Automated Reverse Engineering Project Proposal
Factors Influencing Gadget Discovery with ROPGadget
EEL4930
November 18, 2024

Shane Ferrell

Abstract:

The purpose of this project is to examine the factors influencing an attacker's ability to discover return-oriented programming gadgets in x86 executable code. This project investigates one of the main claims made in the 2007 paper by Shacham, which states that for any sufficiently large x86 binary, there will exist at least one chain of gadgets that enables arbitrary computation. To investigate, 144 different binaries with various properties were analyzed using ROPgadget, and the results were explored with descriptive statistics and visual inspection. The analysis verified Shacham's assertion and found that the most impactful factor in the number of gadgets and gadget discovery time is code size and its related factors like linkage type, and that other factors such as program purpose, structure, or optimization level did not have a significant impact.

Introduction & Background:

This project investigates one of the main claims made in the 2007 paper "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)" by Shacham, which states that for any sufficiently large x86 binary, there will exist at least one chain of gadgets that enables arbitrary computation. A return-oriented programming (henceforth referred to as ROP) gadget is a short sequence of machine instructions ending in a ret instruction, which allows the CPU to jump to a memory address specified on the stack. By carefully crafting the stack to chain multiple ROP gadgets, an attacker can execute arbitrary computations by manipulating the program's execution flow without introducing new code.

Shacham's assertion raises several questions: What is a "sufficiently large" binary? What factors determine the density of ROP gadgets within a binary executable? How do compilation techniques, optimization levels, and other properties of executable code influence an attacker's ability to construct chains? These questions highlight the need for analysis of ROP gadget discovery under varying conditions.

The goal of this analysis is to test Shacham's claim: does there exist a sufficiently large x86 binary that has no potential ROP gadgets?

Methodology & Challenges:

To determine how various properties of executable binaries impact gadget discovery, ROPGadget is the tool of choice. ROPGadget is an open-source tool that automates the search of gadgets for ROP exploitation in executable binaries. ROPGadget was used to identify gadgets in the benchmark dataset.

This experiment uses factorial experimental design, where multiple factors are tested simultaneously and evaluated at multiple levels. The dataset consists of 18 programs written in C++ that are categorized under 3 purposes: Arithmetic Operations, String Manipulation, and File I/O. For each purpose, the programs are categorized under 2 structures: Dense Loops and Sparse Function Calls. Each program was implemented with 3 code sizes: small, medium, and large. The source code for each program is in a file called main.cpp. Each main.cpp program was compiled with 4 levels of optimization using gcc: -O0, -O1, -O2, and -O3 and linked both statically and dynamically with a Bash script compile.sh, producing 144 executable binaries.

The 18 programs, 144 executable binaries that make up the dataset, and the script for compiling these executables can be found at the following repository: https://github.com/shaneferrellwv/gadget-discovery

ROPgadget was used to search for gadgets in each executable binary. The search was set to filter for strictly ROP gadgets rather than all types of gadgets and used a maximum depth of 10. ROPgadget also attempted chain generation for each executable. For each search, the program purpose, structure, source code size (categorical), source code size (in bytes), binary size, optimization level, linkage, number of unique gadgets discovered by ROPgadget, and time to complete gadget search and chain generation were recorded for analysis.

Analysis consisted of comparing basic descriptive statistics for how the independent variables of:

1) program purpose (arithmetic operations, string manipulation, file I/O)
2) source code size (small, medium, large)
3) executable binary size (in bytes)
4) program structure (dense loops vs. sparse function calls)
5) compiler optimization level (-O0, -O1, -O2, -O3)
6) linking (dynamic vs. static)

influence the dependent variables of:

1) number of gadgets discovered by ROPGadget
2) time to complete search of gadgets by ROPGadget

along with visualization of the data in a jupyter notebook using pandas, matplotlib, and seaborn for inspection.

This analysis exclusively employs descriptive statistics to summarize and explore the data, focusing on trends and patterns without making inferential claims. As a result, the findings should be interpreted as a preliminary exploration rather than a definitive evaluation of the factors influencing ROP gadget discovery.

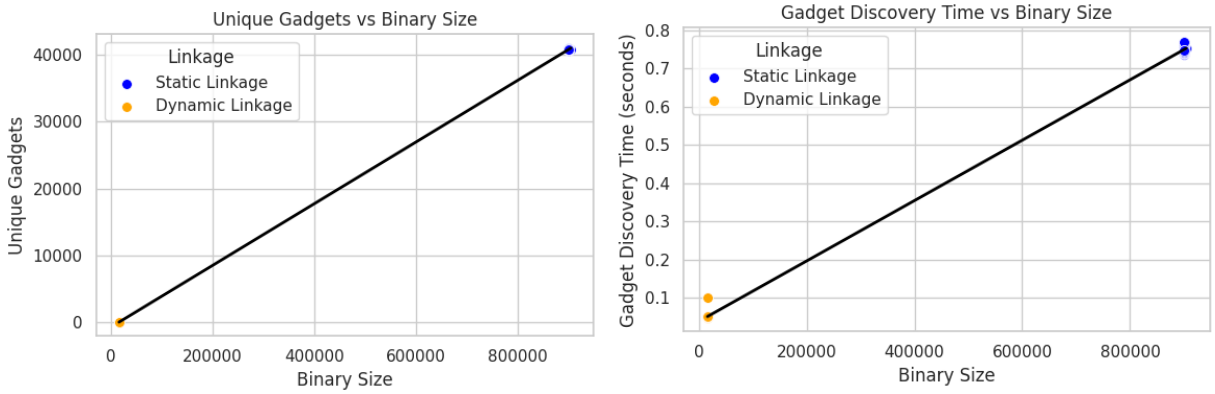
Another challenge is the impact of linkage's strong effects on ROPgadget's discovery. To keep the data clear, visualizations are displayed separately for linkage types.

Results:

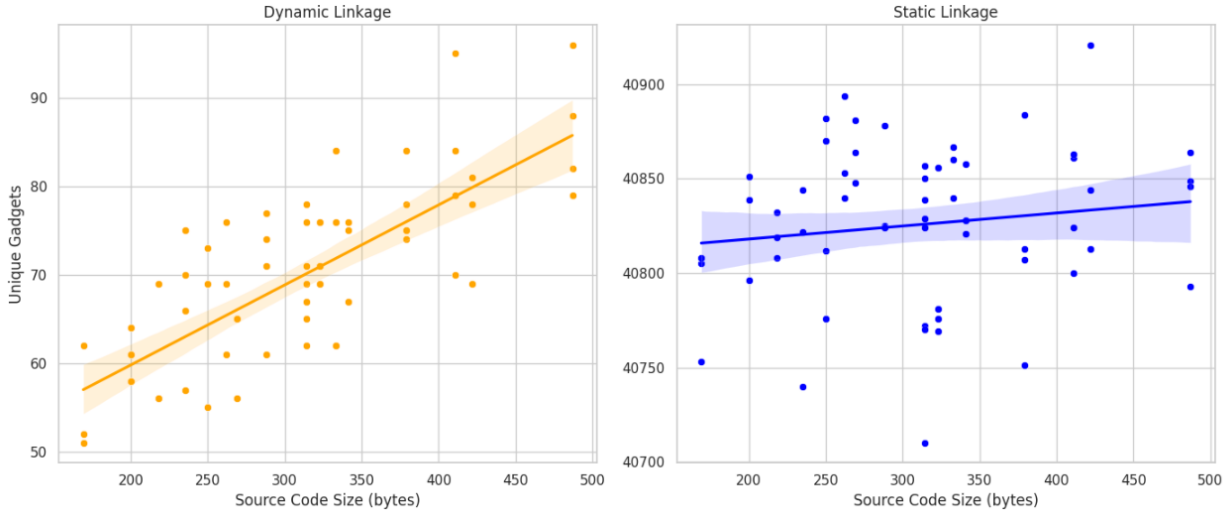
A table of all data and respective analysis can be viewed in the analysis.html file at the following repository: https://github.com/shaneferrellwv/gadget-discovery

Code Size

The first remarkable relationship is between binary size and the number of gadgets discovered and gadget discovery time, aligning with the expectation that a program with more instructions will have more potential ROP gadgets available and therefore spend more time searching. This relationship is not immediately clear when isolating for linkage but becomes evident when inspecting the entire dataset.



Similarly, the effect of source code size aligns with this finding, as programs with greater source code size generally result in greater executable binary sizes. This effect is more pronounced in the smaller, dynamically-linked executables of this dataset and does not appear as significant in the larger, statically-linked executables.



Linkage

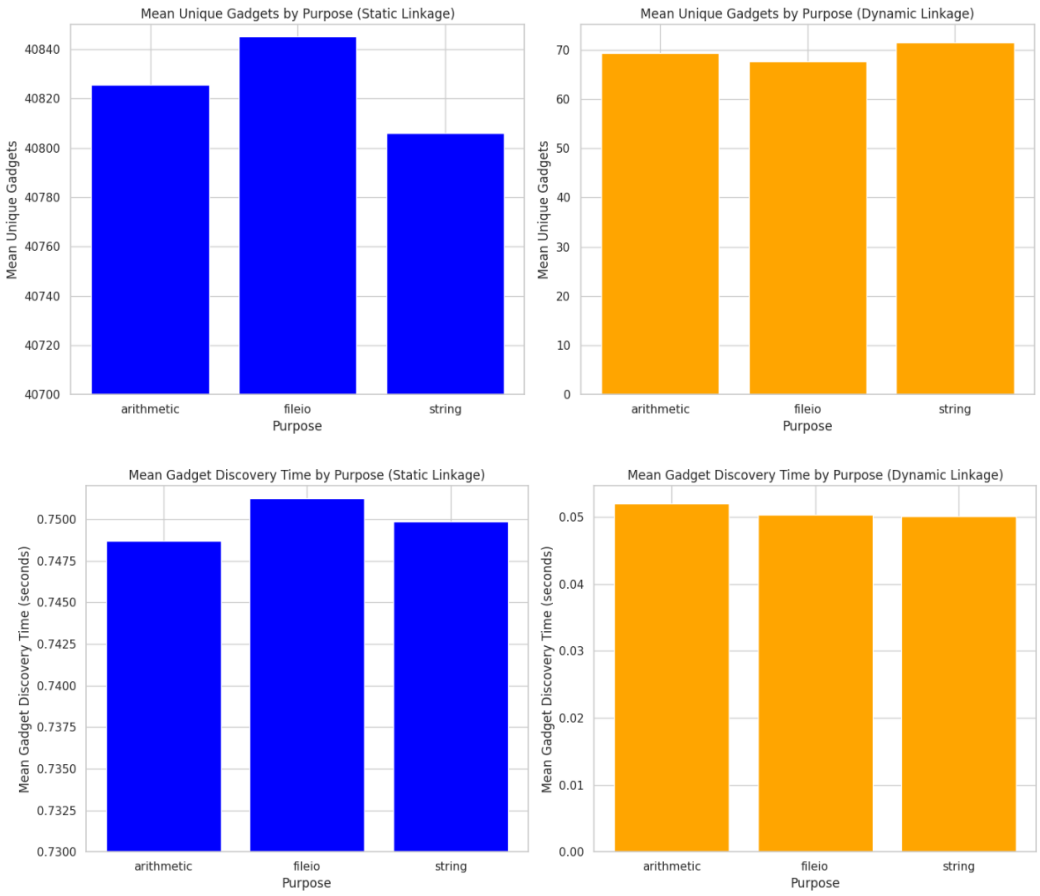
Another remarkable relationship is apparent between linkage type and the number of gadgets discovered and gadget discovery time, aligning with the idea that statically-linked binaries must contain more instructions from its linked libraries within the executable, resulting in larger binaries. The tables below display this strong relationship.

Linkage	Unique Gadgets
dynamic	69.555556
static	40825.513889

Linkage	Average Gadget Discovery Time (seconds)
dynamic	69.555556
static	40825.513889

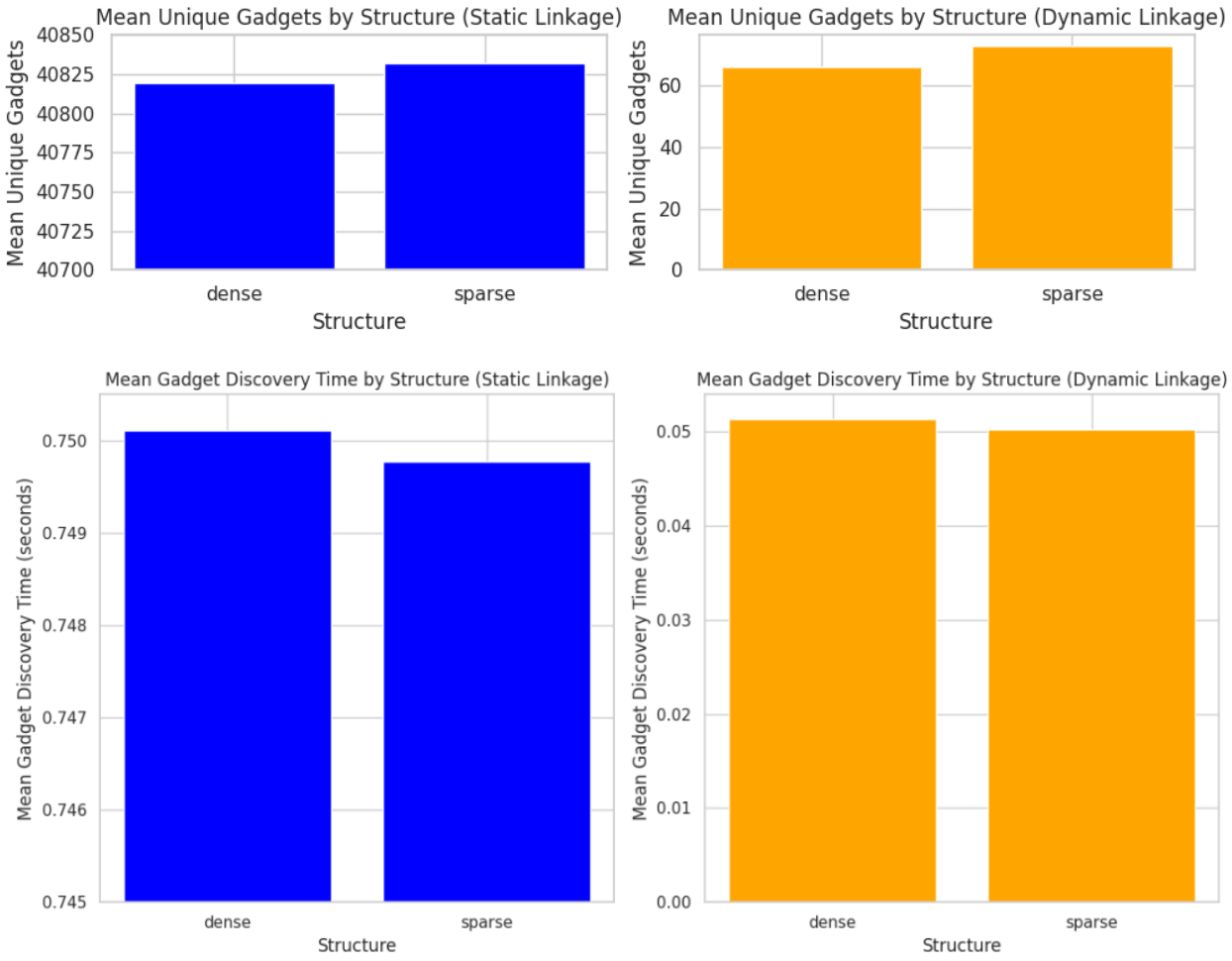
Program Purpose

By inspection of the means of the data, there is no apparent relationship between program purpose and the number of gadgets discovered or gadget discovery time. The small discrepancies in the results appear to be insignificant across linkage types and code size.



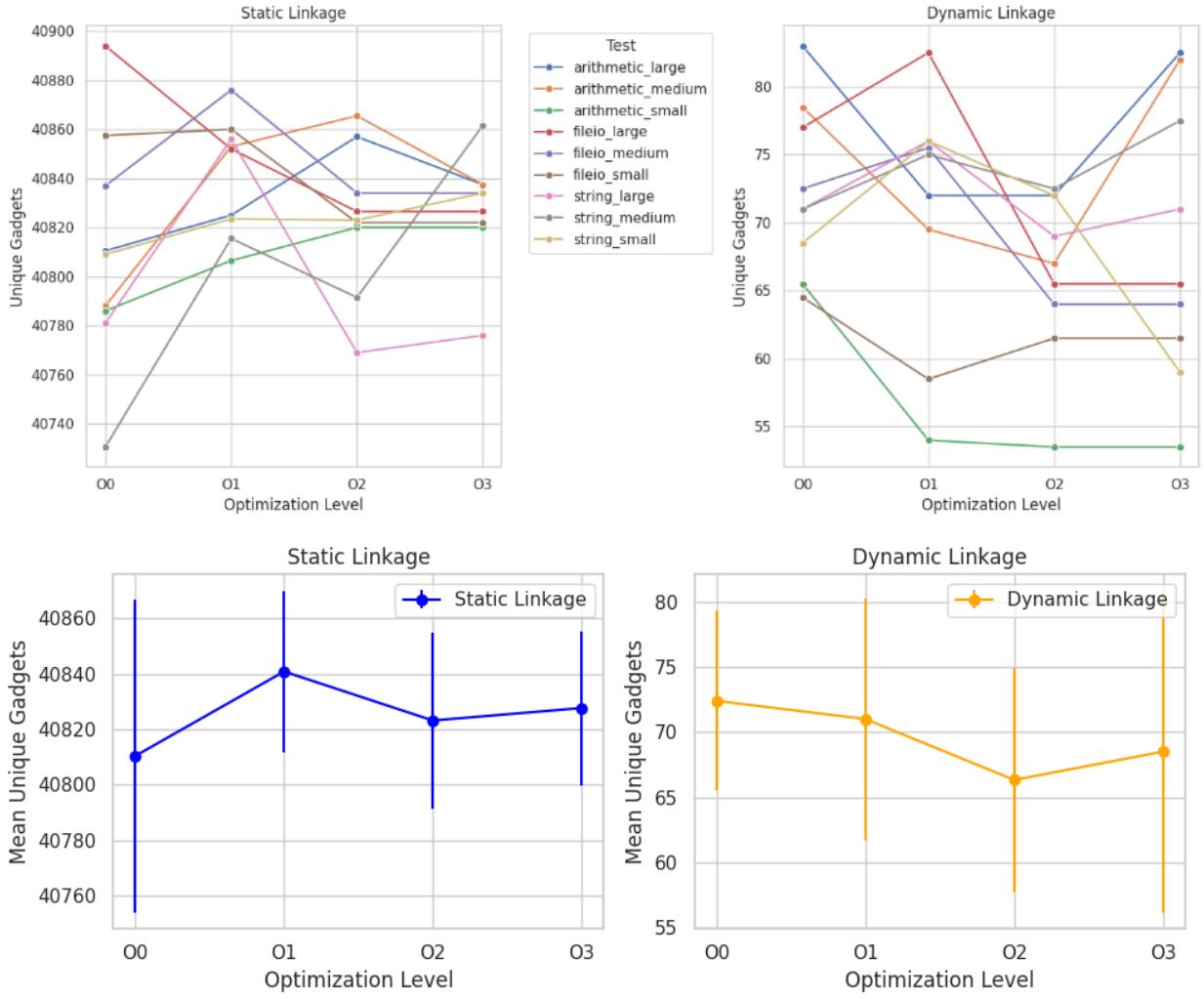
Code Structure

By inspection of the means of the data, programs with sparse function calls result in more gadgets discovered than programs with dense loops. However, gadget search appears the same if not slightly faster for sparse function calls, even though programs with sparse function calls appear to contain more gadgets.



Optimization Levels

There is no apparent relationship between optimization level and number of gadgets discovered or gadget discovery time.



Conclusion:

From this analysis, the claim that any sufficiently large x86 executable binary will contain at least one ROP gadget chain that enables arbitrary computation is valid. Of all the combinations of programs under various conditions, there were no factors identified that can produce an executable free of ROP gadgets.

The most influential factor affecting the number of ROP gadgets in an executable binary is the code size. Larger binaries contain more instructions and, therefore, more potential ROP gadgets. The other variables related to code size all aligned with this finding, as statically linked executables, which are typically much greater in size than dynamically-linked executables, contained many more gadgets than their smaller counterpart.

Programs with longer source code, which generally results in greater binary code size, were found to contain more gadgets for exploitation.

Analysis showed that there was no apparent relationship between program purpose and number of gadgets or time spent searching for gadgets. There was also no clear effect resulting from the optimization level of the executable on the number of gadgets or gadget discovery time, and there was a slight but seemingly insignificant impact resulting from code structure on number of gadgets found and time spent searching for gadgets.

I expected Shacham's claims to be valid, and I expected the number of gadgets and gadget discovery time to be directly related to code size. However, I was expecting a clearer effect resulting from the optimization level, since greater optimization is loosely tied to reduced code size. From my initial reaction to hearing Shacham's claim, I was also expecting the smallest, simplest, dynamically linked programs to contain fewer gadgets, but there are still dozens available, which makes sense considering essential functions in program initialization results in ret instructions always being present. I think this analysis could benefit from more advanced statistical modeling techniques, but I made sure not to overgeneralize nor make broad unfounded claims.

References:

Shacham, H. (2007). *The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)*. Proceedings of the 14th ACM Conference on Computer and Communications Security. https://doi.org/10.1145/1315245.1315313

This study used ROPgadget (version 7.5) by Jonathan Salwan, available at https://github.com/JonathanSalwan/ROPgadget