RE-ALign

AHAMAN ULLAH Efaz
EURECOM
Efaz.Ahaman-Ullah@eurecom.fr

Abstract

Modern binary tools often rely on disassembler to transform raw binary data into assembly code. Linear sweep algorithm is one of the most used technique in this case. However, it is vulnerable to misalignment. Through this project, our goal was to conduct an experiment on 64-bit and 32 bit benign and non-obfuscated binaries by choosing randomly an offset as entry point and to see after how many instructions there will be a realignment or an invalid output.

1 Introduction

A disassembler is a computer program that translates machine language (raw data) into assembly language. Two main techniques used in this case are : Linear sweep algorithm (LSA) and Recursive traversal (RT). In LSA the disassembler disassembles the code sequentially and assumes that all code in the region is valid and in-order instructions. Whereas, RT follows control-flow to decode only reachable instructions. Other techniques like Hybrid disassembly and Dynamic disassembly exists too.

LSA is fast but error-prone if alignment is lost. The misalignment often occurs in obfuscated, malicious assembly code. Understanding disassembler reliability is critical in reverse engineering, debugging and static analysis.

When a disassembler starts from an incorrect entry point how long does it take to realign, if ever?

I tried to build a framework that naively performs LSA over a binary from a randomly chosen offset and then we comapare the output with the ground truth assembly code produced by Ghidra. Then I checked whether there is a realignment or invalid outputs.

This is an empirical study of how LSA behaves with respect to instruction alignment in a diverse set of binaries. The framework allows us to control the perturbation of instruction alignment and to compare it with the ground truth.

2 Code

The code is well commented.

3 Experiments

The experiment consisted on doing the followings :

1. Given a benign, non-obfuscated, program binary, produce a correct disassembly (e.g., by using IDA, Ghidra..) and save the result (offset -> instruction).

2. Then, run a linear sweep disassembly on the same code, starting at random position (which are not the offset of a correct instruction) and produce a disassembly.
3. Compare the output with the ground truth obtained in step 1 and produce as output the number of instruction before an illegal instruction or a re-alignment.

Experiment:

1. Repeat point 3 above for many offset inside the binary and aggregate the results (e.g., how many times we got an illegal and how many time we got a re-alignment and a distribution graph of the number of instructions in both cases).
2. Repeat for 32 and 64 bit.
3. Repeat on many 68 diverse applications (If all versions were counted, there would be 92 different binaries)
4. Compile the application by ourselves and compute the results based on optimization level (O0-O3 and Os).

I used an x86 architecture PC. And for the results that will be shown in this report the iteration number is 1000.

I tried to use a lot of diverse applications : math functions, compression, core utils, memory operations etc.

4 Results

In this part, I tried to show the plots of some of the binaries. I tried to show the interesting ones sometimes and ignored the binaries that behaves in the same manner. But you can find the plots for all of them in "result_dir". I also ignored the intermediate optimaization versions 1, 2 and fast in the upcoming part, but you can find the results in "result_dir". For some of the binaries some I also tried iteration 5000 and 10000 (you can find them in "result_dir"). The result doesn't change that much : the CDF curve becomes more smooth and the stop type distribution changes a little bit.

I just want to mention that all the PE binaries are collected from microsoft's official website, and for the ELF binaries they are from the "/bin" directoy of linux. And if you open the "journal.txt", you can see that some files are from "/home/swisscat/Downloads/..." these files are PE files. And the which comes from this directory "/home/swisscat/.cargo/bin" are rustc compiled files (there are only few ones).

4.1 Optimization 32bit vs 64 bit

4.1.1 pcalc 32 bit with optimization

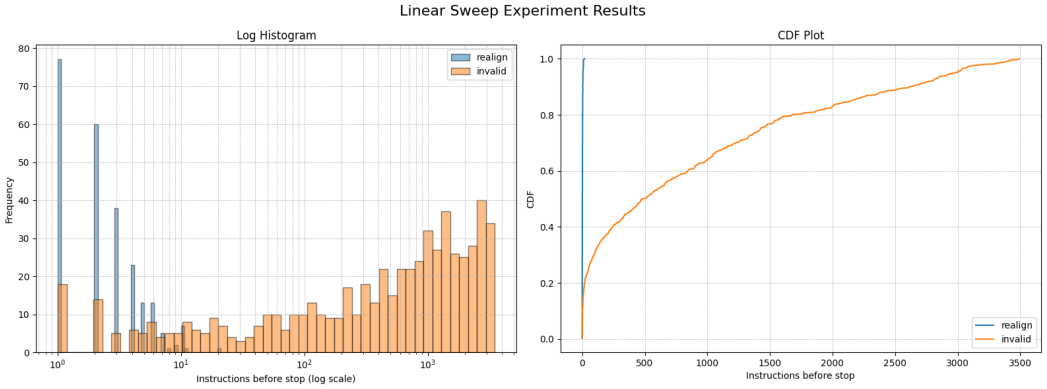

Figure 1: pcalc 32 optim 0

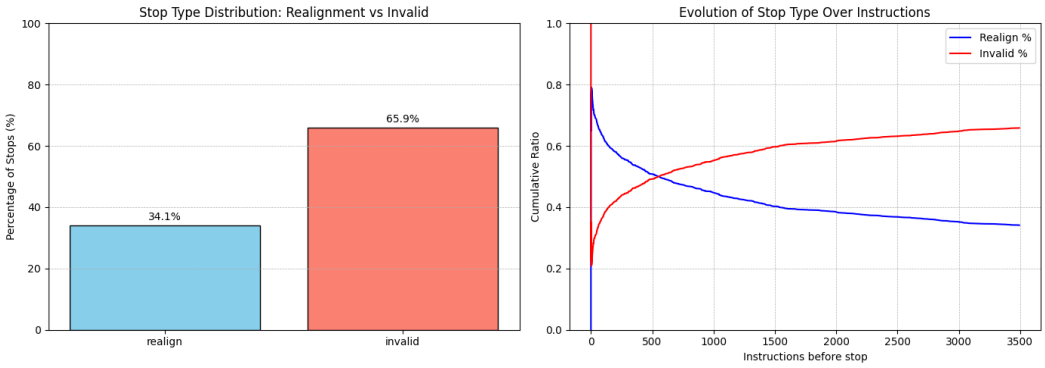

Figure 2: pcalc 32 optim 0

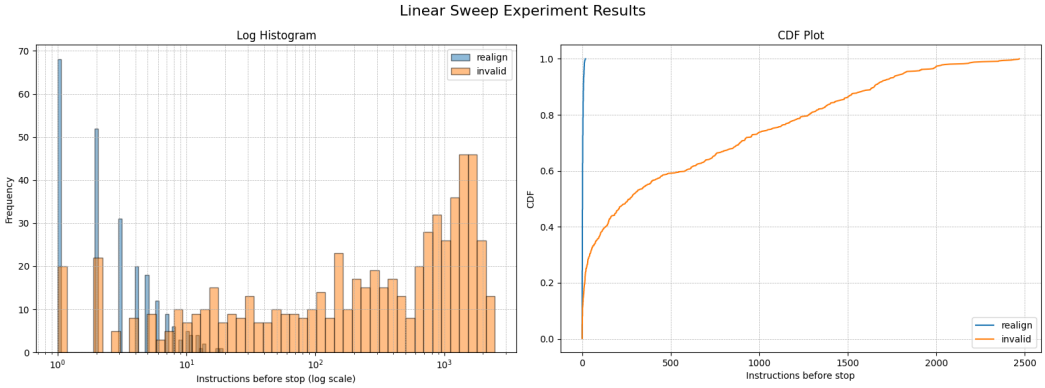

Figure 3: pcalc 32 optim 3

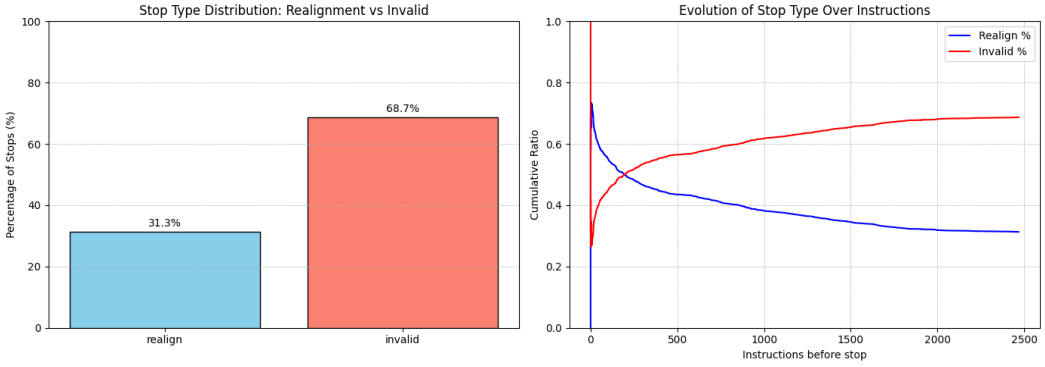

Figure 4: pcalc 32 optim 3

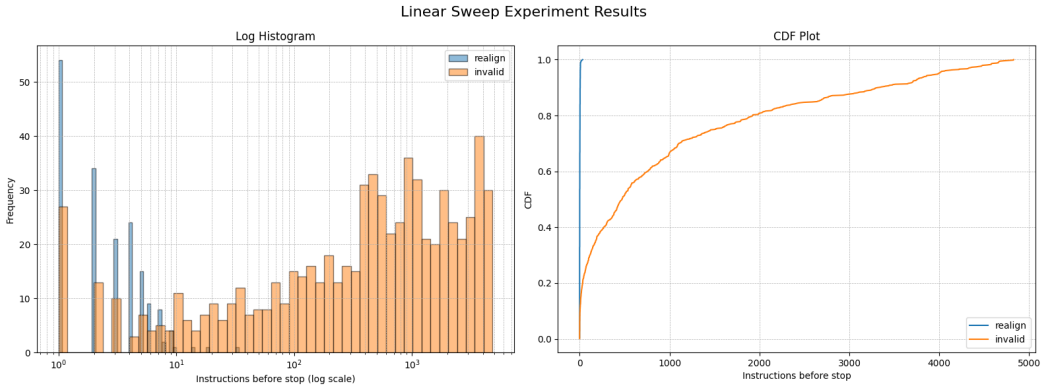

Figure 5: pcalc 32 optim s

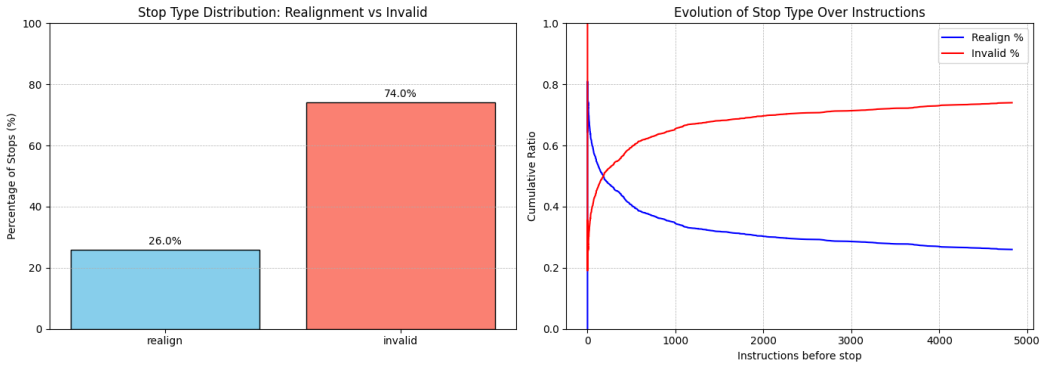

Figure 6: pcalc 32 optim s

4.1.2 pcalc 64 bit with optimization

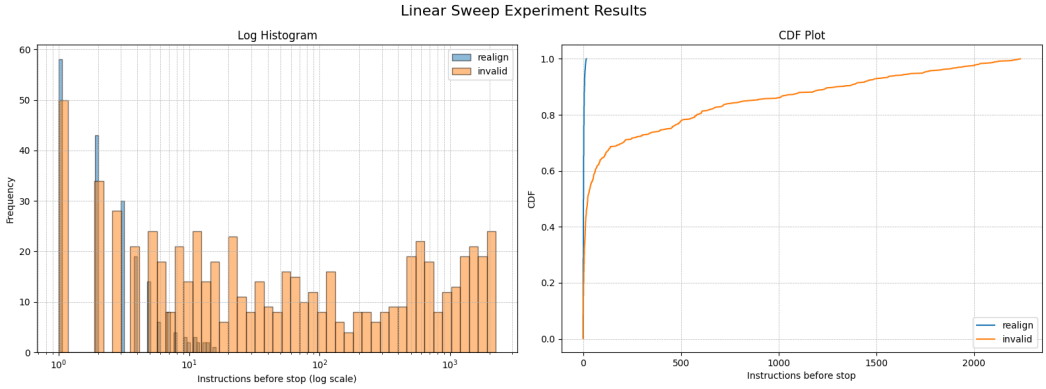

Figure 7: pcalc 64 optim 0

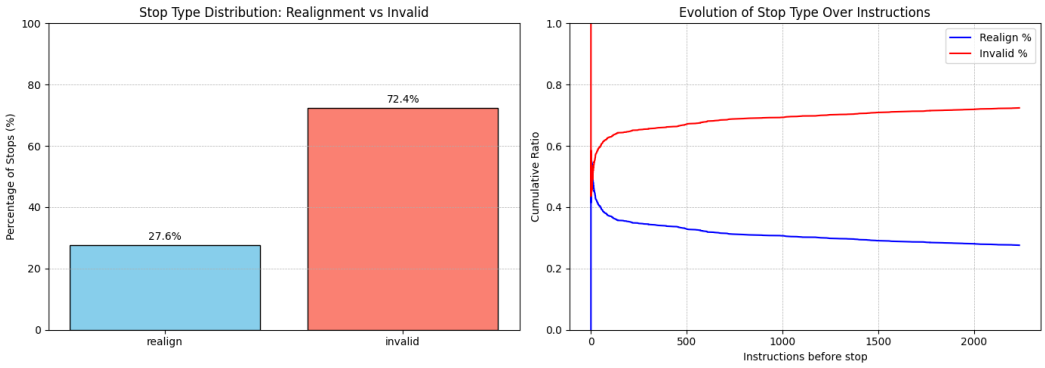

Figure 8: pcalc 64 optim 0

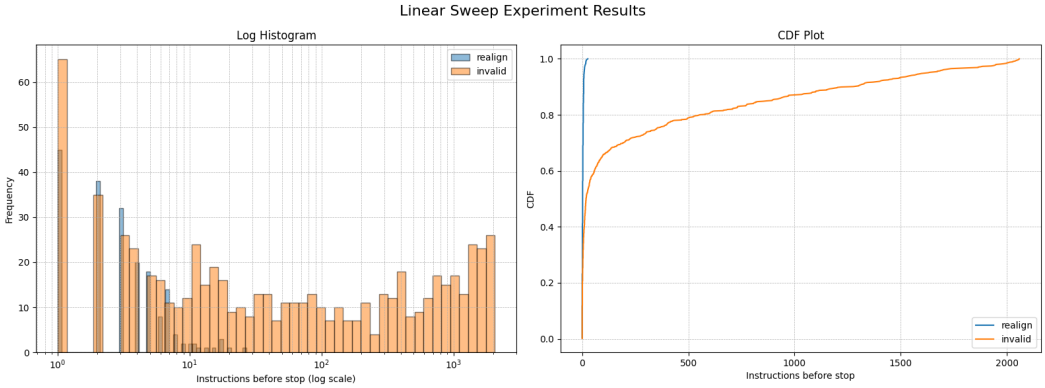

Figure 9: pcalc 64 optim 3

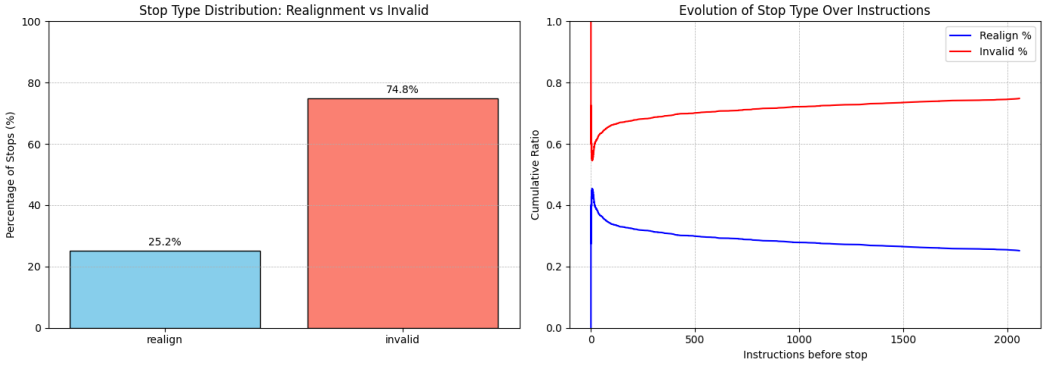

Figure 10: pcalc 64 optim 3

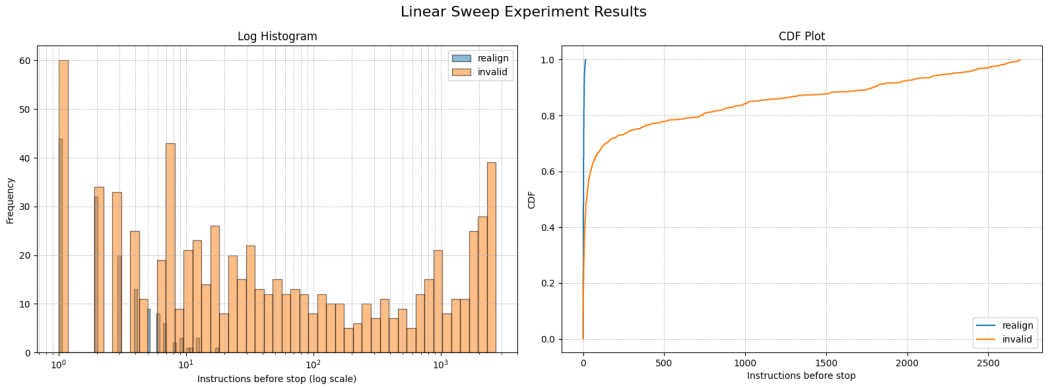

Figure 11: pcalc 64 optim s

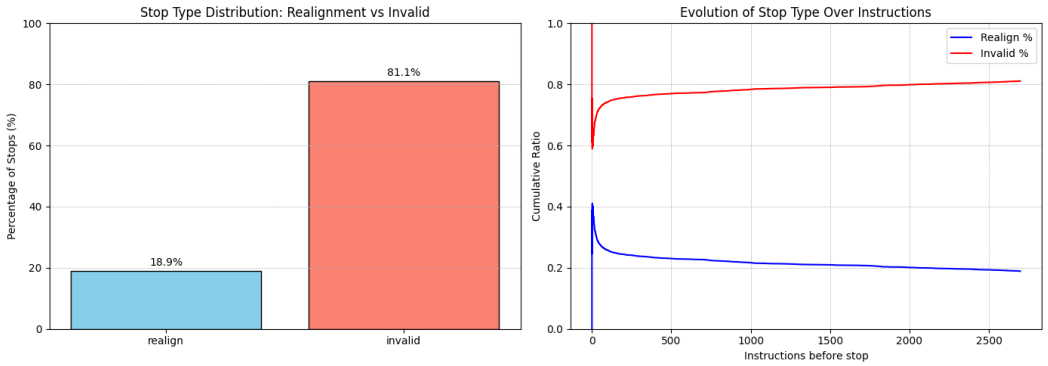

Figure 12: pcalc 64 optim s

4.1.3 8cc 32 bit with optimization

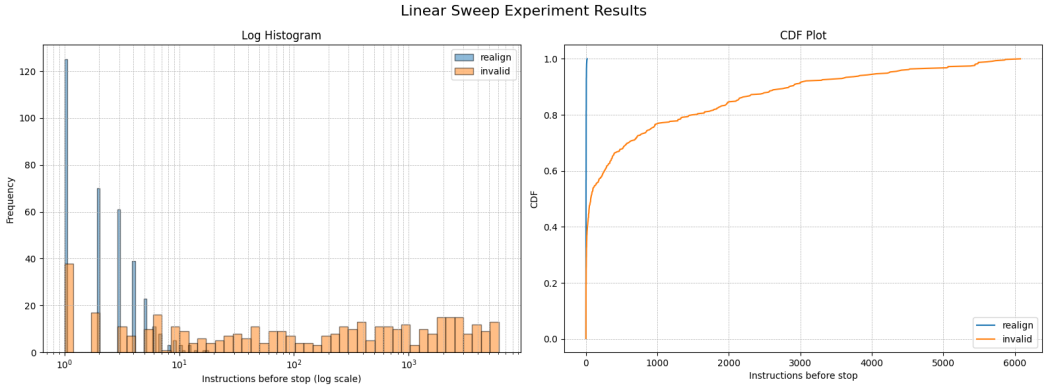

Figure 13: 8cc 32 optim 0

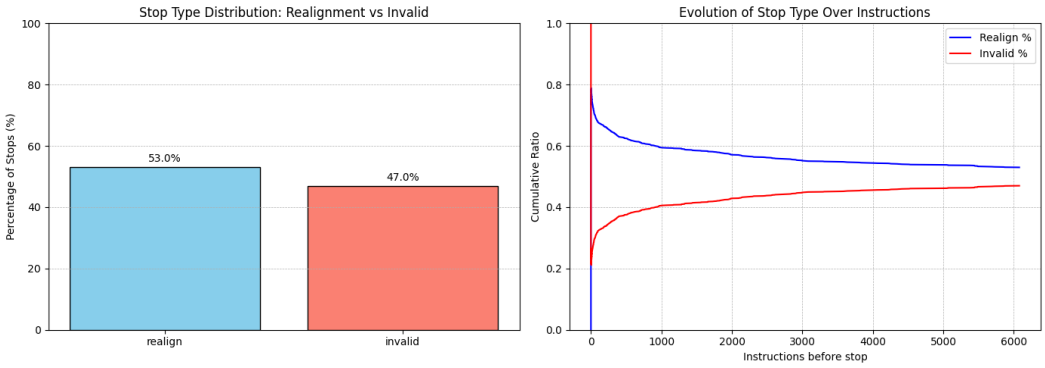

Figure 14: 8cc 32 optim 0

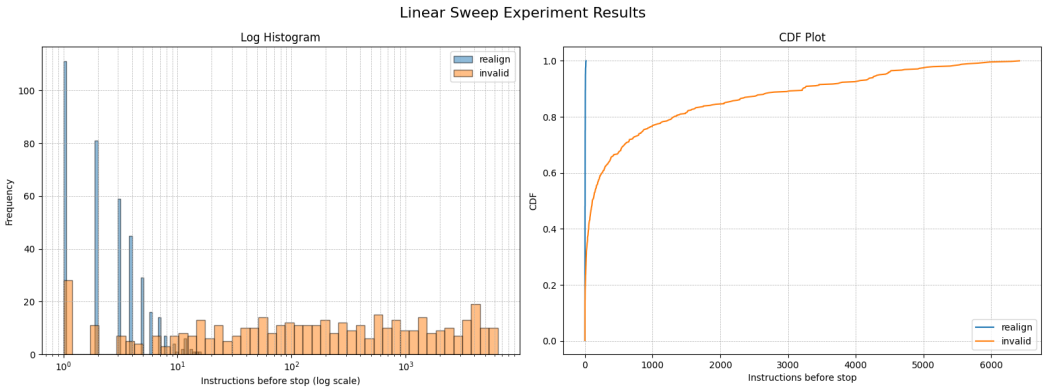

Figure 15: 8cc 32 optim 3

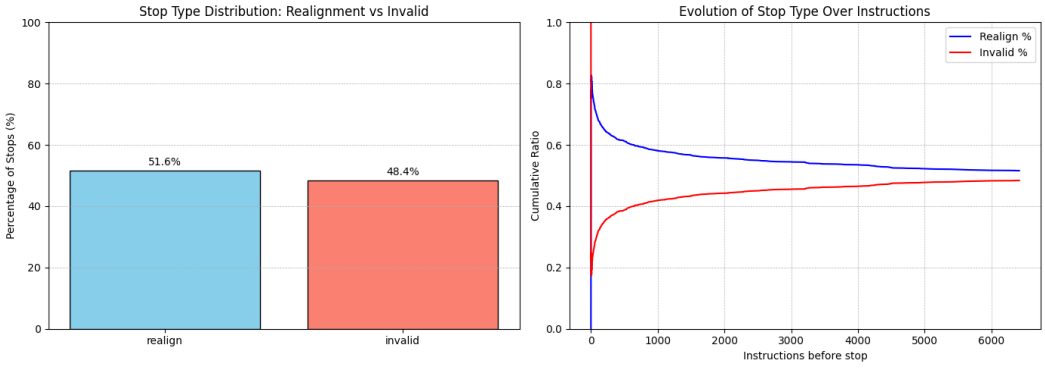

Figure 16: 8cc 32 optim 3

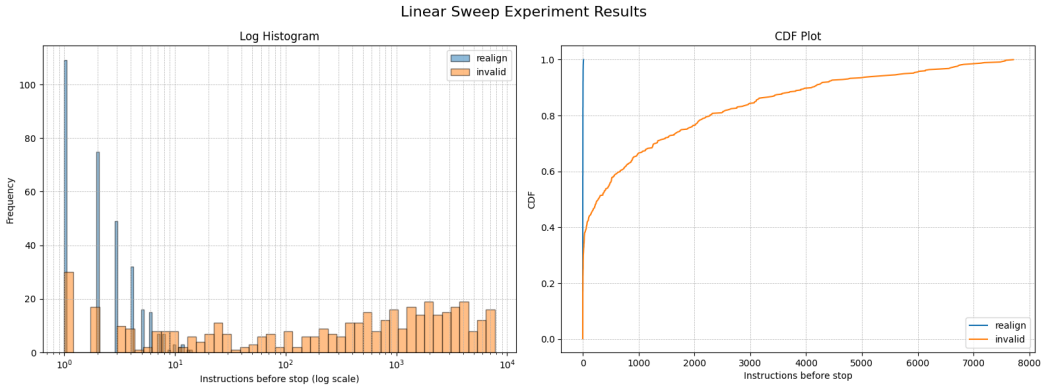

Figure 17: 8cc 32 optim s

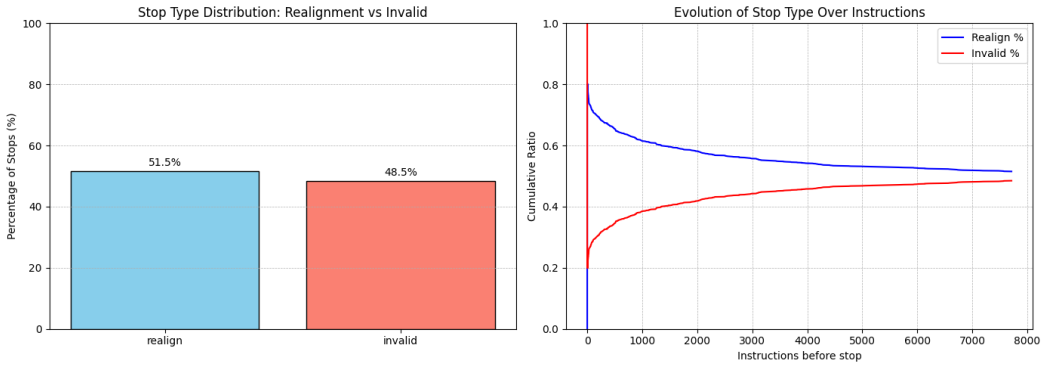

Figure 18: 8cc 32 optim s

4.1.4 8cc 64 bit with optimization

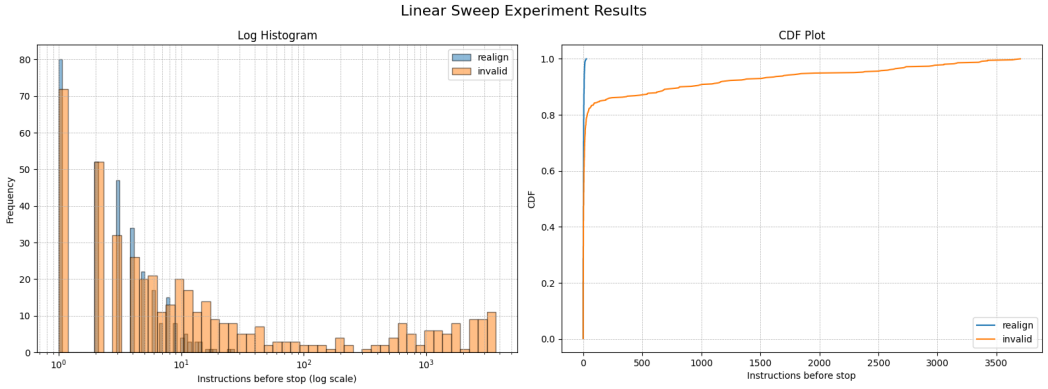

Figure 19: 8cc 64 optim 0

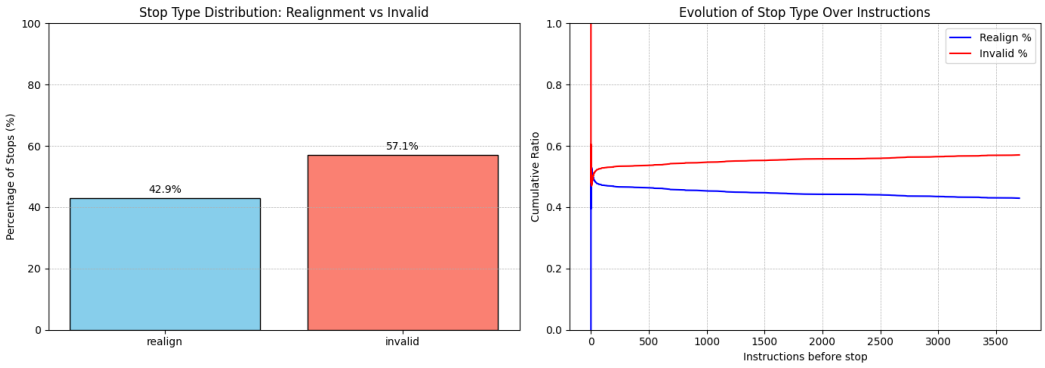

Figure 20: 8cc 64 optim 0

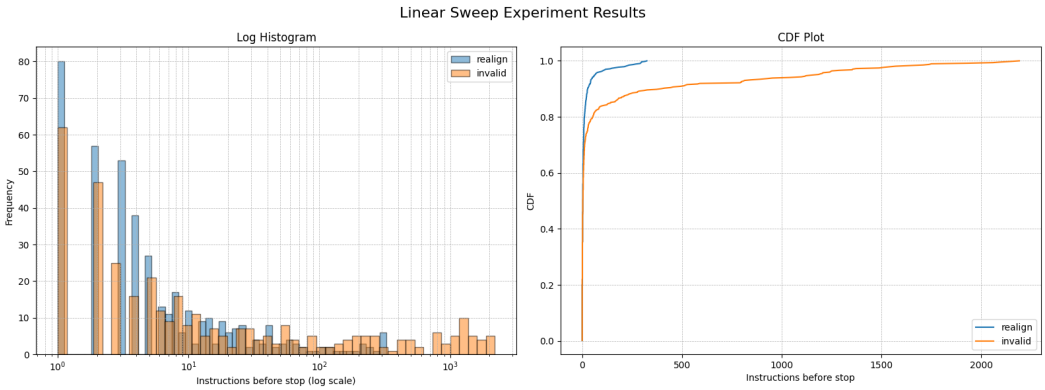

Figure 21: 8cc 64 optim 3

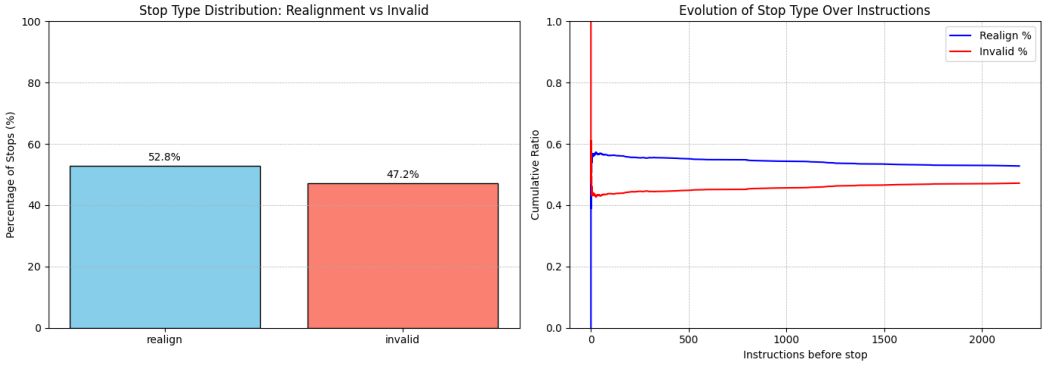

Figure 22: 8cc 64 optim 3

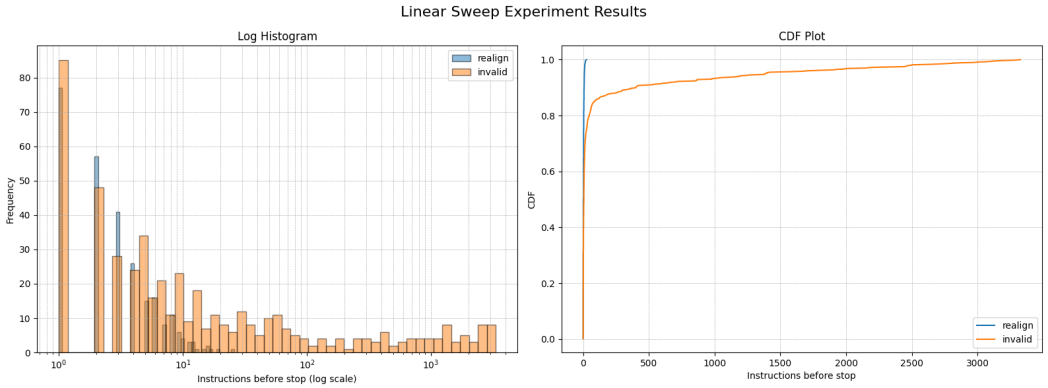

Figure 23: 8cc 64 optim s

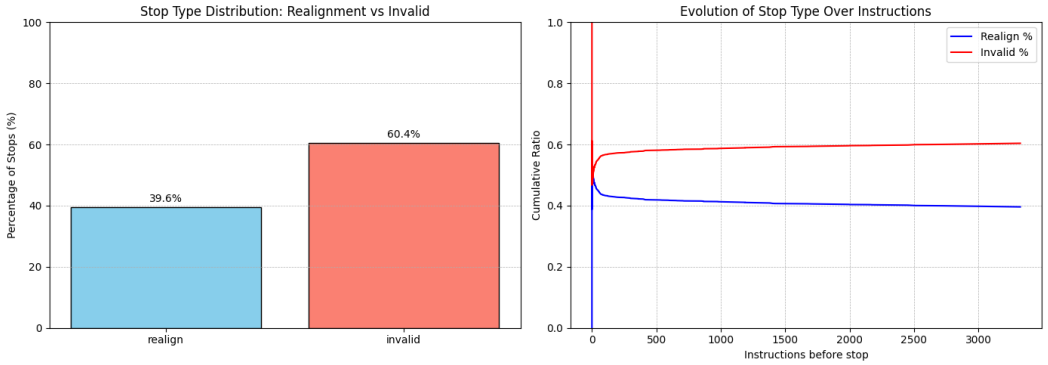

Figure 24: 8cc 64 optim s

4.2 PE binary 32bit vs 64bit

4.2.1 Cacheset

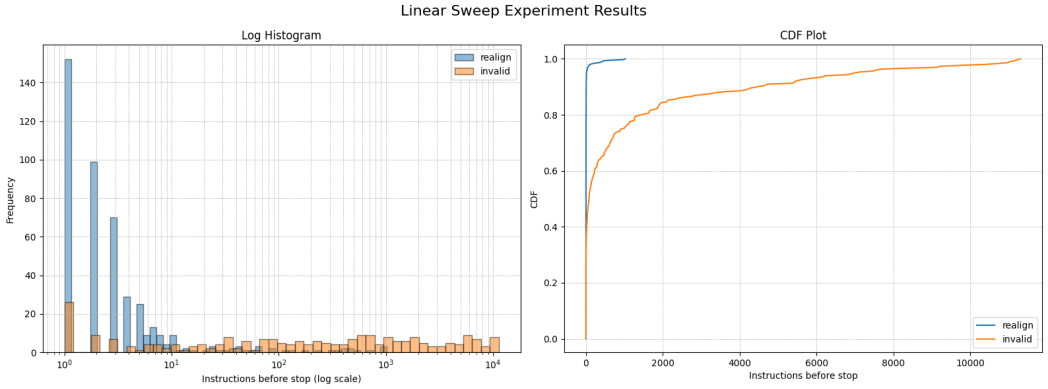

Figure 25: Cacheset 32

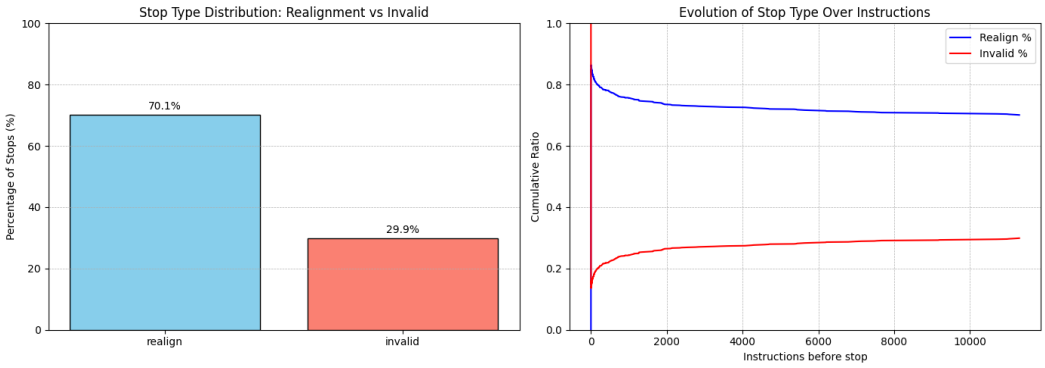

Figure 26: Cacheset 32

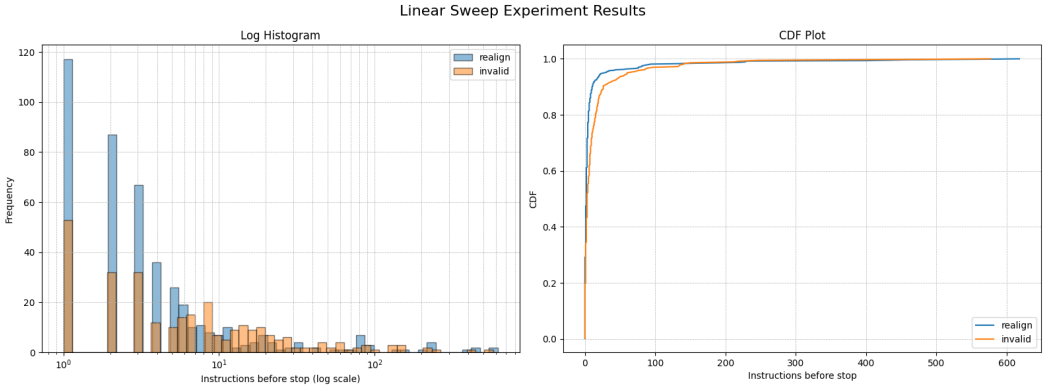

Figure 27: Cacheset 64

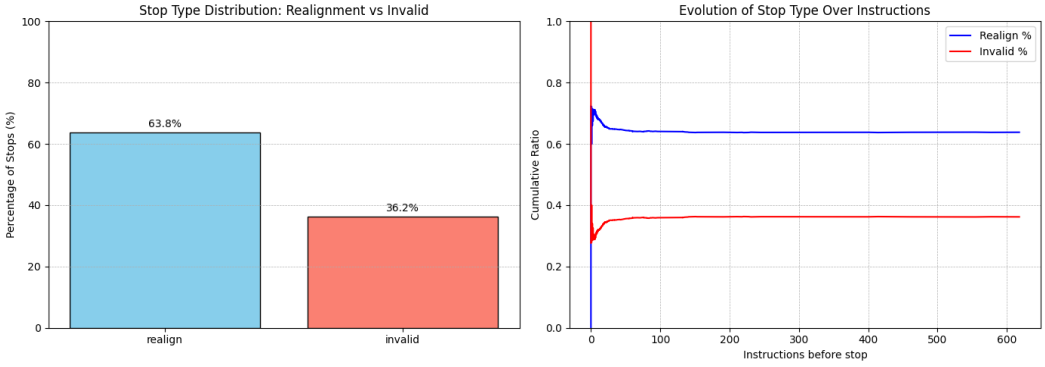

Figure 28: Cacheset 64

4.2.2 Clockres

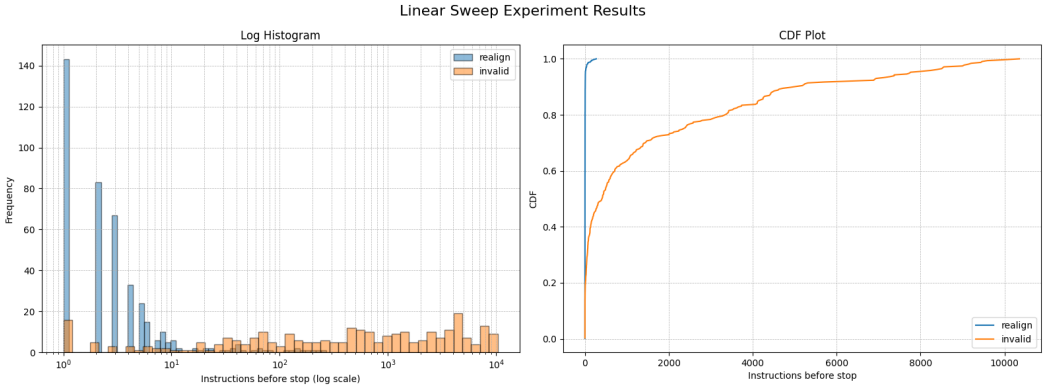

Figure 29: Clockres 32

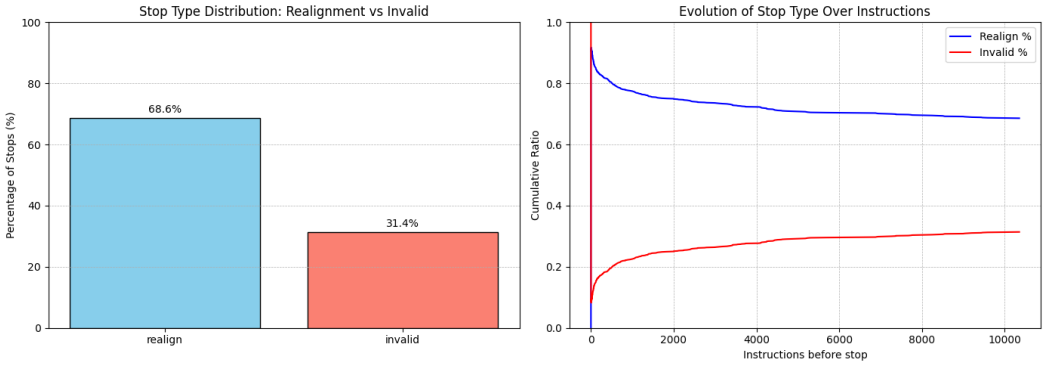

Figure 30: Clockres 32

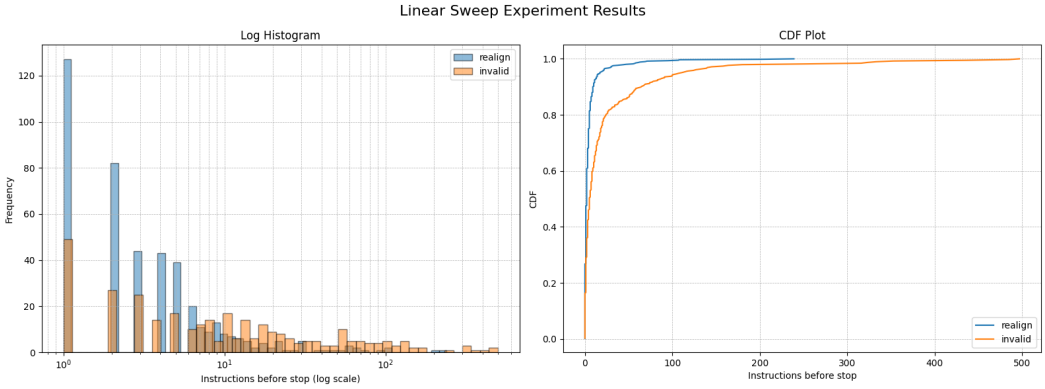

Figure 31: Clockres 64

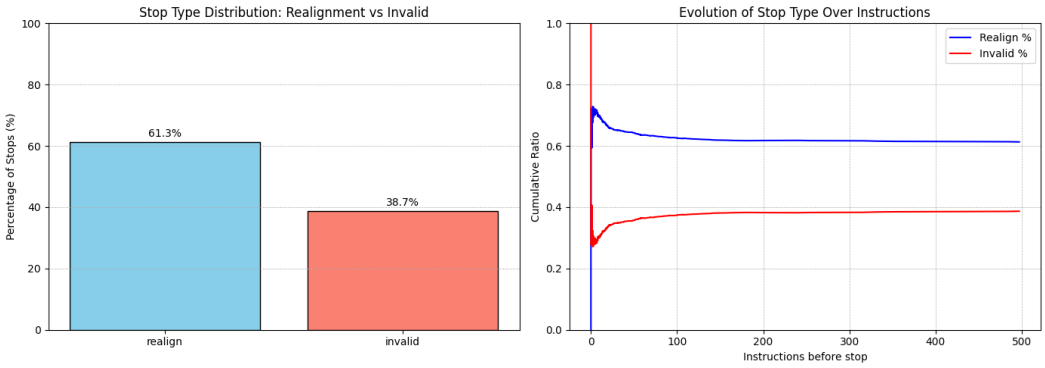

Figure 32: Clockres 64

4.3 ELF binary

4.3.1 Plocate

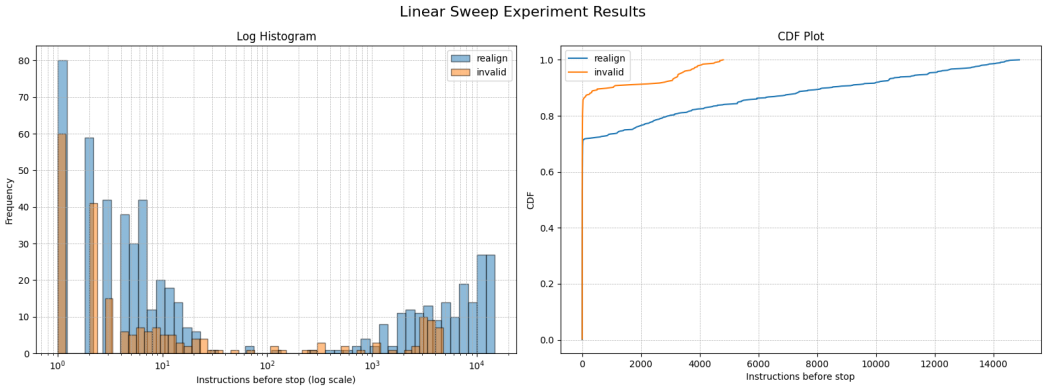

Figure 33: plocate 64

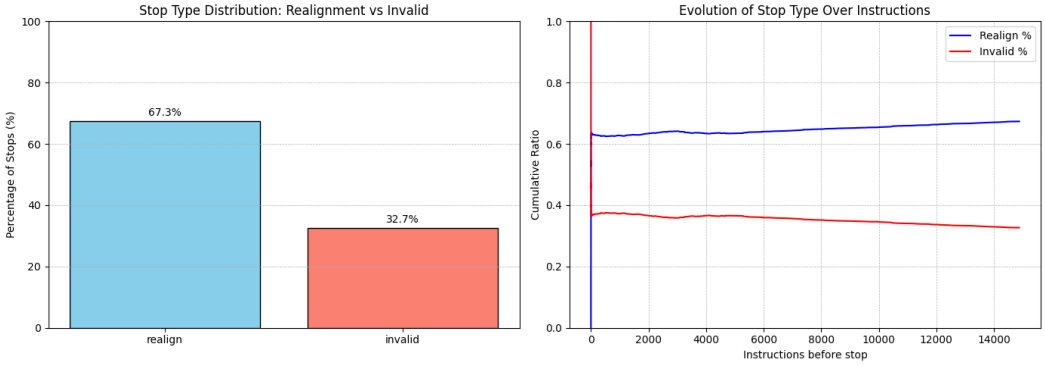

Figure 34: plocate 64

4.3.2 TCPdump

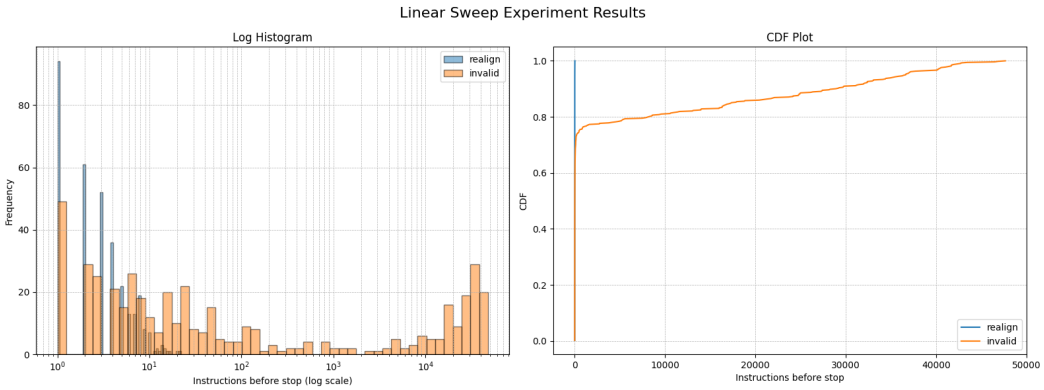

Figure 35: tcpdump 64

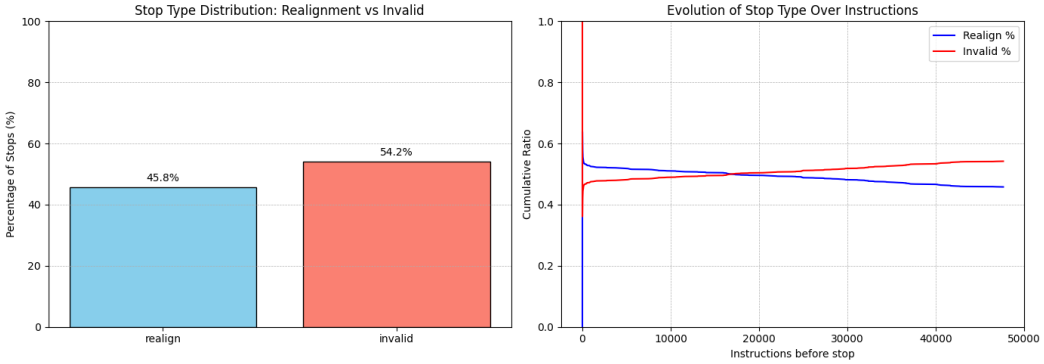

Figure 36: tcpdump 64

4.3.3 ffmpeg

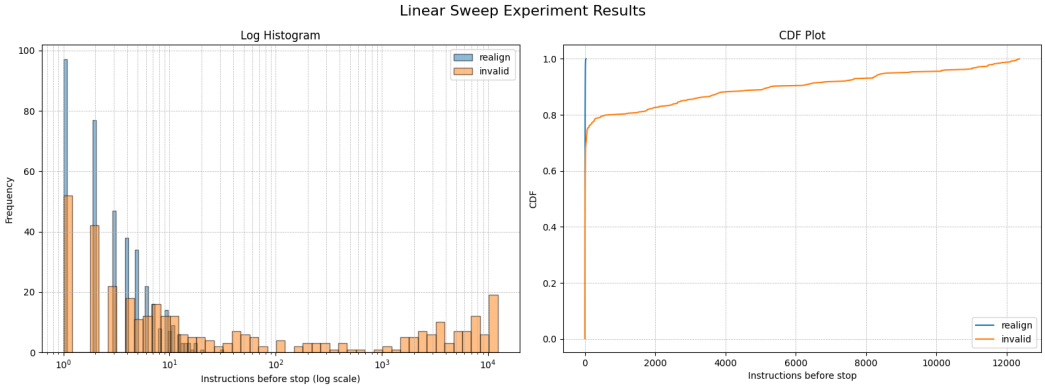

Figure 37: ffmpeg 64

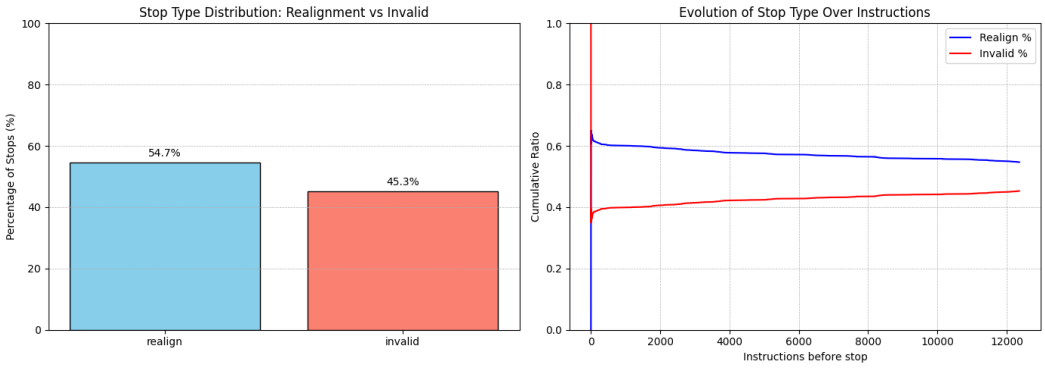

Figure 38: ffmpeg 64

4.4 Compilers

4.4.1 Rustc

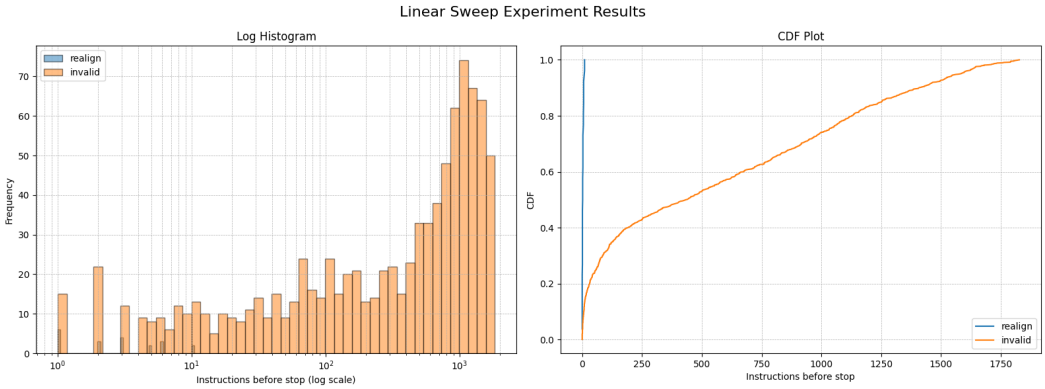

Figure 39: rustc 64

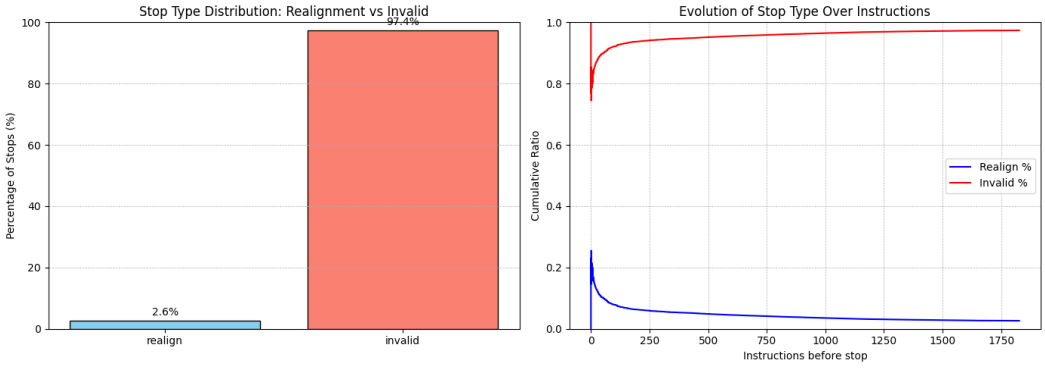

Figure 40: rustc 64

4.4.2 Gcc

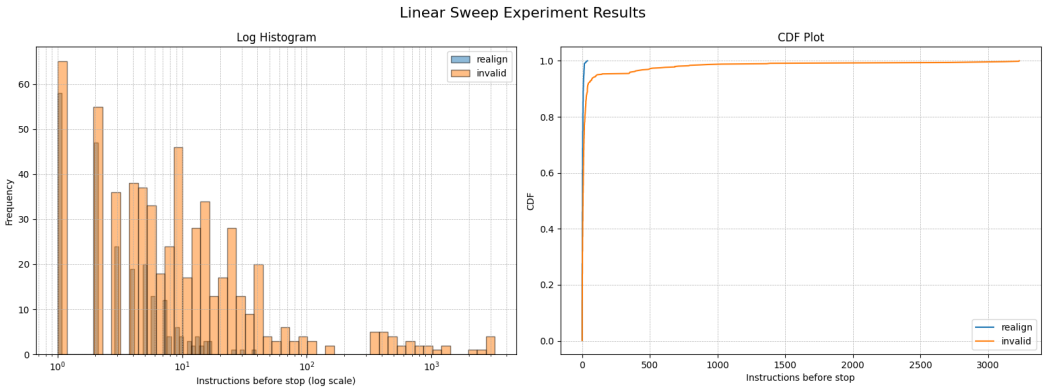

Figure 41: gcc 64

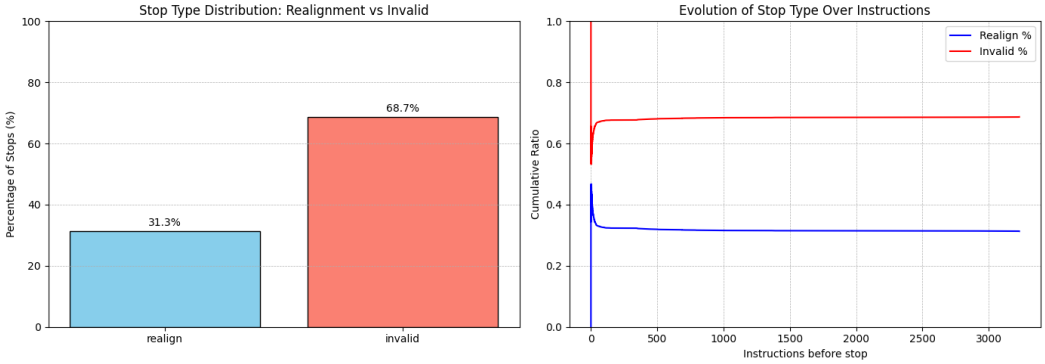

Figure 42: gcc 64

4.4.3 fpc (Pascal)

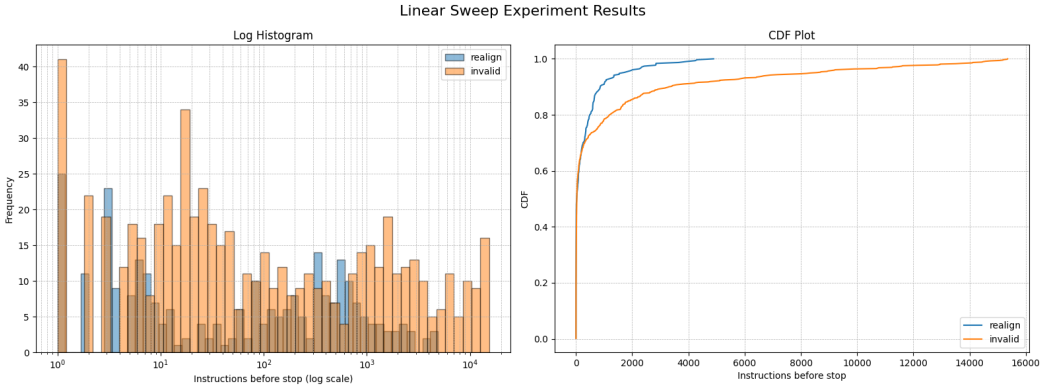

Figure 43: fpc 64

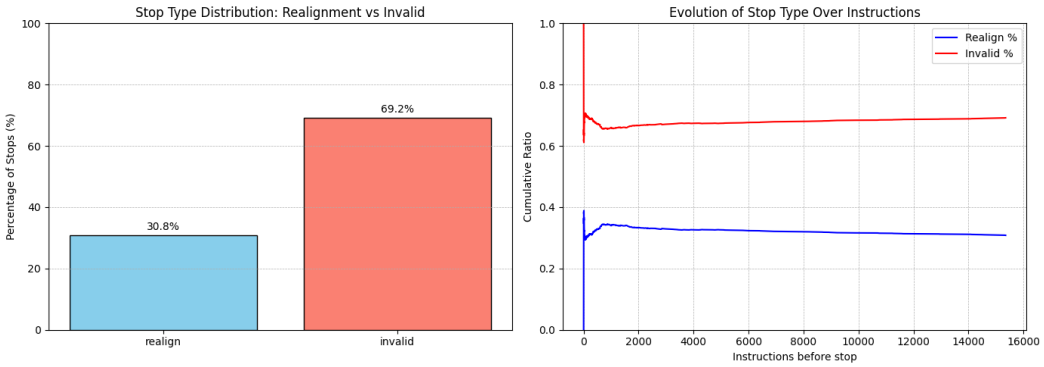

Figure 44: fpc 64

4.5 Binary using math functions

4.5.1 Openssl

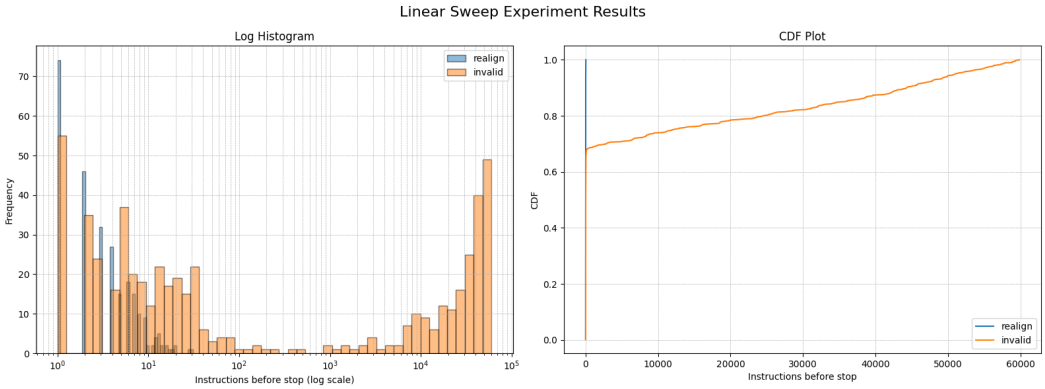

Figure 45: openssl 64

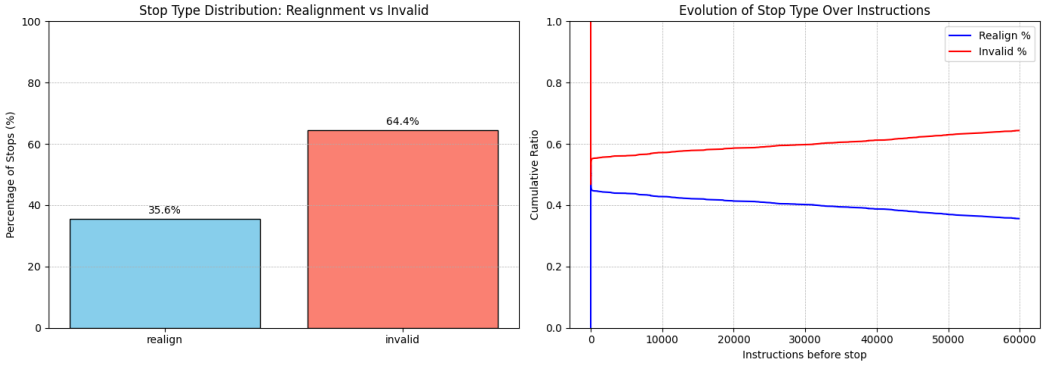

Figure 46: openssl 64

4.5.2 BC (Calculator)

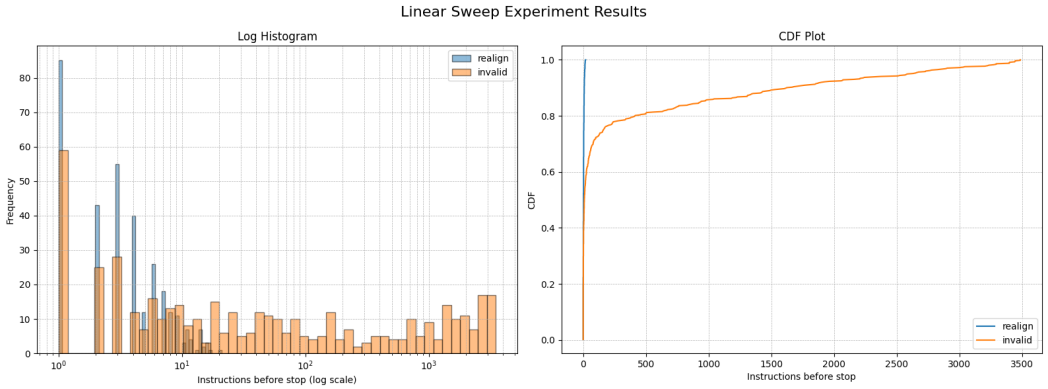

Figure 47: bc 64

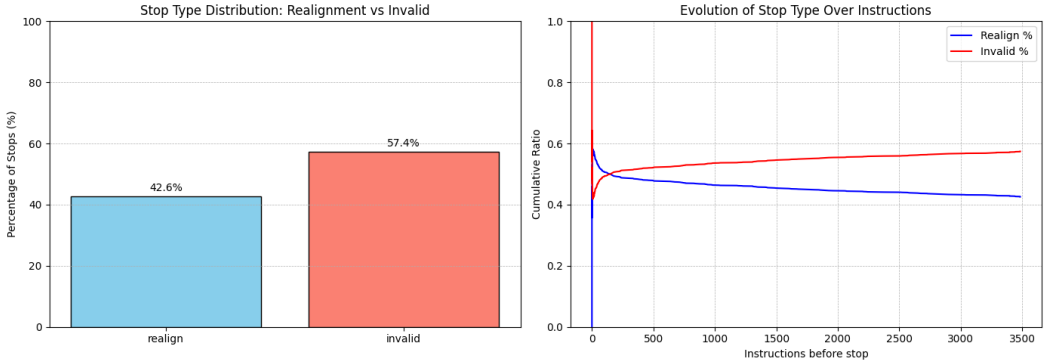

Figure 48: bc 64

4.5.3 GPG

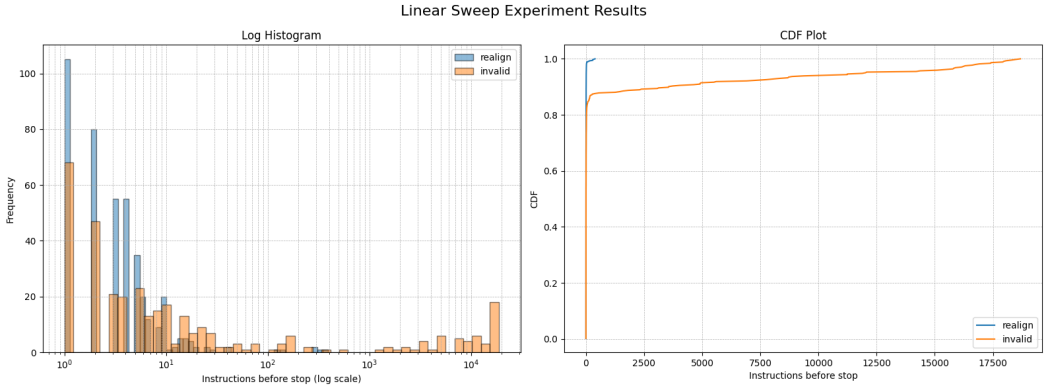

Figure 49: gpg 64

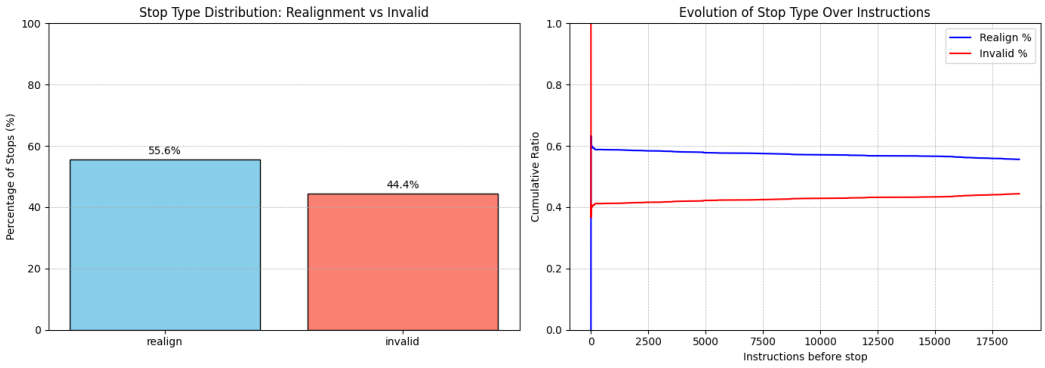

Figure 50: gpg 64

4.6 Binary using compression functions

4.6.1 Zstd

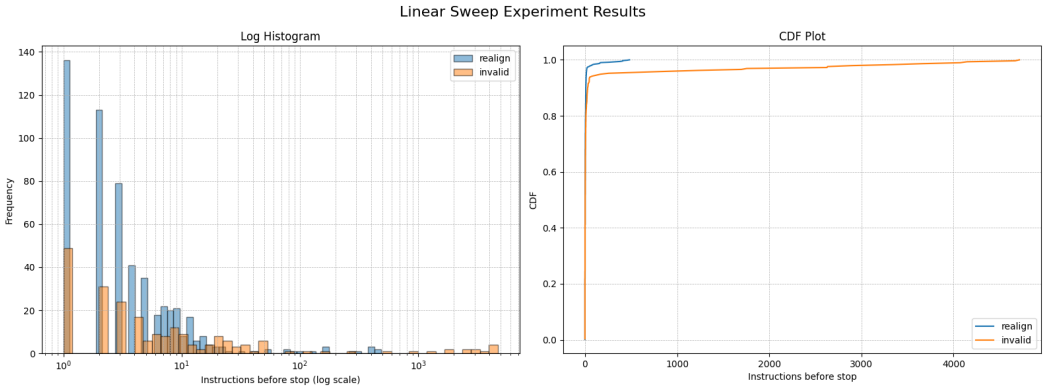

Figure 51: zstd 64

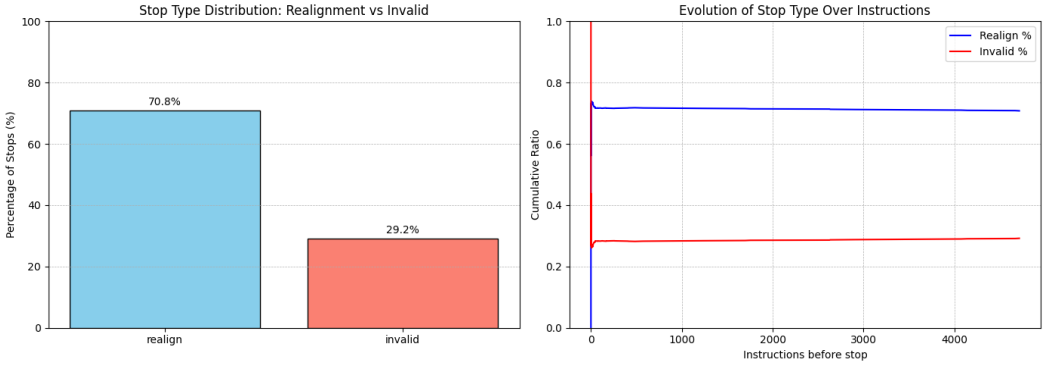

Figure 52: zstd 64

4.6.2 fpc (Pascal)

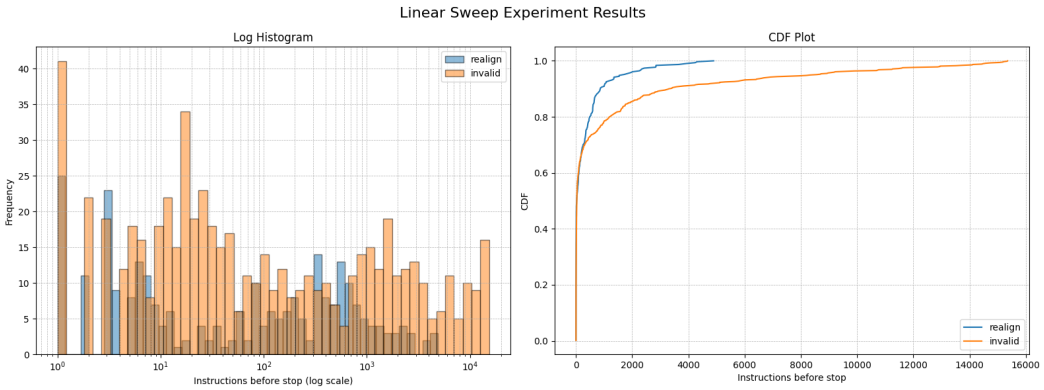

Figure 53: fpc 64

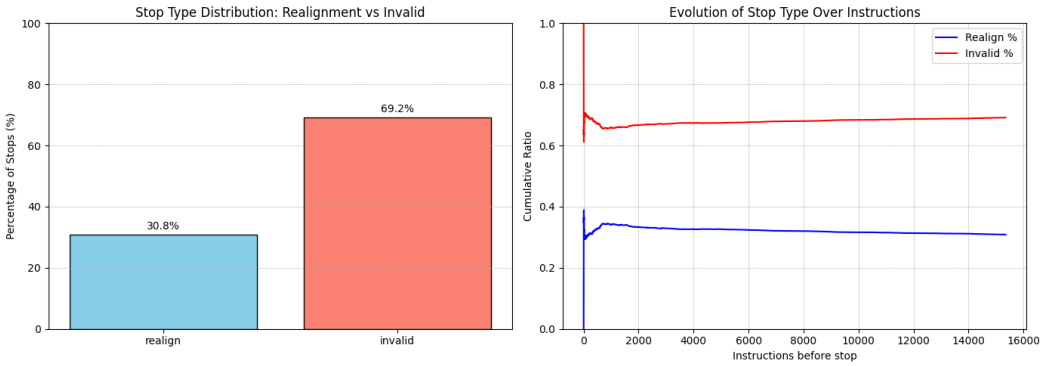

Figure 54: fpc 64

4.7 Compiled with rustc

4.7.1 Procs (ps like command line tool)

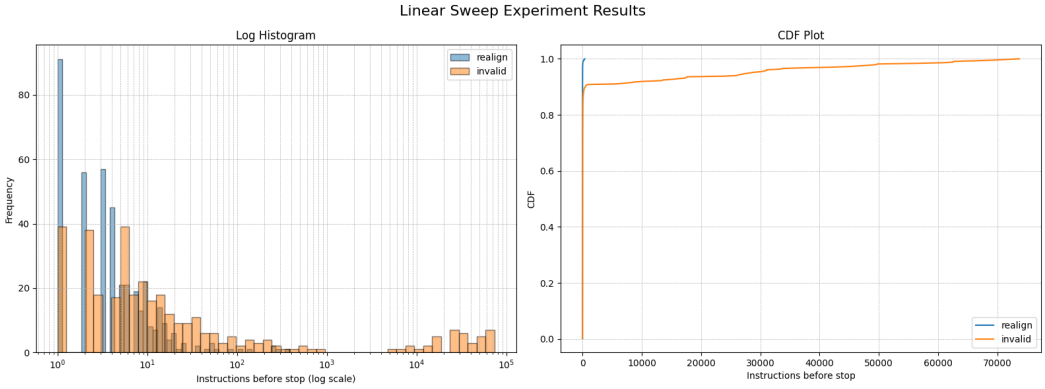

Figure 55: procs 64

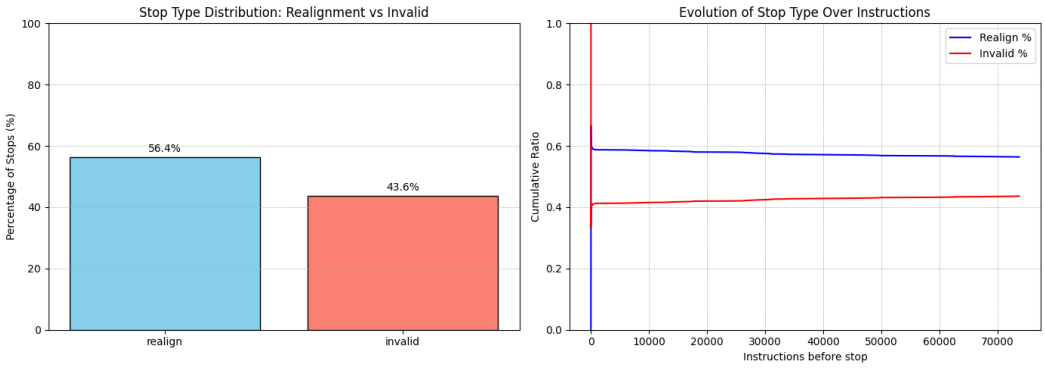

Figure 56: procs 64

4.7.2 btm (htop like)

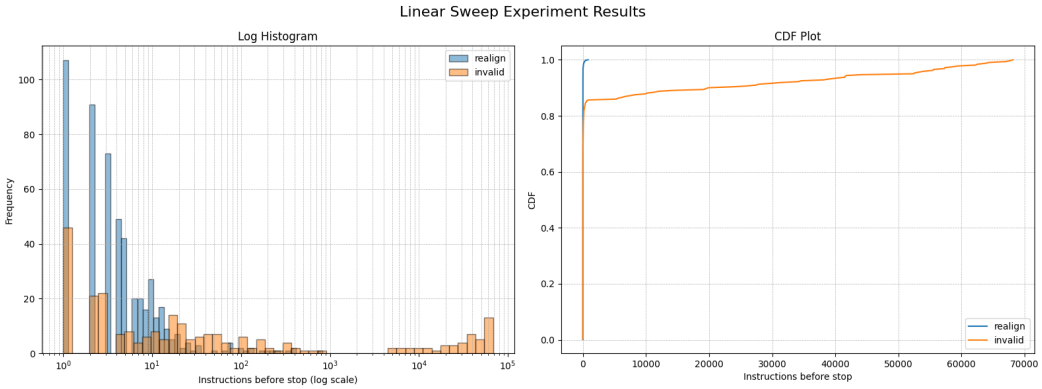

Figure 57: btm 64

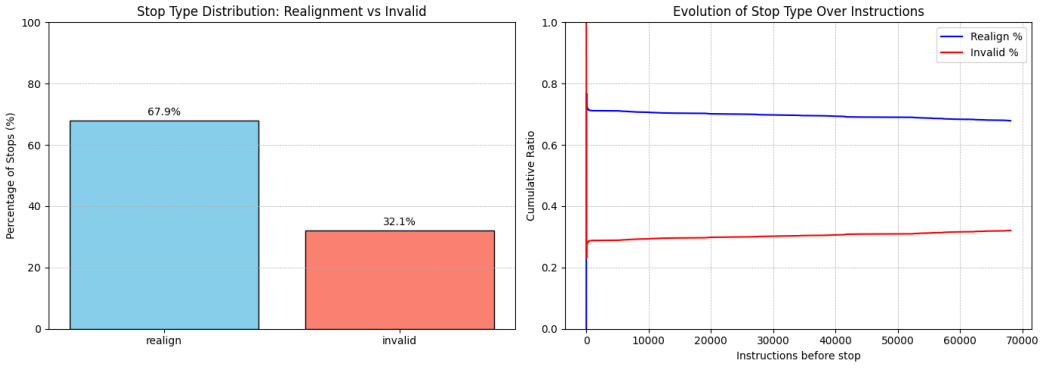

Figure 58: btm 64

4.7.3 fd (File Descriptor)

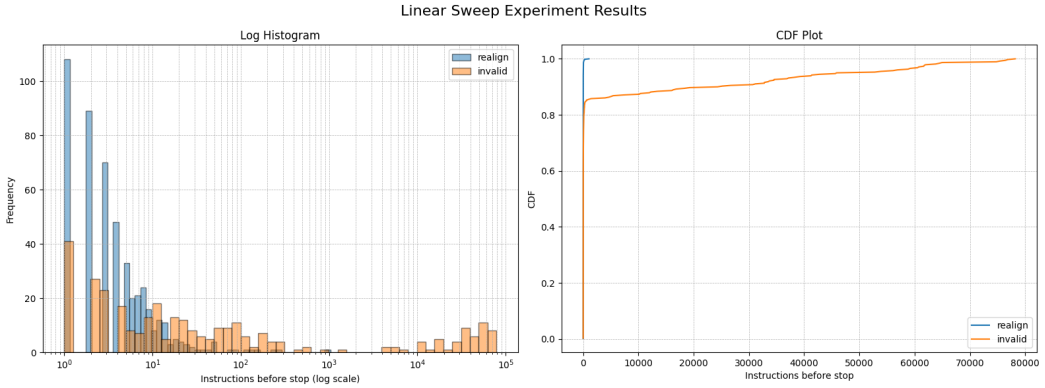

Figure 59: fd 64

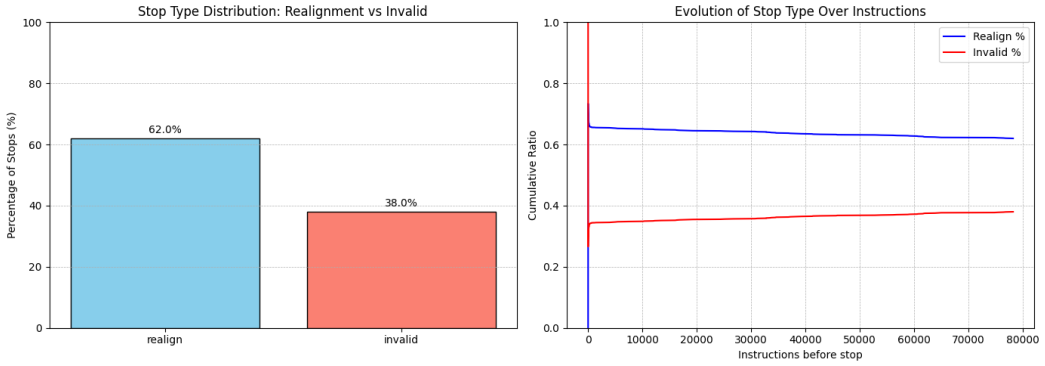

Figure 60: fd 64

4.8 Benchmark

4.8.1 Gnome Clock

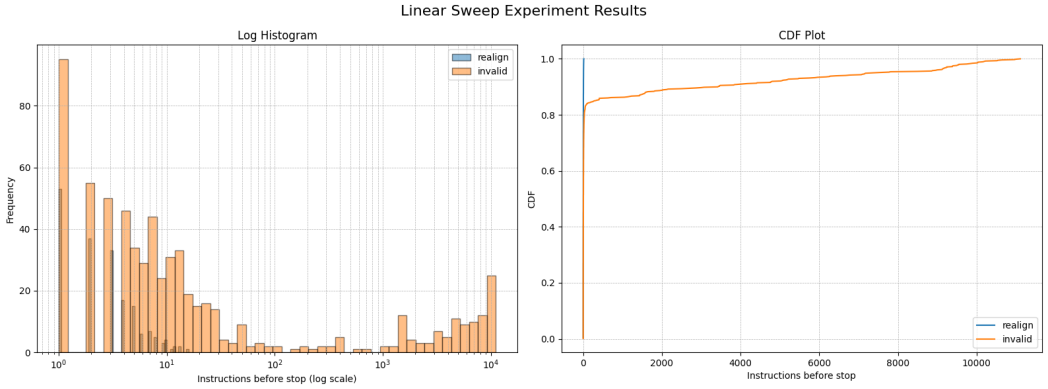

Figure 61: Gnome Clock 64

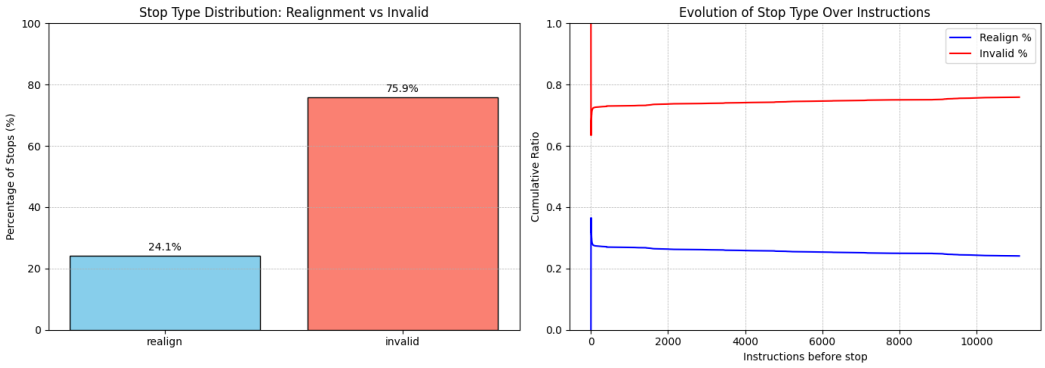

Figure 62: Gnome Clock 64

4.8.2 Hyperfine (compiled by rustc)

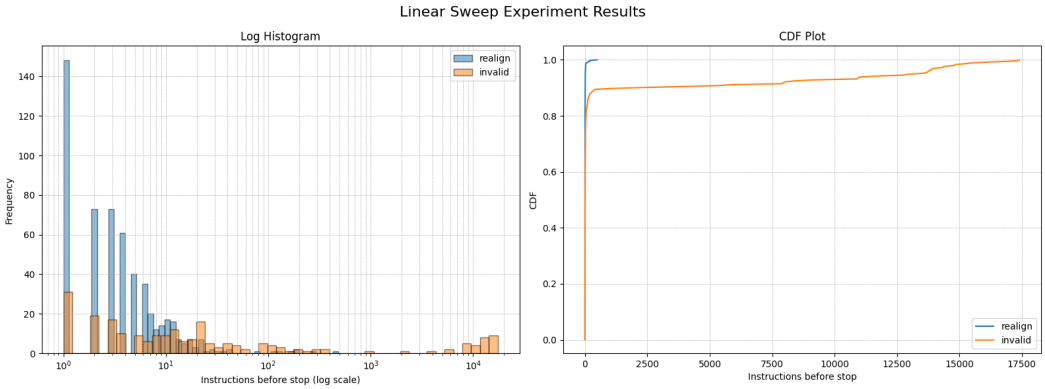

Figure 63: Hyperfine 64

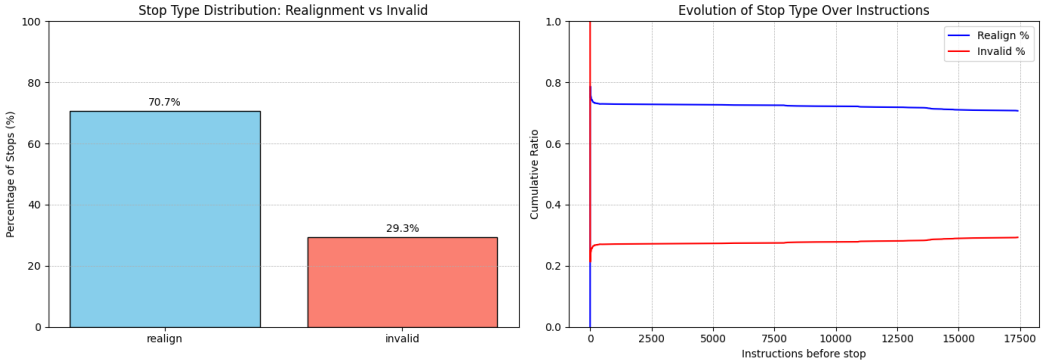

Figure 64: Hyperfine 64

5 Conclusion

Unfortunately I didn't have enough time to analyse more deeply this huge quantity of data. But during the experiment I noticed some patterns among groups of binaries : binaries compiled by rustc, binaries using math functions, binaries that are optimized in different levels etc. I noticed some differences and similarities among different compilers (I studied 7

compilers here). And also some binaries that has some unique behaviour : plocate 64 for instance.

Acknowledgements

This project was completed as part of the course "Forensics" at EURECOM. I would like to thank **Prof. Davide Balzarotti** for his guidance and valuable feedback.