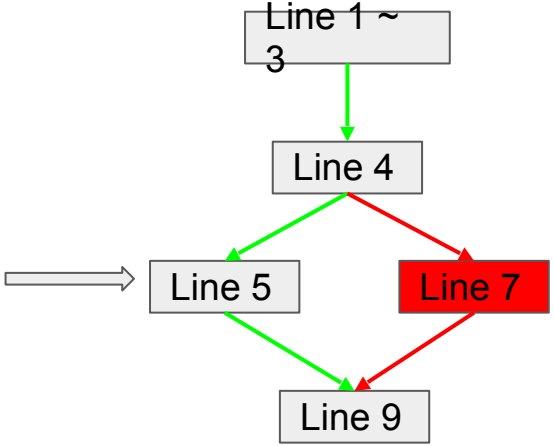
Line 7 is not executed.

The path (*Line 4* → *Line 7* → *Line 9*): *Line 7* is not executed, but **it does not call functions.**

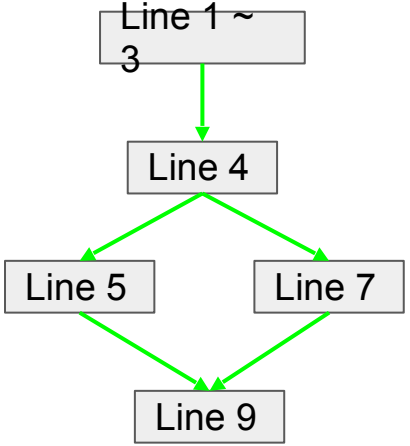
Heuristic **zCall**

➢ Find a path ***p*** not in the executed CFG, add ***p*** if there are no call instructions on ***p***.

```
 1 int a = read_a();
 2 int b = read_b();
 3 unsigned int abs = 0;
 4 if (a > b) {
 5     abs = a - b;
 6 } else {
 7     abs = b - a;
 8 }
 9 return abs;
10
```



zCall



The traces only cover the “true” branch.

Line 7 is not executed.

Line 4 → *Line 7* → *Line 9* is added.

The path (*Line 4* → *Line 7* → *Line 9*): *Line 7* is not executed, but **it does not call functions.**

Heuristic **zLib**

- Find a path ***p*** not in the executed CFG, add ***p*** if ***p*** does not call non-executed library functions.

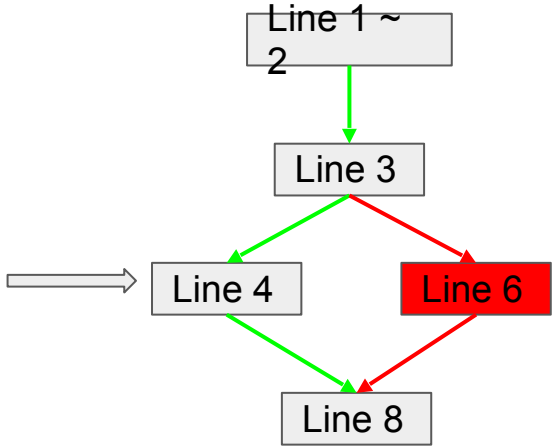
```
1 int a = read_a();
2 int b = read_b();
3 if (a > b) {
4   printf("a > b\n");
5 } else {
6   printf("a <= b\n");
7 }
8 return;
9
```

The traces only cover the “true” branch.

Heuristic **zLib**

- Find a path ***p*** not in the executed CFG, add ***p*** if ***p*** does not call non-executed library functions.

```
1 int a = read_a();
2 int b = read_b();
3 if (a > b) {
4   printf("a > b\n");
5 } else {
6   printf("a <= b\n");
7 }
8 return;
9
```



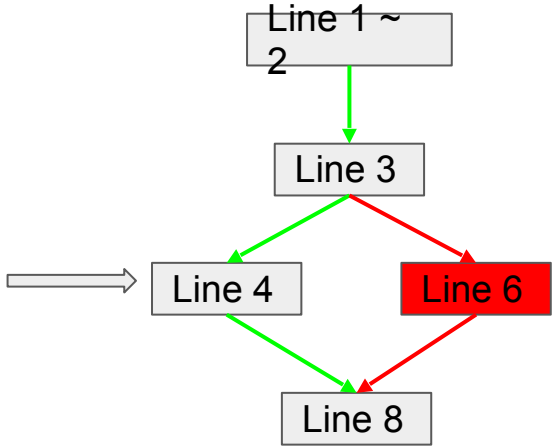
The traces only cover the “true” branch.

Line 6 is not executed.

Heuristic **zLib**

➢ Find a path ***p*** not in the executed CFG, add ***p*** if ***p*** does not call non-executed library functions.

```
1 int a = read_a();
2 int b = read_b();
3 if (a > b) {
4   printf("a > b\n");
5 } else {
6   printf("a <= b\n");
7 }
8 return;
9
```



The traces only cover the “true” branch.

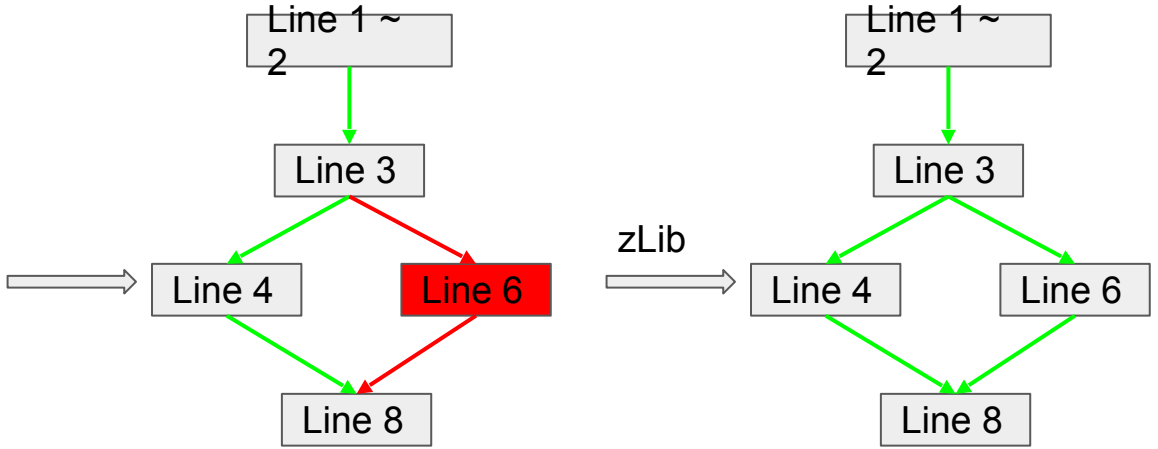
Line 6 is not executed.

The path (*Line 3* → *Line 6* → *Line 8*): *Line 6* is not executed and it calls “printf”, but **“printf” is executed.**

Heuristic **zLib**

- Find a path ***p*** not in the executed CFG, add ***p*** if ***p*** does not call non-executed library functions.

```
1 int a = read_a();
2 int b = read_b();
3 if (a > b) {
4   printf("a > b\n");
5 } else {
6   printf("a <= b\n");
7 }
8 return;
9
```



The traces only cover the “true” branch.

Line 6 is not executed.

Line 3 → *Line 6* → *Line 8* is added.

The path (*Line 3* → *Line 6* → *Line 8*): *Line 6* is not executed and it calls “printf”, but **“printf” is executed.**

Heuristic **zFunc**

- ➢ Find a path ***p*** not in the executed CFG, add ***p*** if ***p*** does not call non-executed library functions that share different functionalities.

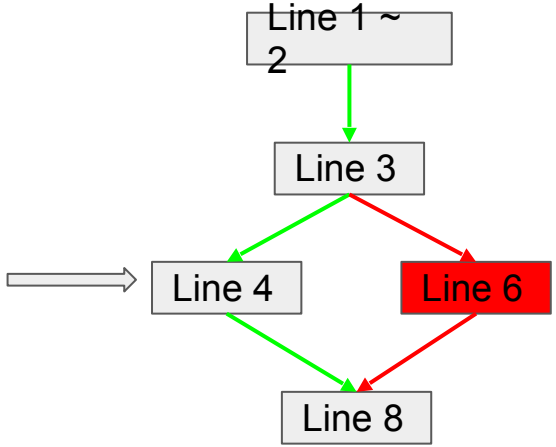
```
1 int a = read_a();
2 int b = read_b();
3 if (a > b) {
4   printf("a > b\n");
5 } else {
6   fprintf(stdout, "a <= b\n");
7 }
8 return;
9
```

The traces only cover the “true” branch.

Heuristic **zFunc**

- Find a path ***p*** not in the executed CFG, add ***p*** if ***p*** does not call non-executed library functions that share different functionalities.

```
1 int a = read_a();
2 int b = read_b();
3 if (a > b) {
4   printf("a > b\n");
5 } else {
6   fprintf(stdout, "a <= b\n");
7 }
8 return;
9
```



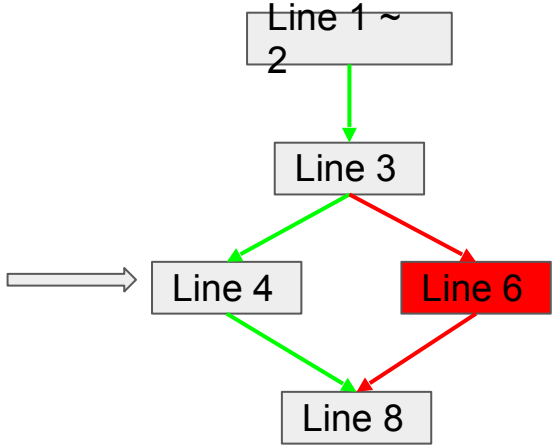
The traces only cover the “true” branch.

Line 6 is not executed.

Heuristic **zFunc**

- Find a path ***p*** not in the executed CFG, add ***p*** if ***p*** does not call non-executed library functions that share different functionalities.

```
1 int a = read_a();
2 int b = read_b();
3 if (a > b) {
4   printf("a > b\n");
5 } else {
6   fprintf(stdout, "a <= b\n");
7 }
8 return;
9
```



The traces only cover the “true” branch.

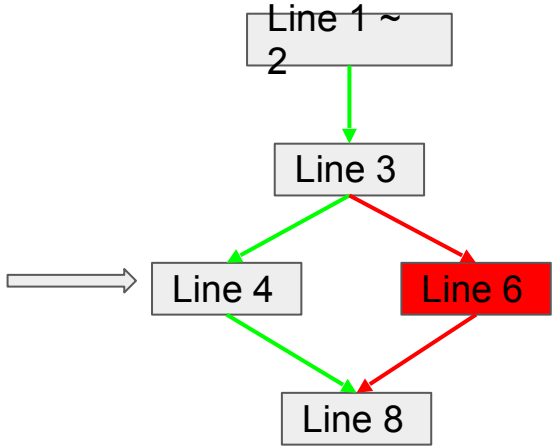
Line 6 is not executed.

The path (*Line 3* → *Line 6* → *Line 8*): *Line 6* is not executed and it calls “fprintf”, which is not executed, but: **(1) “printf” is executed; (2) “fprintf” shares the similar functionality with “printf”.**

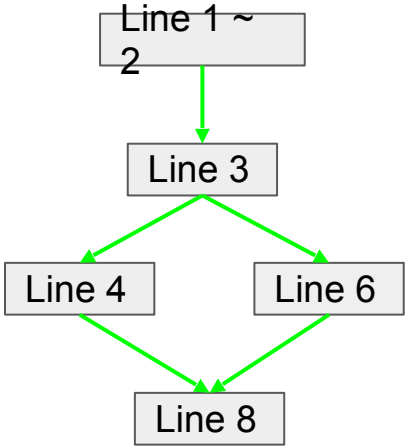
Heuristic **zFunc**

➢ Find a path ***p*** not in the executed CFG, add ***p*** if ***p*** does not call non-executed library functions that share different functionalities.

```
1 int a = read_a();
2 int b = read_b();
3 if (a > b) {
4   printf("a > b\n");
5 } else {
6   fprintf(stdout, "a <= b\n");
7 }
8 return;
9
```



zFunc



The traces only cover the “true” branch.

Line 6 is not executed.

Line 3 → *Line 6* → *Line 8* is added.

The path (*Line 3* → *Line 6* → *Line 8*): *Line 6* is not executed and it calls “fprintf”, which is not executed, but: **(1) “printf” is executed; (2) “fprintf” shares the similar functionality with “printf”.**

Generator

- **Assembler**
  - **Disassembles the binary based on the expanded CFG.**
  - **Symbolizes basic blocks.**
- Instrumenter
  - Concretizes targets of indirect calls/jumps.
  - Fixes callback function pointers.
  - Enforce allowed control-flows.
- Rewriter
  - Compiles the instrumented assembly code to an object file.
  - Copies the code section into original binary.

Generator

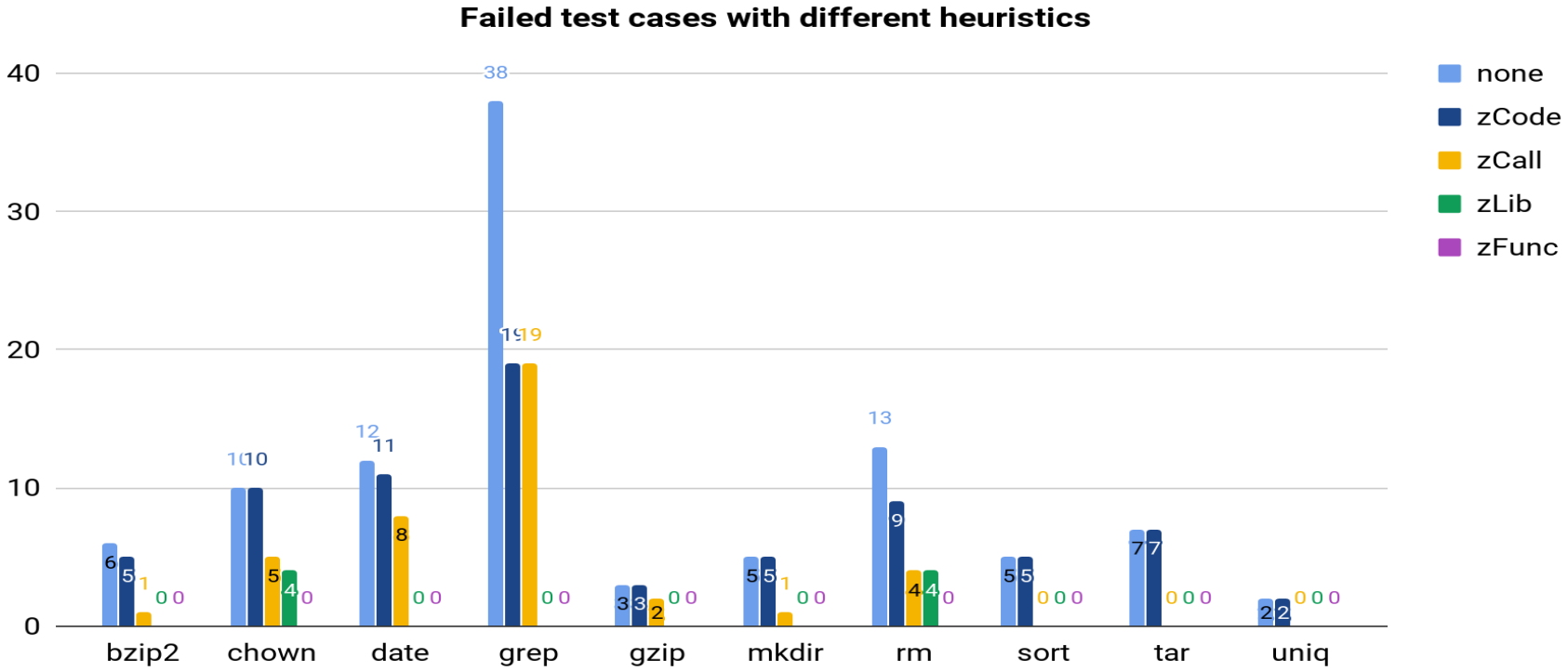
- Assembler
  - Disassembles the binary based on the expanded CFG.
  - Symbolizes basic blocks.
- **Instrumenter**
  - **Concretizes targets of indirect calls/jumps.**
  - **Fixes callback function pointers.**
  - **Enforces allowed control-flows.**
- Rewriter
  - Compiles the instrumented assembly code to an object file.
  - Copies the code section into original binary.

Generator

- Assembler
  - Disassembles the binary based on the expanded CFG.
  - Symbolizes basic blocks.
- Instrumenter
  - Concretizes targets of indirect calls/jumps.
  - Fixes callback function pointers.
  - Enforce allowed control-flows.
- **Rewriter**
  - **Compiles the instrumented assembly code to an object file.**
  - **Copies the code section into the original binary.**

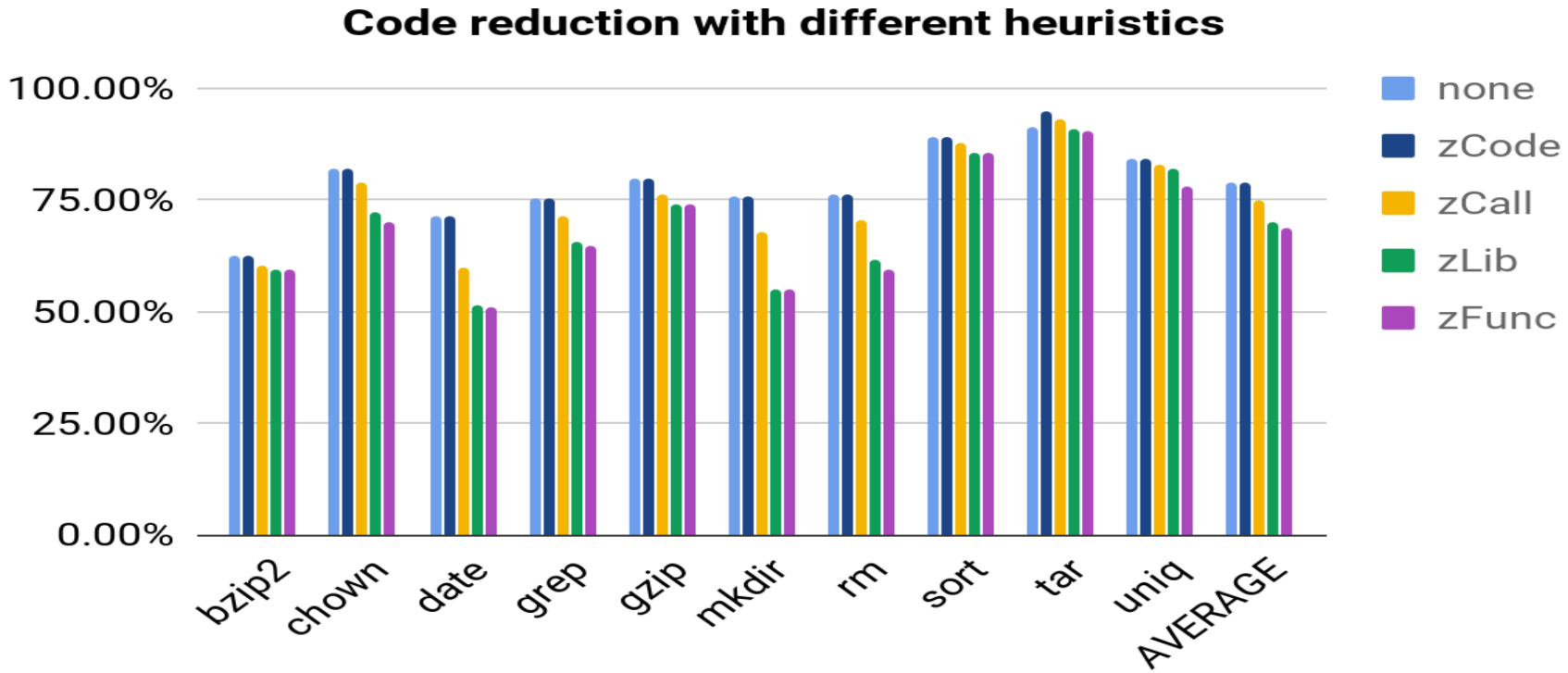
Effectiveness of Heuristics

- Run the debloated binaries on the different test cases.



Effectiveness of Heuristics

- Run the debloated binaries on the different test cases.



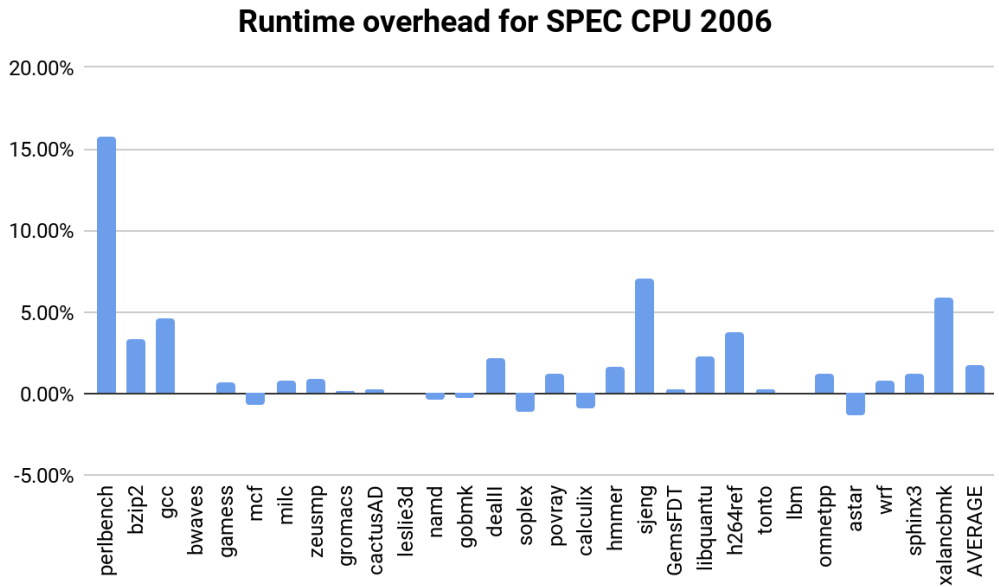
Security Benefits

Program	CVE	Orig	Chisel	Razor
bzip2	CVE-2010-0405	✔		
	CVE-2008-1372	✘	✔	
	CVE-2005-1260	✘	✔	
chown	CVE-2017-18018*	✔	✘	✘
date	CVE-2014-9471*	✔	✘	✔
grep	CVE-2015-1345*	✔	✘	✘
	CVE-2012-5667	✘	✔	
gzip	CVE-2005-1228*	✔	✘	✘
	CVE-2009-2624	✔		
	CVE-2010-0001	✔	✘	✘
mkdir	CVE-2005-1039*	✔		
rm	CVE-2015-1865*	✔		
tar	CVE-2016-6321*	✔	✘	✔

✔ binary is vulnerable to the CVE.
✘ binary is not vulnerable to the CVE.
* CVEs with * are evaluated by

Runtime Overhead

- On average, Razor introduces a 1.7% slowdown.
  - 15.8% overhead for *perlbench*



Real-world Software Debloating

- Firefox
  - Load Top 50 Alexa websites.
  - Randomly pick 25 websites for debloating, and use the other 25 websites for testing.
- FoxitReader
  - Open and scroll 55 different PDF files.
  - Randomly pick 15 files for debloating, and use the other 40 files for testing.

Heuristic	Firefox		FoxitReader	
	crash-sites	code-reduction	crash-PDFs	code-reduction
none	13	67.6%	39	89.8%
zCode	13	68.0%	10	89.9%
zCall	2	63.1%	5	89.4%
zLib	0	60.1%	0	87.0%
zFunc	0	60.0%	0	87.0%

Summary

- ➢ Performs code reduction for deployed **binaries**.
- ➢ Uses **heuristics** to infer related code for given test cases.
- ➢ Limitations:
  - ○ Original code section is not removed.
  - ○ Not 100% guaranteed robust.

Applications

- ➢ Legacy software on IoT devices.
- ➢ Software used for very specific purposes.
  - ○ Per-site browser isolation
  - ○ Bloated sensor software
    (e.g., optical, temperature, biometric)

Demos

https://github.com/cxreet/razor/wiki

Demo 1

- In this demo, we do not use any heuristics to debloat the binary.
- Run Razor step by step:
  - Trace the runnings.
  - Dump the executed instructions.
  - Instrument the executed instructions.
  - Rewrite the binary.
- To try the demo, please follow the instructions here: https://github.com/cxreet/razor/wiki/A-Simple-Demo-Without-Heuristics

Demo 1

```
🐋  8e22822c2f34 ~/workspace/razor # cd simple-demo/
🐋  8e22822c2f34 ~/workspace/razor/simple-demo # ls
debloat_simple.py  simple  simple.c
🐋  8e22822c2f34 ~/workspace/razor/simple-demo #
```

- simple.c → the demo's source code.
- simple → the demo binary.
- debloat_simple.py → the helper script for running Razor:
  - **-c trace -a *first_argument* -b *second_argument*** : trace the binary with arguments (*first_argument*, *second_argument*).
  - ***-c dump_inst*** : dump the executed instructions.
  - ***-c instrument*** : instrument the executed instructions.
  - ***-c rewrite*** : rewrite the binary.
  - ***-c clean*** : clean the outputs under current directory.

Simple.c

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Simple.c

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Two arguments: ***taken***, ***target***.

Simple.c

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Two arguments: ***taken***, ***target***.

Conditional branch: Print out “Taken” when ***taken*** is 1, else print out “Non-taken”.

Simple.c

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Two arguments: ***taken***, ***target***.

Conditional branch: Print out “Taken” when ***taken*** is 1, else print out “Non-taken”.

Indirect call: It calls “*printYes*” when ***target*** is ‘y’, otherwise, it calls “*printNo*”.

Goal

- Use the following inputs for tracing:
  - taken = 1 or 0, target = ‘y’
- Debloat the binary to only support the inputs used for tracing.

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

❶: taken = 1, target = ‘y’
❷: taken = 0, target = ‘y’

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    [redacted] Remove it!
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    [redacted] Remove it!
  (*fptr)();

  return 0;
}
```

Step 1: Trace the Runnings

```
🐳  ad3d20f94889 ~/workspace/razor/simple-demo # python debloat_simple.py -c trace -a 1 -b y
../tracers/scripts/trace_with_dynamorio.sh ./simple 1 y
+ bin_cmd='./simple 1 y'
++ dirname ../tracers/scripts/trace_with_dynamorio.sh
+ cur_dir=../tracers/scripts
++ readlink -m ../tracers/scripts/../
+ root_dir=/root/workspace/razor/tracers
+ /root/workspace/razor/tracers/dynamorio/bin64/drrun -c /root/workspace/razor/tracers/bin/libcbr_indcall.so -- ./simple 1 y
+ mv /root/workspace/razor/tracers/bin/cbr_indcall.simple.00033.0000.log ./
mkdir -p logs; mv *.log logs/
🐳  ad3d20f94889 ~/workspace/razor/simple-demo # python debloat_simple.py -c trace -a 0 -b y
../tracers/scripts/trace_with_dynamorio.sh ./simple 0 y
+ bin_cmd='./simple 0 y'
++ dirname ../tracers/scripts/trace_with_dynamorio.sh
+ cur_dir=../tracers/scripts
++ readlink -m ../tracers/scripts/../
+ root_dir=/root/workspace/razor/tracers
+ /root/workspace/razor/tracers/dynamorio/bin64/drrun -c /root/workspace/razor/tracers/bin/libcbr_indcall.so -- ./simple 0 y
+ mv /root/workspace/razor/tracers/bin/cbr_indcall.simple.00043.0000.log ./
mkdir -p logs; mv *.log logs/
🐳  ad3d20f94889 ~/workspace/razor/simple-demo # python debloat_simple.py -c merge_log
python ../stitcher/src/merge_log.py ./logs simple
mv logs/simple-trace.log ./
🐳  ad3d20f94889 ~/workspace/razor/simple-demo #
```

- ➢ Razor uses Dynamorio to trace the binary with inputs: (1, y), (0, y)
- ➢ Razor merges the logs for different runnings → ***simple-trace.log***

Tracing Results

```
🐋  ccaa1b4f2a36 ~/workspace/razor/simple-demo # cat simple-trace.log
```

Tracing Results

```
ccaa1b4f2a36 ~/workspace/razor/simple-demo # cat simple-trace.log
```

Executed Basic Blocks

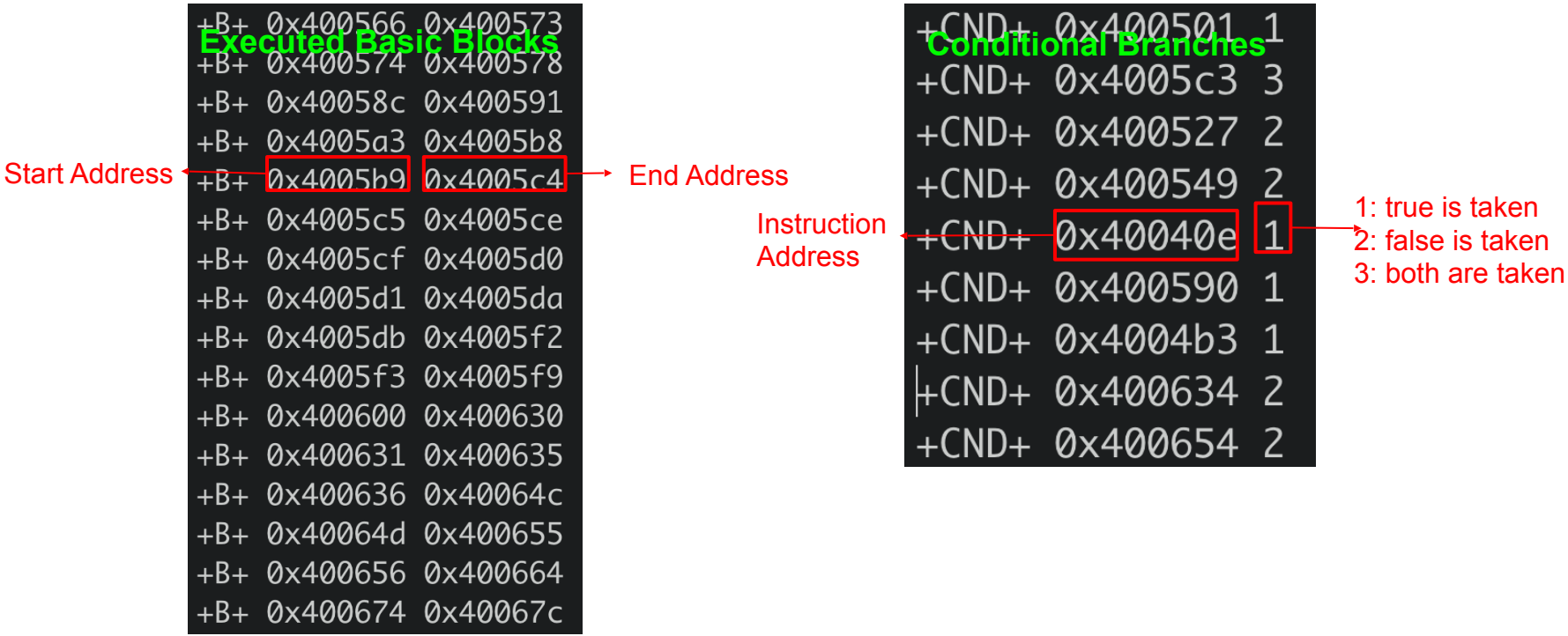
```
+B+ 0x400566 0x400573
+B+ 0x400574 0x400578
+B+ 0x40058c 0x400591
+B+ 0x4005a3 0x4005b8
+B+ 0x4005b9 0x4005c4
+B+ 0x4005c5 0x4005ce
+B+ 0x4005cf 0x4005d0
+B+ 0x4005d1 0x4005da
+B+ 0x4005db 0x4005f2
+B+ 0x4005f3 0x4005f9
+B+ 0x400600 0x400630
+B+ 0x400631 0x400635
+B+ 0x400636 0x40064c
+B+ 0x40064d 0x400655
+B+ 0x400656 0x400664
+B+ 0x400674 0x40067c
```

Start Address

End Address

Tracing Results

```
ccaa1b4f2a36 ~/workspace/razor/simple-demo # cat simple-trace.log
```



Tracing Results

```
ccaa1b4f2a36 ~/workspace/razor/simple-demo # cat simple-trace.log
```

Executed Basic Blocks

```
+B+ 0x400566 0x400573
+B+ 0x400574 0x400578
+B+ 0x40058c 0x400591
+B+ 0x4005a3 0x4005b8
+B+ 0x4005b9 0x4005c4
+B+ 0x4005c5 0x4005ce
+B+ 0x4005cf 0x4005d0
+B+ 0x4005d1 0x4005da
+B+ 0x4005db 0x4005f2
+B+ 0x4005f3 0x4005f9
+B+ 0x400600 0x400630
+B+ 0x400631 0x400635
+B+ 0x400636 0x40064c
+B+ 0x40064d 0x400655
+B+ 0x400656 0x400664
+B+ 0x400674 0x40067c
```

Start Address

End Address

Conditional Branches

```
+CND+ 0x400501 1
+CND+ 0x4005c3 3
+CND+ 0x400527 2
+CND+ 0x400549 2
+CND+ 0x40040e 1
+CND+ 0x400590 1
+CND+ 0x4004b3 1
+CND+ 0x400634 2
+CND+ 0x400654 2
```

Instruction Address

1: true is taken
2: false is taken
3: both are taken

Indirect Calls

```
+IND+ 0x400440 0x400446#2
+IND+ 0x400450 0x400456#2
+IND+ 0x400426 0x7f56238a8430#3 0x7ffa0eb93430#3
+IND+ 0x400649 0x400540#2
+IND+ 0x400430 0x7f56233c3910#1 0x7ffa0e6ae910#1 0x400436#2
+IND+ 0x4005f1 0x400566#2
```

Instruction Address

Target Address # Frequency

Executed Basic Blocks

```
+B+ 0x400566 0x400573
+B+ 0x400574 0x400578
+B+ 0x40058c 0x400591
+B+ 0x4005a3 0x4005b8
+B+ 0x4005b9 0x4005c4
+B+ 0x4005c5 0x4005ce
+B+ 0x4005cf 0x4005d0
+B+ 0x4005d1 0x4005da
+B+ 0x4005db 0x4005f2
+B+ 0x4005f3 0x4005f9
+B+ 0x400600 0x400630
+B+ 0x400631 0x400635
+B+ 0x400636 0x40064c
+B+ 0x40064d 0x400655
+B+ 0x400656 0x400664
+B+ 0x400674 0x40067c
```

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Executed Basic Blocks

```
+B+ 0x400566 0x400573
+B+ 0x400574 0x400578
+B+ 0x40058c 0x400591
+B+ 0x4005a3 0x4005b8
+B+ 0x4005b9 0x4005c4
+B+ 0x4005c5 0x4005ce
+B+ 0x4005cf 0x4005d0
+B+ 0x4005d1 0x4005da
+B+ 0x4005db 0x4005f2
+B+ 0x4005f3 0x4005f9
+B+ 0x400600 0x400630
+B+ 0x400631 0x400635
+B+ 0x400636 0x40064c
+B+ 0x40064d 0x400655
+B+ 0x400656 0x400664
+B+ 0x400674 0x40067c
```

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Executed Basic Blocks

```
+B+ 0x400566 0x400573
+B+ 0x400574 0x400578
+B+ 0x40058c 0x400591
+B+ 0x4005a3 0x4005b8
+B+ 0x4005b9 0x4005c4
+B+ 0x4005c5 0x4005ce
+B+ 0x4005cf 0x4005d0
+B+ 0x4005d1 0x4005da
+B+ 0x4005db 0x4005f2
+B+ 0x4005f3 0x4005f9
+B+ 0x400600 0x400630
+B+ 0x400631 0x400635
+B+ 0x400636 0x40064c
+B+ 0x40064d 0x400655
+B+ 0x400656 0x400664
+B+ 0x400674 0x40067c
```

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Conditional Branches

```
+CND+ 0x400501 1
+CND+ 0x4005c3 3
+CND+ 0x400527 2
+CND+ 0x400549 2
+CND+ 0x40040e 1
+CND+ 0x400590 1
+CND+ 0x4004b3 1
+CND+ 0x400634 2
+CND+ 0x400654 2
```

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Conditional Branches

```
+CND+ 0x400501 1
+CND+ 0x4005c3 3
+CND+ 0x400527 2
+CND+ 0x400549 2
+CND+ 0x40040e 1
+CND+ 0x400590 1
+CND+ 0x4004b3 1
+CND+ 0x400634 2
+CND+ 0x400654 2
```

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Conditional Branches

```
+CND+ 0x400501 1
+CND+ 0x4005c3 3
+CND+ 0x400527 2
+CND+ 0x400549 2
+CND+ 0x40040e 1
+CND+ 0x400590 1
+CND+ 0x4004b3 1
+CND+ 0x400634 2
+CND+ 0x400654 2
```

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

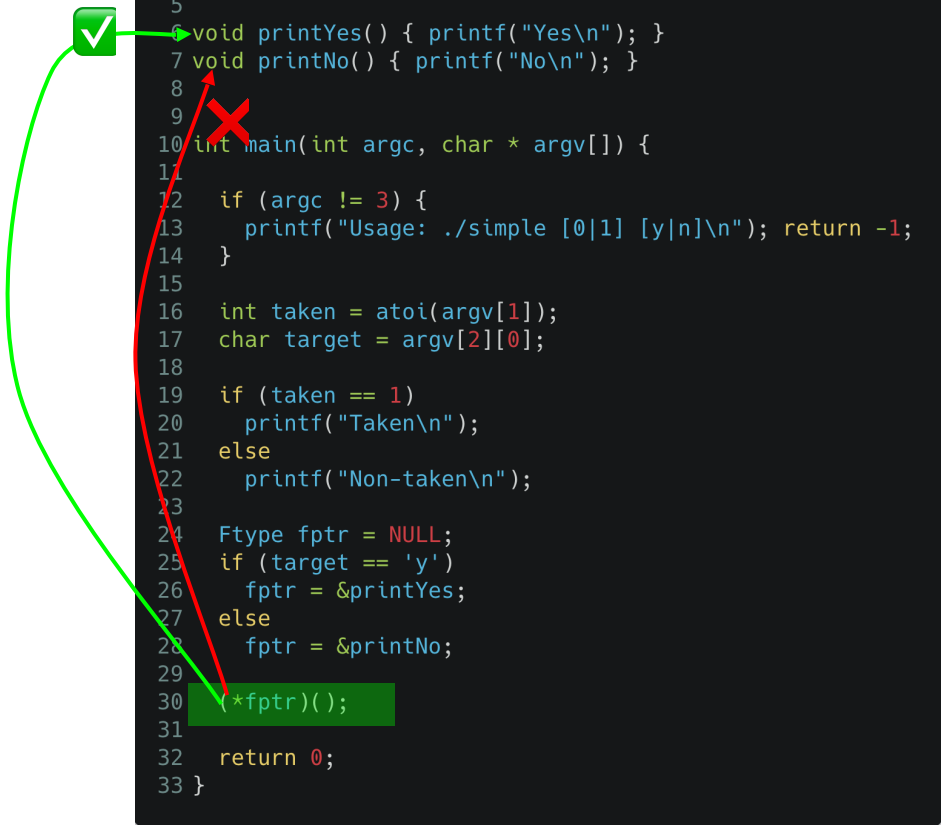
Indirect Calls

```
+IND+ 0x400440 0x400446#2
+IND+ 0x400450 0x400456#2
+IND+ 0x400426 0x7f56238a8430#3
+IND+ 0x400649 0x400540#2
+IND+ 0x400430 0x7f56233c3910#1
+IND+ 0x4005f1 0x400566#2
```

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Indirect Calls

```
+IND+ 0x400440 0x400446#2
+IND+ 0x400450 0x400456#2
+IND+ 0x400426 0x7f56238a8430#3
+IND+ 0x400649 0x400540#2
+IND+ 0x400430 0x7f56233c3910#1
+IND+ 0x4005f1 0x400566#2
```



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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

Step 2: Dump the Executed Instructions

```
🐳  ad3d20f94889 ~/workspace/razor/simple-demo # python debloat_simple.py -c dump_inst
python ../stitcher/src/instr_dumper.py ./simple-trace.log ./simple ./instr.s
🐳  ad3d20f94889 ~/workspace/razor/simple-demo # cat instr.s
```

- First, Razor disassembles the binary.
- Second, Razor dumps the executed code based on the “executed basic blocks” in traces.
- The executed code is dumped as assembly instructions, which is saved in file ***instr.s***.

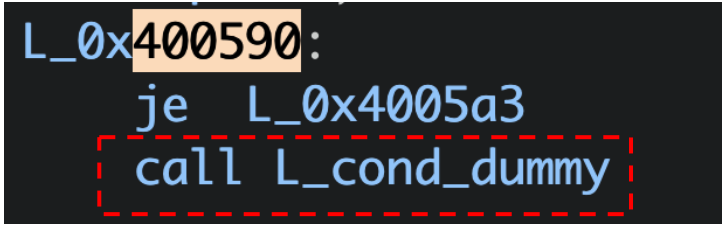
Step 3: Instrument the Executed instructions

```
🐋  ad3d20f94889 ~/workspace/razor/simple-demo # python debloat_simple.py -c instrument
python ../stitcher/src/find_symbols.py ./simple ./instr.s
python ../stitcher/src/stitcher.py ./simple-trace.log ./simple ./simple.s ./callbacks.txt
🐋  ad3d20f94889 ~/workspace/razor/simple-demo # cat ./simple.s
```

- Based on the control flow informations (i.e., conditional branches, indirect calls/jumps), Razor instruments the “conditional branches” and “indirect calls/jumps” in ***instr.s***.
- The instrumented instructions are save in file ***simple.s***.

Conditional Branch Instrumentation

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```



Conditional Branch Instrumentation

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef void (*Ftype)();
5
6 void printYes() { printf("Yes\n"); }
7 void printNo() { printf("No\n"); }
8
9
10 int main(int argc, char * argv[]) {
11
12   if (argc != 3) {
13     printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
14   }
15
16   int taken = atoi(argv[1]);
17   char target = argv[2][0];
18
19   if (taken == 1)
20     printf("Taken\n");
21   else
22     printf("Non-taken\n");
23
24   Ftype fptr = NULL;
25   if (target == 'y')
26     fptr = &printYes;
27   else
28     fptr = &printNo;
29
30   (*fptr)();
31
32   return 0;
33 }
```

```
L_0x400590:
    je  L_0x4005a3
    call L_cond_dummy
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef void (*Ftype)();
5
6 void printYes() { printf("Yes\n"); }
7 void printNo() { printf("No\n"); }
8
9
10 int main(int argc, char * argv[]) {
11
12   if (argc != 3) {
13     42/0; // Throw Floating Point Exception.
14   }
15
16   int taken = atoi(argv[1]);
17   char target = argv[2][0];
18
19   if (taken == 1)
20     printf("Taken\n");
21   else
22     printf("Non-taken\n");
23
24   Ftype fptr = NULL;
25   if (target == 'y')
26     fptr = &printYes;
27   else
28     fptr = &printNo;
29
30   (*fptr)();
31
32   return 0;
33 }
```

Indirect Call Instrumentation

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

```
L_indcall_0x4005f1_target_0x400566:
    cmp rdx, 0x400566
    jne L_indcall_dummy
    call L_0x400566
```

Indirect Call Instrumentation

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

```
L_indcall_0x4005f1_target_0x400566:
    cmp rdx, 0x400566
    jne L_indcall_dummy
    call L_0x400566
```

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  if (fptr == &printYes)
    printYes()
  else
    43/0; // Throw Floating Point Exception
  return 0;
}
```

Step 4: Rewrite the Binary

```
🐳  ad3d20f94889 ~/workspace/razor/simple-demo # python debloat_simple.py -c rewrite
python ../stitcher/src/merge_bin.py simple simple.s
🐳  ad3d20f94889 ~/workspace/razor/simple-demo #
```

- Razor compiles the instrumented instructions (i.e., *simple.s*) into an object file.
- Razor rewrites the original binary with the object file.
- The debloated binary is ***./simple_temp/simple.debloated***.

The Debloated Binary

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
void printNo() { printf("No\n"); }


int main(int argc, char * argv[]) {

  if (argc != 3) {
    printf("Usage: ./simple [0|1] [y|n]\n"); return -1;
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  else
    fptr = &printNo;

  (*fptr)();

  return 0;
}
```

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

typedef void (*Ftype)();

void printYes() { printf("Yes\n"); }
[redacted]


int main(int argc, char * argv[]) {

  if (argc != 3) {
    42/0; // Throw Floating Point Exception.
  }

  int taken = atoi(argv[1]);
  char target = argv[2][0];

  if (taken == 1)
    printf("Taken\n");
  else
    printf("Non-taken\n");

  Ftype fptr = NULL;
  if (target == 'y')
    fptr = &printYes;
  if (fptr == &printYes)
    printYes()
  else
    43/0; // Throw Floating Point Exception
  return 0;
}
```

Step 5: Test the Debloated Binary

```
🐋  ccaa1b4f2a36 ~/workspace/razor/simple-demo # ./simple_temp/simple.debloated 0 n
Non-taken
Floating point exception (core dumped)
🐋  ccaa1b4f2a36 ~/workspace/razor/simple-demo # ./simple_temp/simple.debloated 1 n
Taken
Floating point exception (core dumped)
🐋  ccaa1b4f2a36 ~/workspace/razor/simple-demo # ./simple_temp/simple.debloated 0 y
Non-taken
Yes
🐋  ccaa1b4f2a36 ~/workspace/razor/simple-demo # ./simple_temp/simple.debloated 1 y
Taken
Yes
🐋  ccaa1b4f2a36 ~/workspace/razor/simple-demo # ./simple_temp/simple.debloated 1 y 1
Floating point exception (core dumped)
🐋  ccaa1b4f2a36 ~/workspace/razor/simple-demo #
```

- The debloated binary only takes in two arguments, the argument ***taken*** can be any integer, and the argument ***target*** can only be `y`.

- Razor rewrites a binary that supports the same inputs used for tracing.
- For different inputs that share same functionalities with the inputs used for tracing, the debloated binary does not support them. → Over-debloating.
- Razor uses heuristics to infer code not covered by given inputs.

Demo 2

- In this demo, Razor applies different heuristics to debloat the binary.
- Run Razor step by step:
  - Trace the runnings.
  - **Use heuristics to extend the original traces.**
  - Dump the executed instructions.
  - Do the instrumentation.
  - Rewrite the binary.
- To try the demo, please follow the instructions here: https://github.com/cxreet/razor/wiki/A-Simple-Demo-With-Heuristics

Demo 2

```
🐳  ccaa1b4f2a36 ~/workspace/razor/heuristic-demo # cd ~/workspace/razor/heuristic-demo/
🐳  ccaa1b4f2a36 ~/workspace/razor/heuristic-demo # ls
apply_heuristic.sh  debloat_simple.py  simple  simple.c
🐳  ccaa1b4f2a36 ~/workspace/razor/heuristic-demo #
```

- simple.c → the demo's source code.
- simple → the demo binary.
- debloat_simple.py → the helper script for running Razor without heuristics:
  - **-c trace -a *first_argument* -b *second_argument*** : trace the binary with arguments (*first_argument*, *second_argument*).
  - ***-c dump_inst*** : dump the executed instructions.
  - ***-c instrument*** : instrument the executed instructions.
  - ***-c rewrite*** : rewrite the binary.
  - ***-c clean*** : clean the outputs under current directory.
- apply_heuristic.sh → the helper script for running Razor with heuristics.

Simple.c

```c
int main(int argc, char * argv[]) {
    if (argc != 3) {
        printf("Usage: ./simple n1 n2\n");
        return -1;
    }

    int a = atoi(argv[1]);
    int b = atoi(argv[2]);

    test_heuristic_one(a, b);
    test_heuristic_two(a, b);
    test_heuristic_three(a, b);
    test_heuristic_four(a, b);

  return 0;
}
```

- ➔ It takes two arguments: ***a*** and ***b***.
- ➔ The ***main*** function calls four functions:
  - ***test_heuristic_one***
  - ***test_heuristic_two***
  - ***test_heuristic_three***
  - ***test_hrusitic_four***

➢ *test_heuristic_one*

```c
void test_heuristic_one(int a, int b) {
    printf("Testing heuristic one...\n");
    if (a > b) {
        printf("%d > %d\n", a , b);
    }
    printf("Done.\n");

    return;
}
```

➢ *test_heuristic_two*

```c
int test_heuristic_two(int a, int b) {
    printf("Testing heuristic two...\n");
    int ret = 0;
    if (a > b) {
        printf("%d > %d\n", a, b);
        ret = a - b;
    } else {
        ret = b - a;
    }
    printf("Done.\n");
    return ret;
}
```

➢ *test_heuristic_three*

```c
void test_heuristic_three(int a, int b) {
    printf("Testing herusitc three...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        printf("%d <= %d\n", a, b);
    }
    printf("Done.\n");
}
```

➢ *test_heuristic_four*

```c
void call_new_libcalls(int a, int b) {
        putw(a, stdout);
        printf(" <= ");
        putw(b, stdout);
        printf("\n");
}

void test_heuristic_four(int a, int b) {
    printf("Testing heuristic four...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        call_new_libcalls(a, b);
    }
    printf("Done.\n");
}
```

What if the inputs for tracing only cover the case when ***a*** > ***b***?

➢ The false branches are removed without heuristics.

```c
void test_heuristic_one(int a, int b) {
    printf("Testing heuristic one...\n");
    if (a > b) {
        printf("%d > %d\n", a , b);
    }
    printf("Done.\n");

    return;
}
```

```c
int test_heuristic_two(int a, int b) {
    printf("Testing heuristic two...\n");
    int ret = 0;
    if (a > b) {
        printf("%d > %d\n", a, b);
        ret = a - b;
    } else {
        ret = b - a;
    }
    printf("Done.\n");
    return ret;
}
```

```c
void test_heuristic_three(int a, int b) {
    printf("Testing herusitc three...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        printf("%d <= %d\n", a, b);
    }
    printf("Done.\n");
}
```

```c
void call_new_libcalls(int a, int b) {
        putw(a, stdout);
        printf(" <= ");
        putw(b, stdout);
        printf("\n");
}

void test_heuristic_four(int a, int b) {
    printf("Testing heuristic four...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        call_new_libcalls(a, b);
    }
    printf("Done.\n");
}
```

➢ The false branches are removed without heuristics.

```c
void test_heuristic_one(int a, int b) {
    printf("Testing heuristic one...\n");
    if (a > b) {
        printf("%d > %d\n", a , b);
    }
    printf("Done.\n");

    return;
}
```

```c
int test_heuristic_two(int a, int b) {
    printf("Testing heuristic two...\n");
    int ret = 0;
    if (a > b) {
        printf("%d > %d\n", a, b);
        ret = a - b;
    } else {
        ret = b - a;
    }
    printf("Done.\n");
    return ret;
}
```

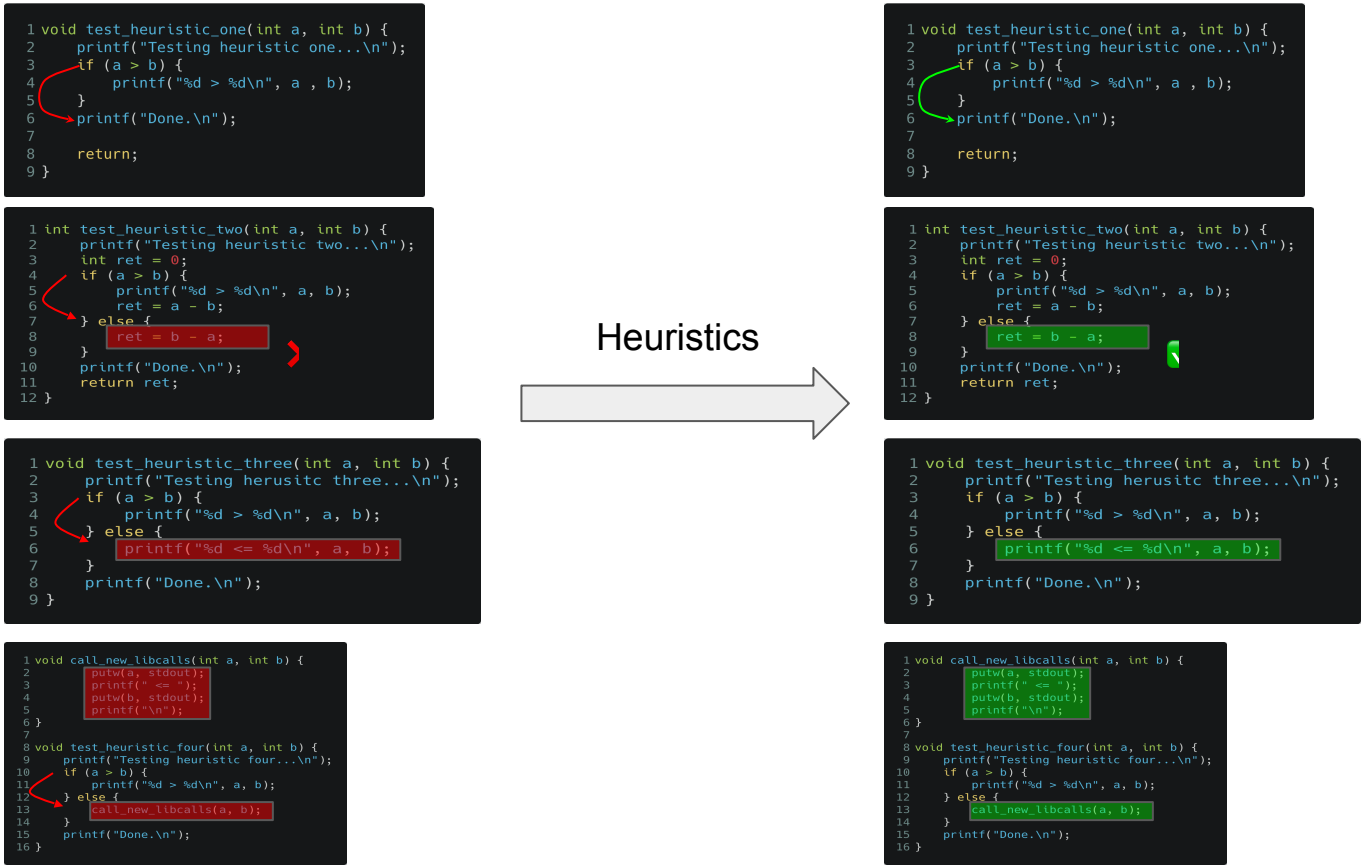
Over-Debloating!

```c
void test_heuristic_three(int a, int b) {
    printf("Testing herusitc three...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        printf("%d <= %d\n", a, b);
    }
    printf("Done.\n");
}
```

```c
_new_libcalls(int a, int b) {
    utw(a, stdout);
    printf(" <= ");
    putw(b, stdout);
    printf("\n");
}

void test_heuristic_four(int a, int b) {
    printf("Testing heuristic four...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        call_new_libcalls(a, b);
    }
    printf("Done.\n");
}
```

Goal

- ➢ Use inputs that only covers the ***true*** branches for tracing.
- ➢ Apply heuristics to generate a binary that includes the **false** branches.



Step 1: Trace the Runnings

```
🐳  83e097359693 ~/workspace/razor/heuristic-demo # python debloat_simple.py -c trace -a 2 -b 1
../tracers/scripts/trace_with_dynamorio.sh ./simple 2 1
+ bin_cmd='./simple 2 1'
++ dirname ../tracers/scripts/trace_with_dynamorio.sh
+ cur_dir=../tracers/scripts
++ readlink -m ../tracers/scripts/../
+ root_dir=/root/workspace/razor/tracers
+ /root/workspace/razor/tracers/dynamorio/bin64/drrun -c /root/workspace/razor/tracers/bin/libcbr_indcall.so -- ./simple 2 1
+ mv /root/workspace/razor/tracers/bin/cbr_indcall.simple.00036.0000.log ./
mkdir -p logs; mv *.log logs/
🐳  83e097359693 ~/workspace/razor/heuristic-demo # python debloat_simple.py -c merge_log
python ../stitcher/src/merge_log.py ./logs simple
mv logs/simple-trace.log ./
🐳  83e097359693 ~/workspace/razor/heuristic-demo #
```

- Use Dynamorio to trace the binary with the input: (a = 2, b = 1).
- The tracing results are saved in the file ***simple-trace.log***.

Step 2 & 3: Dump Executed Instructions and Instrument

```
🐳  83e097359693 ~/workspace/razor/heuristic-demo # python debloat_simple.py -c dump_inst
python ../stitcher/src/instr_dumper.py ./simple-trace.log ./simple ./instr.s
🐳  83e097359693 ~/workspace/razor/heuristic-demo # python debloat_simple.py -c instrument
python ../stitcher/src/find_symbols.py ./simple ./instr.s
```

- Step 2: Dump the executed instructions **without** heuristics.
  - First, Razor disassembles the binary.
  - Second, Razor dumps the executed code based on the “executed basic blocks” in traces.
  - The executed code is dumped as assembly instructions, which is saved in file ***instr.s***.
- Step 3: Instrument the executed instructions.
  - Based on the control flow informations (i.e., conditional branches, indirect calls/jumps), Razor instruments the “conditional branches” and “indirect calls/jumps” in ***instr.s***.
  - The instrumented instructions are save in file ***simple.s***.

Step 4 & 5: Rewrite the Binary and Run the Debloated Binary

```
🐳  83e097359693 ~/workspace/razor/heuristic-demo # python debloat_simple.py -c rewrite
python ../stitcher/src/merge_bin.py simple simple.s
🐳  83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 2 1
Testing heuristic one...
2 > 1
Done.
Testing heuristic two...
2 > 1
Done.
Testing herusitc three...
2 > 1
Done.
Testing heuristic four...
2 > 1
Done.
🐳  83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Floating point exception
🐳  83e097359693 ~/workspace/razor/heuristic-demo #
```

```c
void test_heuristic_one(int a, int b) {
    printf("Testing heuristic one...\n");
    if (a > b) {
        printf("%d > %d\n", a , b);
    }
    printf("Done.\n");

    return;
}
```

```c
int test_heuristic_two(int a, int b) {
    printf("Testing heuristic two...\n");
    int ret = 0;
    if (a > b) {
        printf("%d > %d\n", a, b);
        ret = a - b;
    } else {
        ret = b - a;
    }
    printf("Done.\n");
    return ret;
}
```

```c
void test_heuristic_three(int a, int b) {
    printf("Testing herusitc three...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        printf("%d <= %d\n", a, b);
    }
    printf("Done.\n");
}
```

```c
void call_new_libcalls(int a, int b) {
    putw(a, stdout);
    printf(" <= ");
    putw(b, stdout);
    printf("\n");
}

void test_heuristic_four(int a, int b) {
    printf("Testing heuristic four...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        call_new_libcalls(a, b);
    }
    printf("Done.\n");
}
```

- Step 4: Rewrite the binary.
  - Razor compiles the instrumented instructions (i.e., *simple.s*) into an object file.
  - Razor rewrites the original binary with the object file.
  - The debloated binary is ***./simple_temp/simple.debloated***.

Step 4 & 5: Rewrite the Binary and Run the Debloated Binary

```
83e097359693 ~/workspace/razor/heuristic-demo # python debloat_simple.py -c rewrite
python ../stitcher/src/merge_bin.py simple simple.s
83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 2 1
Testing heuristic one...
2 > 1
Done.
Testing heuristic two...
2 > 1
Done.
Testing herusitc three...
2 > 1
Done.
Testing heuristic four...
2 > 1
Done.
83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Floating point exception
83e097359693 ~/workspace/razor/heuristic-demo #
```

Only **true** branches are triggered!

```c
void test_heuristic_one(int a, int b) {
    printf("Testing heuristic one...\n");
    if (a > b) {
        printf("%d > %d\n", a , b);
    }
    printf("Done.\n");

    return;
}
```

```c
int test_heuristic_two(int a, int b) {
    printf("Testing heuristic two...\n");
    int ret = 0;
    if (a > b) {
        printf("%d > %d\n", a, b);
        ret = a - b;
    } else {
        ret = b - a;
    }
    printf("Done.\n");
    return ret;
}
```

```c
void test_heuristic_three(int a, int b) {
    printf("Testing herusitc three...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        printf("%d <= %d\n", a, b);
    }
    printf("Done.\n");
}
```

```c
void call_new_libcalls(int a, int b) {
    putw(a, stdout);
    printf(" <= ");
    putw(b, stdout);
    printf("\n");
}

void test_heuristic_four(int a, int b) {
    printf("Testing heuristic four...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        call_new_libcalls(a, b);
    }
    printf("Done.\n");
}
```

- Step 4: Rewrite the binary.
  - Razor compiles the instrumented instructions (i.e., *simple.s*) into an object file.
  - Razor rewrites the original binary with the object file.
  - The debloated binary is ***./simple_temp/simple.debloated***.

Step 4 & 5: Rewrite the Binary and Run the Debloated Binary

```
83e097359693 ~/workspace/razor/heuristic-demo # python debloat_simple.py -c rewrite
python ../stitcher/src/merge_bin.py simple simple.s
83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 2 1
Testing heuristic one...
2 > 1
Done.
Testing heuristic two...
2 > 1
Done.
Testing herusitc three...
2 > 1
Done.
Testing heuristic four...
2 > 1
Done.
83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Floating point exception
83e097359693 ~/workspace/razor/heuristic-demo #
```

Only **true** branches are triggered!

False branches are triggered!

```c
void test_heuristic_one(int a, int b) {
    printf("Testing heuristic one...\n");
    if (a > b) {
        printf("%d > %d\n", a , b);
    }
    printf("Done.\n");

    return;
}
```

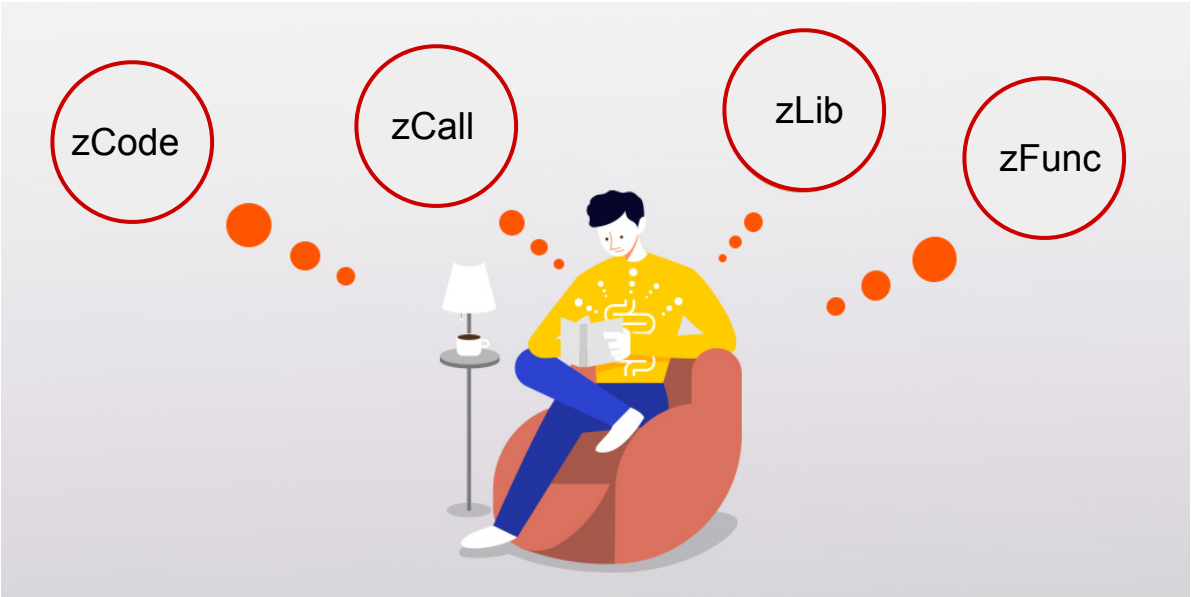
```c
int test_heuristic_two(int a, int b) {
    printf("Testing heuristic two...\n");
    int ret = 0;
    if (a > b) {
        printf("%d > %d\n", a, b);
        ret = a - b;
    } else {
        ret = b - a;
    }
    printf("Done.\n");
    return ret;
}
```

```c
void test_heuristic_three(int a, int b) {
    printf("Testing herusitc three...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        printf("%d <= %d\n", a, b);
    }
    printf("Done.\n");
}
```

```c
void call_new_libcalls(int a, int b) {
    putw(a, stdout);
    printf(" <= ");
    putw(b, stdout);
    printf("\n");
}

void test_heuristic_four(int a, int b) {
    printf("Testing heuristic four...\n");
    if (a > b) {
        printf("%d > %d\n", a, b);
    } else {
        call_new_libcalls(a, b);
    }
    printf("Done.\n");
}
```

- Step 4: Rewrite the binary.
  - Razor compiles the instrumented instructions (i.e., *simple.s*) into an object file.
  - Razor rewrites the original binary with the object file.
  - The debloated binary is ***./simple_temp/simple.debloated***.

Let’s try the heuristics!



➢ Apply Heuristic **zCode**

```
🐳  83e097359693 ~/workspace/razor/heuristic-demo # ./apply_heuristic.sh 1
+ python ../stitcher/src/heuristic/disasm.py ./simple ./simple.asm
objdump -d -w --insn-width=16 ./simple > .tmp.asm
rm .tmp.asm
+ python ../stitcher/src/heuristic/find_more_paths.py ./simple.asm ./simple-trace.log ./simple-extended.log 1
reading trace and constructing cfg...
identifying jump tables...
```

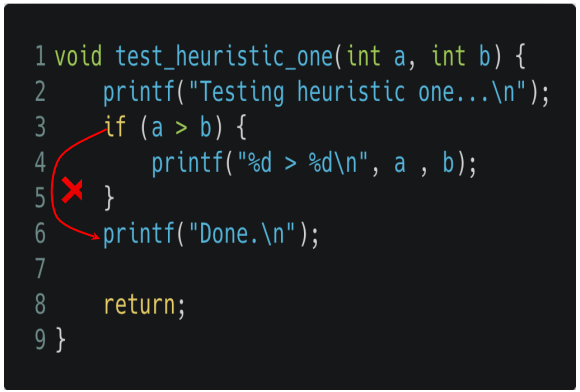
1 → zCode
2 → zCall
3 → zLib
4 → zFunc

➢ Running this command would:
- Use heuristic **zCode** to extend the original trace (i.e., simple-trace.log) and generate the extended trace. → simple-extended.log
- Dump the executed instructions. → instr.s
- Do the instrumentation. → simple.s
- Rewrite the binary. → simple_temp/simple.debloated

Run the Debloated Binary

```
🐋  83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Done.
Testing heuristic two...
Floating point exception
🐋  83e097359693 ~/workspace/razor/heuristic-demo #
```

➢ Applying heuristic **zCode** enables the control flow: line 3 → line 6 in ***test_heuristic_one***.



Run the Debloated Binary

```
🐋  83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Done.
Testing heuristic two...
Floating point exception
🐋  83e097359693 ~/workspace/razor/heuristic-demo #
```

➢ Applying heuristic **zCode** enables the control flow: line 3 → line 6 in ***test_heuristic_one***.

```
1 void test_heuristic_one(int a, int b) {
2     printf("Testing heuristic one...\n");
3     if (a > b) {
4         printf("%d > %d\n", a , b);
5     }
6     printf("Done.\n");
7
8     return;
9 }
```

zCode

```
1 void test_heuristic_one(int a, int b) {
2     printf("Testing heuristic one...\n");
3     if (a > b) {
4         printf("%d > %d\n", a , b);
5     }
6     printf("Done.\n");
7
8     return;
9 }
```

➢ Apply Heuristic **zCall**

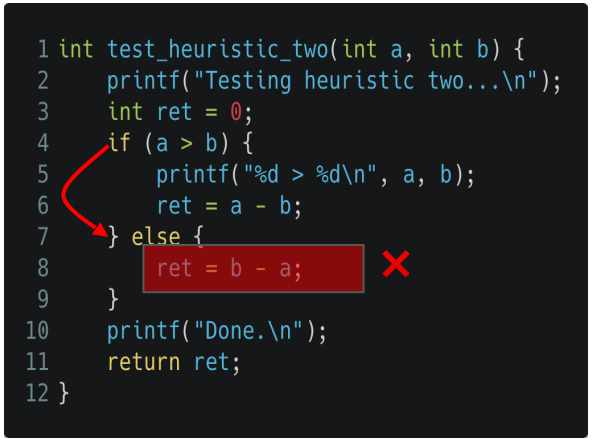
```
🐳  83e097359693 ~/workspace/razor/heuristic-demo # ./apply_heuristic.sh 2
+ python ../stitcher/src/heuristic/disasm.py ./simple ./simple.asm
objdump -d -w --insn-width=16 ./simple > .tmp.asm
rm .tmp.asm
+ python ../stitcher/src/heuristic/find_more_paths.py ./simple.asm ./simple-trace.log ./simple-extended.log 2
reading trace and constructing cfg...
identifying jump tables...
There are  0  jump tables
There are  0 offset tables
```

➢ Running this command would:
- Use heuristic **zCall** to extend the original trace (i.e., simple-trace.log) and generate the extended trace. → simple-extended.log
- Dump the executed instructions. → instr.s
- Do the instrumentation. → simple.s
- Rewrite the binary. → simple_temp/simple.debloated

Run the Debloated Binary

```
🐋  83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Done.
Testing heuristic two...
Done.
Testing herusitc three...
Floating point exception
```

- ➢ Besides enabling the control flow: line 3 → line 6 in ***test_heuristic_one***.
- ➢ Applying heuristic **zCall** adds the **false** branch in ***test_heuristic_two***.



Run the Debloated Binary

```
🐋  83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Done.
Testing heuristic two...
Done.
Testing herusitc three...
Floating point exception
```

- Besides enabling the control flow: line 3 → line 6 in ***test_heuristic_one***.
- Applying heuristic **zCall** adds the **false** branch in ***test_heuristic_two***.

```
1  int test_heuristic_two(int a, int b) {
2      printf("Testing heuristic two...\n");
3      int ret = 0;
4      if (a > b) {
5          printf("%d > %d\n", a, b);
6          ret = a - b;
7      } else {
8          ret = b - a;
9      }
10     printf("Done.\n");
11     return ret;
12 }
```

zCall

```
1  int test_heuristic_two(int a, int b) {
2      printf("Testing heuristic two...\n");
3      int ret = 0;
4      if (a > b) {
5          printf("%d > %d\n", a, b);
6          ret = a - b;
7      } else {
8          ret = b - a;
9      }
10     printf("Done.\n");
11     return ret;
12 }
```

➢ Apply Heuristic **zLib**

```
🐳  83e097359693 ~/workspace/razor/heuristic-demo # ./apply_heuristic.sh 3
+ python ../stitcher/src/heuristic/disasm.py ./simple ./simple.asm
objdump -d -w --insn-width=16 ./simple > .tmp.asm
rm .tmp.asm
+ python ../stitcher/src/heuristic/find_more_paths.py ./simple.asm ./simple-trace.log ./simple-extended.log 3
reading trace and constructing cfg...
identifying jump tables...
There are  0  jump tables
There are  0 offset tables
fixing jmp table targets...
initializing blocks
```

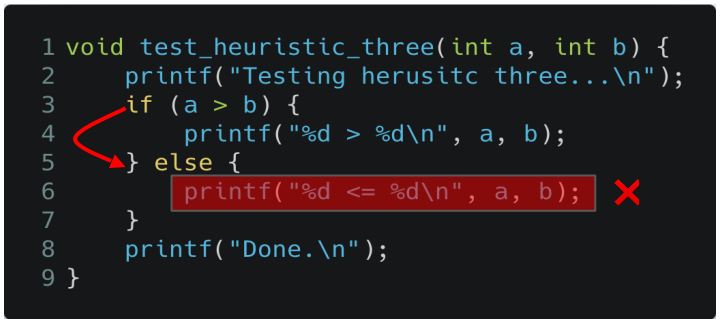
➢ Running this command would:

- Use heuristic **zLib** to extend the original trace (i.e., simple-trace.log) and generate the extended trace. → simple-extended.log
- Dump the executed instructions. → instr.s
- Do the instrumentation. → simple.s
- Rewrite the binary. → simple_temp/simple.debloated

Run the Debloated Binary

```
🐋  83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Done.
Testing heuristic two...
Done.
Testing herusitc three...
1 <= 2
Done.
Testing heuristic four...
Floating point exception
🐋  83e097359693 ~/workspace/razor/heuristic-demo #
```

- Besides **(1)** enableing the control flow: line 3 → line 6 in ***test_heuristic_one***. **(2)** adding the **false** branch in ***test_heuristic_two***.
- Applying heuristic **zLib** adds the **false** branch in ***test_heuristic_three***.



Run the Debloated Binary

```
🐋  83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Done.
Testing heuristic two...
Done.
Testing herusitc three...
1 <= 2
Done.
Testing heuristic four...
Floating point exception
🐋  83e097359693 ~/workspace/razor/heuristic-demo #
```

- Besides **(1)** enableing the control flow: line 3 → line 6 in ***test_heuristic_one***. **(2)** adding the **false** branch in ***test_heuristic_two***.
- Applying heuristic **zLib** adds the **false** branch in ***test_heuristic_three***.

```
1 void test_heuristic_three(int a, int b) {
2     printf("Testing herusitc three...\n");
3     if (a > b) {
4         printf("%d > %d\n", a, b);
5     } else {
6         printf("%d <= %d\n", a, b);
7     }
8     printf("Done.\n");
9 }
```

zLib

```
1 void test_heuristic_three(int a, int b) {
2     printf("Testing herusitc three...\n");
3     if (a > b) {
4         printf("%d > %d\n", a, b);
5     } else {
6         printf("%d <= %d\n", a, b);
7     }
8     printf("Done.\n");
9 }
```

➢ Apply Heuristic **zFunc**

```
🐳  83e097359693 ~/workspace/razor/heuristic-demo # ./apply_heuristic.sh 4
+ python ../stitcher/src/heuristic/disasm.py ./simple ./simple.asm
objdump -d -w --insn-width=16 ./simple > .tmp.asm
rm .tmp.asm
+ python ../stitcher/src/heuristic/find_more_paths.py ./simple.asm ./simple-trace.log ./simple-extended.log 4
reading trace and constructing cfg...
identifying jump tables...
There are  0  jump tables
There are  0 offset tables
fixing jmp table targets...
initializing blocks
```

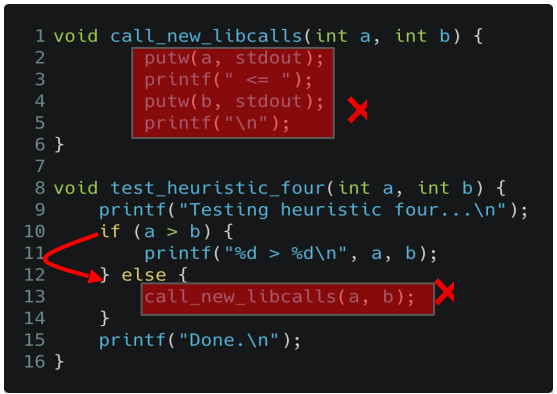
➢ Running this command would:

- Use heuristic **zFunc** to extend the original trace (i.e., simple-trace.log) and generate the extended trace. → simple-extended.log
- Dump the executed instructions. → instr.s
- Do the instrumentation. → simple.s
- Rewrite the binary. → simple_temp/simple.debloated

Run the Debloated Binary

```
83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Done.
Testing heuristic two...
Done.
Testing herusitc three...
1 <= 2
Done.
Testing heuristic four...
 <=
Done.
```

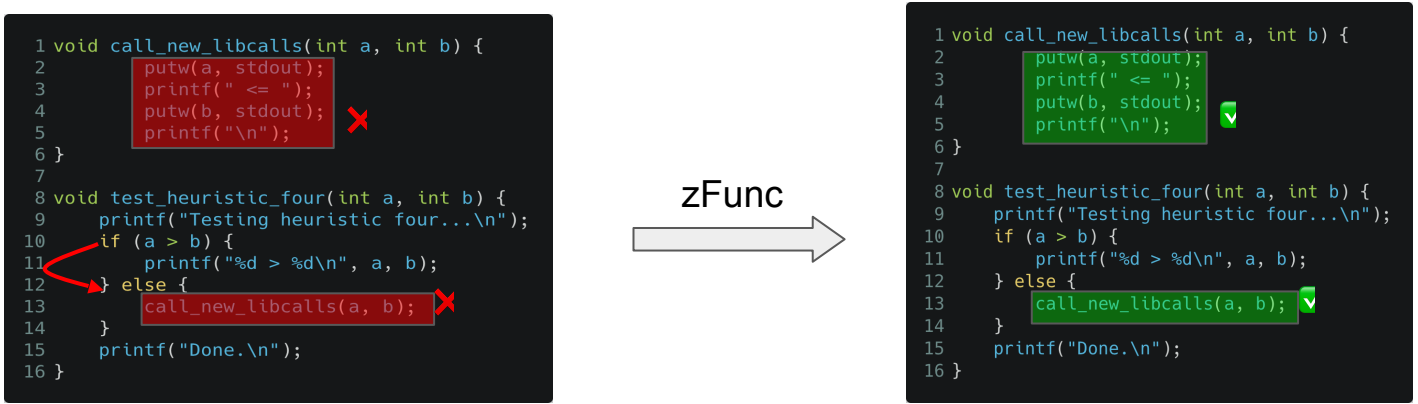
- Besides **(1)** enableing the control flow: line 3 → line 6 in ***test_heuristic_one***. **(2)** adding the **false** branch in ***test_heuristic_two***. **(3)** adding the **false** branch in ***test_heuristic_three***.
- Applying heuristic **zFunc** adds the **false** branch in ***test_heuristic_four***.



Run the Debloated Binary

```
🐋  83e097359693 ~/workspace/razor/heuristic-demo # ./simple_temp/simple.debloated 1 2
Testing heuristic one...
Done.
Testing heuristic two...
Done.
Testing herusitc three...
1 <= 2
Done.
Testing heuristic four...
 <=
Done.
```

- Besides **(1)** enableing the control flow: line 3 → line 6 in ***test_heuristic_one***. **(2)** adding the **false** branch in ***test_heuristic_two***. **(3)** adding the **false** branch in ***test_heuristic_three***.
- Applying heuristic **zFunc** adds the **false** branch in ***test_heuristic_four***.



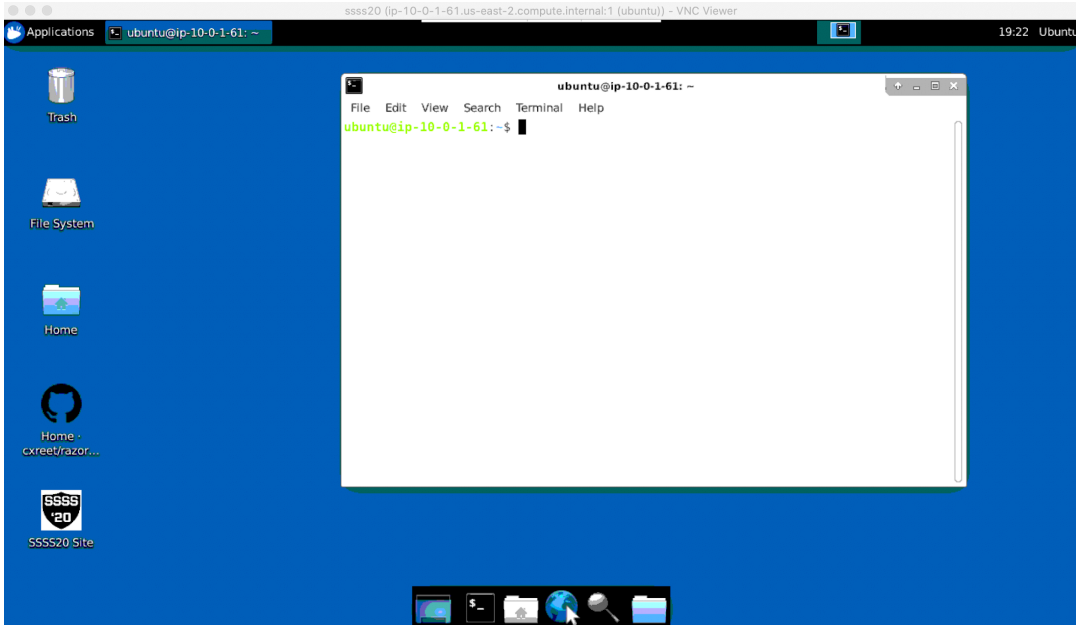
- Razor uses heuristics to infer code not covered by given inputs.
- The debloated binary supports inputs different with the inputs used for tracing but share the same functionality.

Exercises

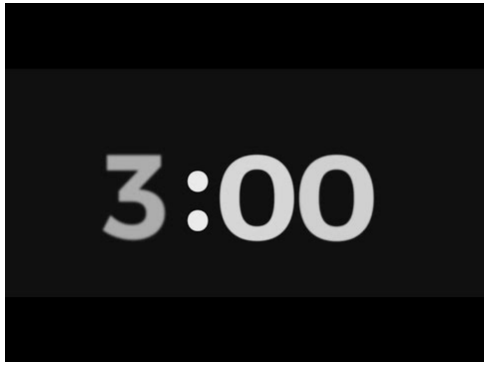
Online Doc: https://github.com/cxreet/razor/wiki/Exercises

Connect to the AWS Instance

```
summer-school-2020 $ ssh -L 5901:127.0.0.1:5901 -J inst1@ssss20.cerias.purdue.edu ubuntu@inst1
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 5.3.0-1030-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Mon Jul 13 15:46:52 UTC 2020

  System load:  0.0               Processes:              112
  Usage of /:   75.7% of 7.69GB   Users logged in:        0
  Memory usage: 4%                IP address for eth0:    10.0.1.61
  Swap usage:   0%                IP address for docker0: 172.17.0.1

 * "If you've been waiting for the perfect Kubernetes dev solution for
   macOS, the wait is over. Learn how to install Microk8s on macOS."

   https://www.techrepublic.com/article/how-to-install-microk8s-on-macos/

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

9 packages can be updated.
0 updates are security updates.


Last login: Mon Jul 13 15:34:00 2020 from 10.0.0.55
ubuntu@ip-10-0-1-61:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
```



Pull and Run the Docker Container



- Pull the container:
  - Run the command: ***sudo docker pull chenxiong/razor:0.01***
  - This takes around three minutes.
- Run the container:
  - Run the command: ***sudo docker run --rm -it chenxiong/razor:0.01***

```
workspace $ sudo docker pull chenxiong/razor:0.01
0.01: Pulling from chenxiong/razor
Digest: sha256:018b16122e04f218365a375fc0e8833bddfb832d5c5af86cf68a955e16342f23
Status: Image is up to date for chenxiong/razor:0.01
docker.io/chenxiong/razor:0.01
workspace $ sudo docker run  --rm -it chenxiong/razor:0.01
🐳  83e097359693 ~/workspace/razor #
```

If you see this dolphin, it means you are in the container!

Make sure that you see the dolphin and are under the correct directory.

- Under directory `/root/workspace/razor/benchmarks/core-utilities`, there are 10 programs.
  - Run the command: ***cd ~/workspace/razor/benchmarks/core-utilities/***
  - Run the command: ***ls***

```
🐳  c459a009f08e ~/workspace/razor # cd ~/workspace/razor/benchmarks/core-utilities/
🐳  c459a009f08e ~/workspace/razor/benchmarks/core-utilities # ls
README.md    chown-8.2  grep-2.19   mkdir-5.2.1  sort-8.16  uniq-8.16
bzip2-1.0.5  date-8.21  gzip-1.2.4  rm-8.4       tar-1.14
🐳  c459a009f08e ~/workspace/razor/benchmarks/core-utilities #
```

- Under each program's directory (i.e., **bzip2-1.0.5**):
  - Run the command: ***ls bzip2-1.0.5***

```
🐋  c459a009f08e ~/workspace/razor/benchmarks/core-utilities # ls bzip2-1.0.5/
bzip2-1.0.5.c.orig.c  bzip2.orig  run_razor.py  test  train
```

- Files under each program's directory:
  - bzip2-1.0.5.c.orig.c → the source code.
  - bzip2.orig → the binary.
  - run_razor.py → the helper script that takes commands:
    - **train**: run the binary with the training inputs and collect the traces.
    - **debloat**: debloat the binary with the traces.
    - **extend_debloat**: use heuristics to extend the traces and debloat the binary.
    - **test**: run the debloated binary with testing inputs.
    - **clean**: remove the outputs generated under current directory.
  - train → the directory that contains training inputs if the inputs are files.
  - test → the directory that contains testing inputs if the inputs are files.

- Trace the program with training inputs and get the traces:
  - Run the command: ***cd bzip2-1.0.5/***
  - Run the command: ***python run_razor.py train***

```
c459a009f08e ~/workspace/razor/benchmarks/core-utilities # cd bzip2-1.0.5/
c459a009f08e ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # python run_razor.py train
running  mkdir -p ./logs
running  cp ../../../tracers/bin/libcbr_indcall.so ./logs/
../../../tracers/dynamorio/bin64/drrun -c ./logs/libcbr_indcall.so -- ./bzip2.orig -c < train/bible.txt > tmp.log
running  ../../../tracers/dynamorio/bin64/drrun -c ./logs/libcbr_indcall.so -- ./bzip2.orig -c < train/bible.txt > tmp.log
../../../tracers/dynamorio/bin64/drrun -c ./logs/libcbr_indcall.so -- ./bzip2.orig -c < train/bib > tmp.log
running  ../../../tracers/dynamorio/bin64/drrun -c ./logs/libcbr_indcall.so -- ./bzip2.orig -c < train/bib > tmp.log
../../../tracers/dynamorio/bin64/drrun -c ./logs/libcbr_indcall.so -- ./bzip2.orig -c < train/obj1 > tmp.log
running  ../../../tracers/dynamorio/bin64/drrun -c ./logs/libcbr_indcall.so -- ./bzip2.orig -c < train/obj1 > tmp.log
../../../tracers/dynamorio/bin64/drrun -c ./logs/libcbr_indcall.so -- ./bzip2.orig -c < train/book1 > tmp.log
```

- The traces would be put under directory *logs*.
  - Run the command: ***ls logs***

```
7bf013a271d6 ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # ls logs/
cbr_indcall.bzip2.orig.00043.0000.log  cbr_indcall.bzip2.orig.00049.0000.log  cbr_indcall.bzip2.orig.00055.0000.log
cbr_indcall.bzip2.orig.00045.0000.log  cbr_indcall.bzip2.orig.00051.0000.log  cbr_indcall.bzip2.orig.00057.0000.log
cbr_indcall.bzip2.orig.00047.0000.log  cbr_indcall.bzip2.orig.00053.0000.log  cbr_indcall.bzip2.orig.00059.0000.log
```

➢ Debloat the binary:
  - Run the command: ***python run_razor.py debloat***

```
7bf013a271d6 ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # python run_razor.py debloat
```

➢ To check the outputs:
  - Run the command: ***ls***
  - Run the command: ***ls bzip2.orig_temp/***

```
020734bd5b0b ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # ls
bzip2-1.0.5.c.orig.c  bzip2-trace.log  bzip2.orig  bzip2.orig_temp  bzip2.s  callbacks.txt  instr.s  logs  run_razor.py  test  tmp.log  train
020734bd5b0b ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # ls bzip2.orig_temp/
bzip2.o  bzip2.orig  bzip2.orig.debloated  bzip2.s  new_obj_text_content
020734bd5b0b ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 #
```

- ***bzip2-trace.log*** → the merged logs.
- ***instr.s*** → the executed instructions in assembly code format.
- ***bzip2.s*** → the instrumented assembly code.
- ***bzip2.orig_temp/bzip2.orig.debloated*** → the debloated binary.

➢ Run the debloated binary with testing inputs:
  - Run the command: ***python run_razor.py test***

```
020734bd5b0b ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # python run_razor.py test
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/plrabn12.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/paper6 > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/E.coli > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/asyoulik.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/obj2.bz2 > tmp.log
Floating point exception
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/progc > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/a.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/pi.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/geo > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/ptt5 > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/book2 > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/bible.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/random.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/grammar.lsp > tmp.log
Floating point exception
```

➔ For some testing inputs, the debloated binary failed with **Floating point exception**.

➢ Extend the original traces using heuristic **zCall** and debloat the binary:
  - Run the command: ***python run_razor.py extend_debloat 2***

```
020734bd5b0b ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # python run_razor.py extend_debloat 2
running  python ../../../stitcher/src/heuristic/disasm.py ./bzip2.orig ./bzip2.orig.asm
objdump -d -w --insn-width=16 ./bzip2.orig > .tmp.asm
rm .tmp.asm
running  python ../../../stitcher/src/heuristic/find_more_paths.py ./bzip2.orig.asm ./bzip2-trace.log ./bzip2-extended.log 2
reading trace and constructing cfg...
identifying jump tables...
There are  0  jump tables
There are  0 offset tables
fixing jmp table targets...
initializing blocks
```

It can be 1, 2, 3, or 4.

- The argument of **extend_debloat**:
  - 1 → **zCode**
  - 2 → **zCall**
  - 3 → **zLib**
  - 4 → **zFunc**

- Run the debloated binary with testing inputs again:
  - Run the command: ***python run_razor.py test***

```
020734bd5b0b ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # python run_razor.py test
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/plrabn12.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/paper6 > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/E.coli > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/asyoulik.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/obj2.bz2 > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/progc > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/a.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/pi.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/geo > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/ptt5 > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/book2 > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/bible.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/random.txt > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/grammar.lsp > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/xargs.1 > tmp.log
running  ./bzip2.orig_temp/bzip2.orig.debloated -c < test/cp.html > tmp.log
```

- Check the code size of the original binary and the debloated binary:
  - Run the command: ***python ../../../tools/get_code_size.py ./bzip2.orig***
  - Run the command: ***python ../../../tools/get_code_size.py ./bzip2.orig_temp/bzip2.orig.debloated***

```
020734bd5b0b ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # python ../../../tools/get_code_size.py ./bzip2.orig
LOAD           0x000000 0x0000000000400000 0x0000000000400000 0x01c86c 0x01c86c R E 0x200000
size: 116844
020734bd5b0b ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 # python ../../../tools/get_code_size.py ./bzip2.orig_temp/bzip2.orig.debloated
LOAD           0x036000 0x0000000000700000 0x0000000000700000 0x00b6f2 0x00b6f2 R E 0x1000
size: 46834
020734bd5b0b ~/workspace/razor/benchmarks/core-utilities/bzip2-1.0.5 #
```

- The code size is reduced from 116,844 bytes to 46,834 bytes with the heuristic **zCall**.
  - The code reduction rate (CRR) is:
    - (116844-46834)/116844 = 59.9%.

Exercise #1

- For the 10 programs under directory “/root/workspace/razor/benchmarks/core-utilities”:
  - First, run Razor to debloat the binary without any heuristics. Calculate the code reduction rate (CRR) and the number of failed cases of testing inputs.
  - Then, run Razor to debloat the binary with applying heuristic **zCode**, **zCall**, **zLib**, **zFunc**. Calculate the CRR and the number of failed cases of testing inputs for each heuristic. (Stop at any heuristic if the debloated binary failed zero testing inputs.)
  - Fill the table on the next page.
  - Please following the instructions: https://github.com/cxreet/razor/wiki/Exercises

	None		zCode		zCall		zLib		zFun	
	CRR (%)	# Failed Cases	CRR (%)	# Failed Cases	CRR (%)	# Failed Cases	CRR (%)	# Failed Cases	CRR (%)	# Failed Cases
bzip2										
chown										
date										
grep										
gzip										
mkdir										
rm										
sort										
tar										
uniq										

CRR: code reduction rate.

Exercise #2

- For programs that still got `Floating point exception` after applying the ***zFunc*** heuristic.
  - Why do the heuristics fail?
    - Hint: The output from running command `***extend_debloat 4***` has *"**Missing lib functions**"*.
  - How to solve the exceptions?
    - Hint: `***/root/workspace/razor/stitcher/src/heuristic/libcall.patch***` is used for heuristics to add library calls that share different functionalities with executed code. Each line contains a function name.

Solutions for Exercises

Exercise #1

	None		zCode		zCall		zLib		zFun	
	CRR (%)	# Failed Cases	CRR (%)	# Failed Cases	CRR (%)	# Failed Cases	CRR (%)	# Failed Cases	CRR (%)	# Failed Cases
bzip2	62.4	6	62.4	5	59.9	1	24.8	0		
chown	81.9	17	82.1	17	79.0	17	56.7	10	56.7	10
date	71.2	33	71.4	33	59.8	33	52.8	2	43.4	0
grep	75.3	38	75.4	38	71.1	38	58.3	0		
gzip	79.6	3	79.6	3	76.1	2	75.2	0		
mkdir	75.4	2	75.5	2	67.8	1	37.0	0		
rm	76.5	20	76.7	20	70.7	20	51.7	5	44.9	5
sort	89.2	28	89.3	28	87.4	28	50.5	0		
tar	95.0	1	95.0	1	92.9	0				
uniq	84.2	40	84.3	40	83.2	40	53.6	0		

Exercise #2