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 various binaries have different gadget locations, 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.

Limitations

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 generally used low optimization level (-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.

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 result 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.

Contribution - Mathias Grindsäter

I implemented the basic block randomization pass located in bbrand.cc. This pass operates at the intermediate representation (IR) level. I evaluated its effectiveness under different compiler optimization levels to determine whether the randomized block layout was preserved in the final binary. This involved inspecting the generated assembly and verifying that control flow correctness was maintained, particularly with respect to jump instructions introduced when breaking fall-through basic blocks.

Since basic block randomization is generally more effective when tailored to a specific target architecture, I also attempted to implement a corresponding pass at the machine IR level. However, due to integration challenges and increased complexity, this effort was ultimately not successfully completed.

I additionally implemented a function randomization pass in func_rand.cc. As with the basic block randomization, I validated its effectiveness by inspecting the final assembly output. The randomization was preserved when compiling with lower optimization levels, confirming that the pass functioned as intended under those conditions.

The stack padding randomization pass was implemented in stackpad_rand.cc. Beyond implementing the pass itself, a key challenge was ensuring that the compiler did not optimize away the introduced padding. This required additional measures to preserve the transformation through later compilation stages, which I verified by analyzing the final binary.

I also implemented a shadow stack pass in shadow_stack.cc, which was the most time-consuming pass to develop. During evaluation, I concluded that the approach was less effective than initially anticipated, as discussed earlier in the report. One potential improvement would be to apply the shadow stack to random subsets of functions, thereby increasing uncertainty for an attacker. A major implementation challenge involved handling multiple return statements within a single function (for example due to conditional branches). This was resolved through the use of PHI nodes.

In addition to the compiler passes, I developed an initial proof-of-concept program demonstrating a return-oriented programming (ROP) attack with a fixed gadget chain. This is located in rop/write4. While the original intention was to use this as an early project demo, another group member developed an alternative solution that we ultimately adopted. The write4 directory includes a README describing the program and payload, and represents an attempted reverse engineering of the fourth (“write4”) challenge from ROP Emporium.

Beyond the direct code contributions, the project required much knowledge in LLVM and IR-level development, ROP attacks, and x86-64 assembly, areas in which I had little prior experience. A rather large portion of the work therefore involved acquiring this knowledge. I primarily relied on LLVM’s official documentation and technical blogs for compiler development, practiced ROP exploitation using challenges from ropemporium.com, and studied x86-64 assembly using the book *x86-64 Assembly Language Programming with Ubuntu* by Ed Jorgensen.