Final Report

Background

Diversification:

Code diversification transforms each compilation of binaries into a functional and structurally unique variant, changing code layout between every run. Ensures that each binary has different gadget locations(not all), instruction sequences and stack layouts. We have implemented this with layout randomization, instruction reordering, code insertion.

An attacker who obtains one diversified binary gains limited information about others. The security level depends on the entropy introduced by each transformation and how effectively transformations break the predictability that attacks require. Our goal is to maximize the cost of attack construction while minimizing performance overhead.

Countering ROP

We focus on countering return-oriented programming (ROP) attacks using code diversification. Three complementary defense levels can be identified: increasing the difficulty of stack-smashing attacks, diversifying the memory locations of potential gadgets, and modifying the gadgets themselves. This project primarily targets the second level by applying diversification to program code, thereby randomizing gadget addresses and reducing their predictability.

Code diversification does not aim to eliminate ROP attacks entirely, but to reduce exploit reliability by breaking assumptions about stable gadget availability and addresses. By increasing unpredictability across program instances or executions, diversification raises the practical cost of constructing reliable ROP chains and is best viewed as a complementary mitigation.

Methodology / design choice

LLVM Framework

We implemented our diversification pipeline as LLVM transformation passes operating on LLVM Intermediate Representation. The IR level provides sufficient abstraction to reason about program behavior while remaining close enough to machine code for meaningful security transformations. Each transformation is implemented as an independent pass that accepts an LLVM Module. This modular design allows flexible composition: passes can be enabled, disabled, or reordered as needed.

Target-Agnostic Passes

Since we were all novel in LLVM, we have used target-independent passes, operating on LLVM’s intermediate representation (IR). We have not implemented target-specific passes in the backend. Target-aware instructions would likely have

increased efficiency against ROP, for example by creating passes with more detailed and aimed effects, such as diversifying chosen registers in the x86-64 architecture.

Compiler Optimization Level
We have disabled optimizations (-O0) when compiling with our passes. This is mainly because our passes often have a negative effect on efficiency which the compiler in later backend stages may undo.

Implementation details

Basic Block Randomization (bbrand.cc)
This pass randomizes the physical order of basic blocks within each function while keeping the entry block first, preserving control-flow semantics and affecting only code layout to shift gadget addresses. However, IR-level block ordering is not guaranteed to survive code generation, as backend optimizations, particularly machine-level block placement for fall-through, may override the layout. We observed that the randomization is more likely to persist at lower optimization levels, whereas higher levels tend to undo it; a backend (Machine IR) implementation scheduled late in the pipeline would provide more stable diversification.

Constant Alteration (constant_altering.cc)
Diversifies integer constants by replacing them with equivalent XOR expressions. For each constant C, generates a random mask R and rewrites the constant as $(C \oplus R) \oplus R$, which evaluates back to C at runtime. Some constants in LLVM IR have to remain as simple constants, but we did not find a way to easily determine if that was required of a specific constant, so instead we made it skip constants used in switch instructions, alloca sizes, and intrinsic calls.

Sometimes, a constant embedded in an instruction may be parsable by the CPU as an instruction itself. This pass makes that method completely unreliable, as the constants will be different in each compilation.

Function Randomization (func_rand.cc)
This pass is similar to the basic block randomization pass. It shuffles the order of function definitions in the module. It collects all non-declaration functions into a vector, shuffles with the provided RNG, then reorders them in the module's function list. The first shuffled function moves to the front, subsequent functions insert after the previous one. Changes function addresses in the compiled binary since position in the object file determines memory layout. As such, it also randomizes the gadgets' addresses.

Function Splitting (func_splitter.cc)
Splits functions by extracting blocks after a split point into a new function. Finds candidate blocks with a single predecessor ending in an unconditional branch.

Identifies live-in values which become parameters to the new function. Clones extracted blocks into the new function, remaps all value references, then replaces the original branch with a call to the split function followed by a return. Skips main, vararg functions, and exception handling. Changes function boundaries and call graph structure, preventing code matching across variants. Only splits one function per pass run.

Garbage Insertion (garbage_insert.cc)
Inserts semantically neutral operations that don't change program behavior. Tracks available integer values as it iterates through each basic block, then randomly inserts dead instructions using one of five patterns: x + n - n, x * 2 / 2, x ^ n ^ n, x | 0, or x & -1. Skips terminators, PHI nodes, and allocas to preserve IR validity. The inserted instructions are never used, but they increase code size, shift subsequent instruction addresses, and pollute gadget searches with useless sequences.

Instruction Reordering (inst_reorder.cc)
Randomly reorders independent instructions within basic blocks. Identifies consecutive instruction pairs that can safely swap by checking: no data dependencies neither uses the other's result, no special instructions (allocas, terminators, calls, PHI nodes), and no conflicting memory accesses. For memory operations, proves safety by checking if pointers point to definitely different locations such as different allocas, different globals, or alloca vs global. Each reorderable pair has 50% chance of swapping. Changes instruction sequences that attackers might use as gadgets without affecting program semantics.

Instruction Substitution (inst_sub.cc)
Replaces arithmetic operations with mathematically equivalent but structurally different sequences. For addition: a + b becomes either a - (0 - b) or 0 - ((0 - a) + (0 - b)). For subtraction: a - b becomes a + (0 - b). For multiplication: a * b becomes either 0 - ((0 - a) * b) or a * (b - 1) + a. Collects all integer add/sub/mul operations, randomly selects a variant, builds replacement instructions using IRBuilder, then replaces all uses of the original and erases it. One instruction expands to 2-4 instructions with identical results but completely different binary patterns, breaking gadget signatures.

Loop Flattening (loop_flatten.cc)
Randomly selects simple bounded loops with one additional bounded loop inside in its body, and flattens it into a single bounded loop. For example, if the outer loop iterates x from 1 to 10, and the inner loop iterates y from 5 to 15, then the result would be a single loop that iterates k from 1 to 100, with x = 1 + (k // 10) and y = 5 + (k % 10). This eliminates one jump for when going from the inner loop to the outer, and generally shifts memory addresses unpredictably.

Loop Splitting (loop_split.cc)

Randomly selects simple bounded loops and replaces them with multiple loops on the same level that functionally do the same thing, each covering a subset of the original loop’s range, without overlapping with each other. For example, if a loop iterates from 1 to 10, you can have one loop from 1 to 3, then another loop from 4 to 10. This modifies memory addresses by introducing unnecessary code duplication.

Stack Variable Reordering (stack_reorder.cc)
Shuffles the order of stack allocations to randomize stack layout. Collects all contiguous alloca instructions from the function's entry block, stops at first non-alloca. Shuffles the collected allocas using the RNG, then reinserts them at the block beginning in the new order using moveBefore(). Requires at least 2 allocas to be meaningful. Buffer overflows now affect unpredictable variables, and the distance from buffers to return addresses varies per build. Zero runtime overhead since it only changes allocation order, not the allocations themselves.

Stack Padding Randomization (stackpad_rand.cc)
This pass adds random padding to each function's stack frame to make return address offsets unpredictable. It generates a random multiplier (1-16) and allocates 16 * k bytes (16 to 256 bytes, 16-byte aligned for x86-64). It inserts the padding alloca at the start of the entry block, before existing allocas. Performs a volatile store of zero to the first byte to prevent the compiler from optimizing away the unused allocation. The results is that it makes the distance from local variables to the return address vary per function and per build, breaking stack smashing exploits that rely on fixed offsets.

Extra Passes
Shadow Stack (shadow_stack.cc)
This pass is an exploratory implementation of a shadow stack to harden return addresses against stack-smashing attacks by requiring an attacker to corrupt both the process stack and a separate shadow stack. Because the shadow stack resides in the same address space as the main stack, it does not provide strong standalone protection against ROP but may still be useful as part of a defense-in-depth or diversification strategy. The pass maintains a parallel stack using two globals (shadow_stack[1024] and shadow_sp), stores the return address obtained via llvm.returnaddress(0) at function entry, and verifies it at function exit, aborting execution on mismatch. All original returns are replaced with branches to a shared epilogue, using PHI nodes to propagate non-void return values.

Evaluation

Entropy

The evaluation is done by measuring two metrics before and after the transformation: file entropy and gadget count. For this task, an arbitrary C program with a buffer overflow vulnerability was created (see Appendix A).

File Entropy

If the entropy of the transformed binary executable is higher than the original, we can say that it is more randomized and harder to predict. Using binwalk's --entropy operation, we can calculate and generate the following graphs that show entropy values across the respective file.

Note that the diagrams below normalize the entropy in the vertical axis: 1.0 signifies maximum randomization (8 bits/byte), while 0.0 means perfect predictability. We are mainly interested in the .text section in the binary, which houses the code body that is to be executed.

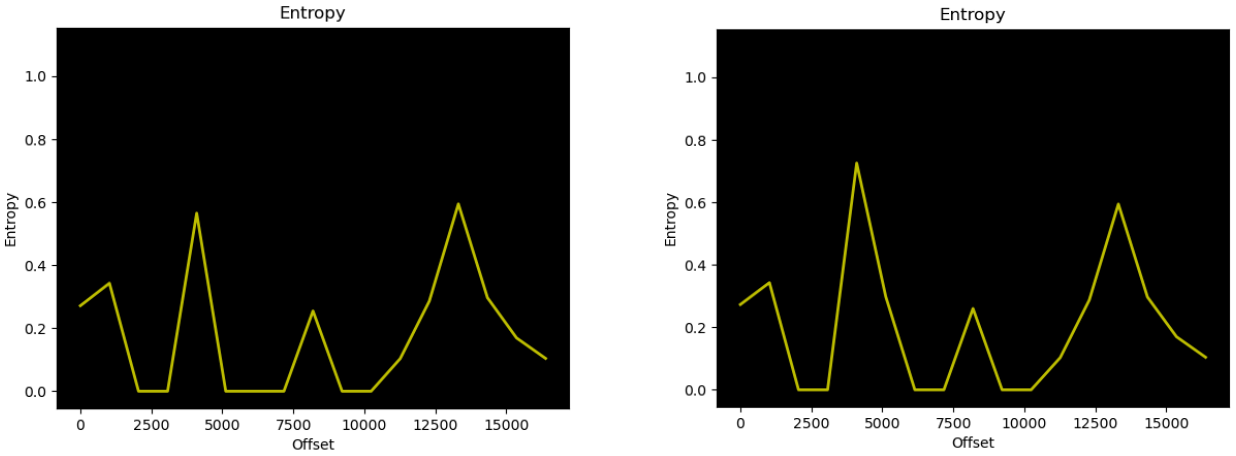

Diagram 1 (left): Entropy of the original, unmodified program.
Diagram 2 (right): Entropy of the transformed program after all passes.

In the original program (diagram 1), the .text section spans 0x4010c0-0x401320, or in decimal offset, 4288 to 4896, which matches the entropy spike in the left diagram — approximately 0.6 when normalized.

In the program after transformation by all passes (diagram 2), the .text section spans 0x4010c0-0x401533, or in decimal offset, 4288 to 5427, which matches the taller entropy spike in the right diagram — approximately 0.7 after normalization, proving success at increasing entropy. Individual pass evaluation and attempts at reordering passes indicate that this increase is almost entirely attributed to the constant altering pass (constant_altering.cc) (diagram 3).

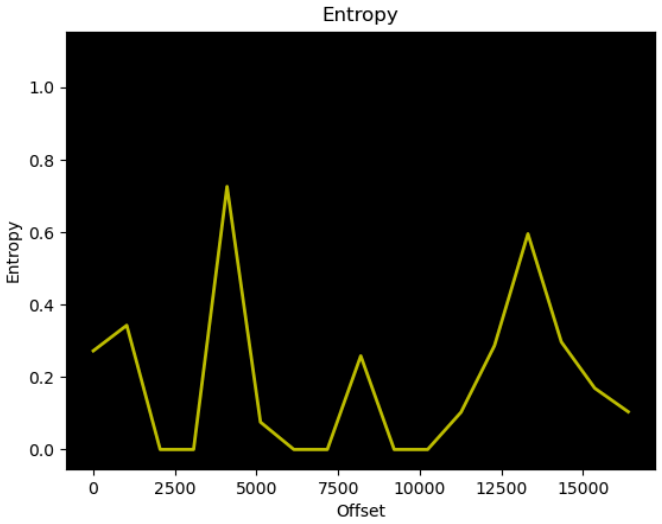

Diagram 3: Entropy of the transformed program with only the constant_altering pass.

Gadget count
Using ROPgadget, the original program had 97 unique gadgets. Meanwhile, one instance of the transformed program (with all passes) had 118.
ropper found 112 (original) and 122 (transformed) gadgets respectively, while kropr, which is said to exclude a subset of useless or duplicate gadgets, found 34 (orig) and 43 (transformed) respectively.

From each of these tools' outputs we could identify that several gadgets are being repositioned and modified. For example, in one instance, the gadget `call qword ptr [rsi+2];` at 0x123e was changed to `jmp qword ptr [rsp+rsi*4+0x7d];` at 0x11fa. In addition to getting a new address itself, the gadget's target address is made complicated, and the `call` is changed to `jmp`.

As expected from the randomness in our passes, re-compiling the program provided different gadgets in different addresses, and the amount of gadgets also varied.

Correctness

To test whether our passes keep programs' functionality intact or break them, we wrote small C programs to test our passes on during development. Later, we also set up a slightly more comprehensive correctness test. We did this by running our passes on a larger open source project. Specifically we searched for a project written in pure Zig, since compiling Zig to LLVM IR yields one big file rather than many small ones as is typically produced when compiling C, C++ or Rust. We specifically found TigerBeetle, which has around 100 000 lines of Zig code. We performed the test by compiling TigerBeetle to LLVM IR, running our passes on the IR and then compiling the transformed IR using the Zig compiler. Running the TigerBeetle program with our passes, we set up a cluster of three replicas and connected to it with the project's

repl, and tried interacting with it as provided by the project's "Getting Started" documentation page.

The results of this test was that the passes *Function Randomization*, *Stack Variable Reordering*, *Basic Block Randomization*, *Instruction Substitution*, *Garbage Insertion*, *Loop Flattening*, and *Loop Splitting* seem to work correctly, while *Constant Alteration*, *Instruction Reordering*, *Instruction Substitution*, and *Stack Padding Randomization* have some bugs that made TigerBeetle either not compile or crash when starting.

Individual contributions: Zhongmin Hu

My contributions to the project are mainly the two loop-level obfuscation passes: Loop Flattening (loop_flatten.cc) and Loop Splitting (loop_split.cc), as well as conducting the entropy analysis for the evaluation.

Loop flattening and loop splitting are two conceptually similar passes, which is why I chose to implement the two together. As I had initially focused on other courses and failed to read up on LLVM, it took me a while to catch up with the other group members: as such, the code for these passes were primarily composed by AI with some prompt experimenting and engineering from my side. The goal was that once I had a better understanding of LLVM, I could return to the code and understand it in greater detail. Because of this, there aren't many insights I can share in terms of challenges encountered in the coding process – understanding the code was relatively straightforward thanks to code comments, my groupmates' passes serving as simpler examples, and online resources.

Slotting the two passes into the established running order proved to be no issue, as these passes didn't modify the stack in any way that might cause issues for other passes. I also tried to write a third loop-level pass that sought to do the inverse of loop flattening – creating nested loops – but it proved fruitless and was cut due to time constraint.

The entropy analysis was researched and conducted by myself, with suggestions and pointers from other group members and minimal AI input. I discovered and experimented with a variety of command-line tools and learned more about entropy as I went along. One significant problem I had was getting `binwalk` to work on my local machine due to strange dependency mismatches – I ended up avoiding this by running it through Kali Linux in a virtual machine. I also wanted to make the gadget randomization example more relevant by showcasing outputs from automatic ROP chain builders, but struggled to get angrop to comply.

Beyond these main points, I've also done my best to participate in group discussions and put together the presentation. Due to the aforementioned slow start compared to

my groupmates, I could not actively contribute as much during many of our meetings – when this happens, I took the responsibility of notetaking on a few occasions. In spite of this, I believe I have done my part of the group work to an acceptable level.

Appendix A: Demo C program

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <unistd.h>

int main() {
   char buf[256];
   setvbuf(stdout, NULL, _IONBF, 0);
   while (true) {
       printf("[r]ead, [w]rite or [q]uit? ");
       char c; scanf(" %c", &c);
       if (c == 'q') break;
       printf("how much? ");
       size_t n; scanf("%zu", &n);
       if (n > 0x256) break;
       if (c == 'r')
           write(STDOUT_FILENO, buf, n);
       else
           read(STDIN_FILENO, buf, n);
   }
}

void win(uint64_t a, uint32_t b) {
   char *flag = getenv("FLAG");
   if (!flag) flag = "You have won. Very congrats.";
   if (a == 123 && b == 321) puts(flag);
   exit(0);
}

void typical_optimized_function_epilogue() {
   asm volatile (
       ".intel_syntax noprefix\n"
       "pop r12\n"
       "pop r13\n"
       "pop r14\n"
       "pop r15\n"
       "pop rbp\n"
       "ret\n"
   );
}
```