# AMMF: Attention-Based Fusion for Cross-Architecture Binary Vulnerability Detection

Presented at **Conf42 GoLang 2026**

**Akshaya Jayaram**
M&A Security Principal at Salesforce

The Challenge

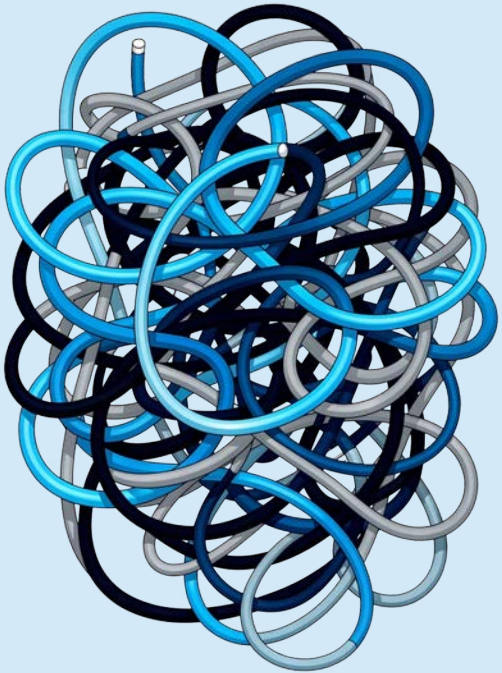
# Why Binary Vulnerability Detection Is Hard

## The Core Problem

Modern software ecosystems like Go enable a single codebase to compile and run across **multiple architectures** x86, ARM, MIPS, and beyond. This cross-compilation power is also a security blind spot: the same vulnerability can manifest differently depending on the target platform, compiler, and optimization level.

## What Makes It Difficult

- Compiler variations alter instruction sequences and calling conventions
- Optimization flags (O0–O3) restructure control flow dramatically
- Architecture differences obscure semantic equivalence
- Scale of real-world binary corpora demands efficient, automated analysis
- High false-positive rates erode analyst trust and slow triage



# The Cross-Architecture Complexity Gap

## Same Code, Different Binaries

Go's cross-compilation produces fundamentally different binary representations for the same source logic. Structural and syntactic matching techniques fail to bridge this gap reliably.

## Brittle Signature-Based Approaches

Traditional signature matching relies on static byte patterns. Cross-architecture variations break these signatures entirely, leaving known vulnerabilities undetected in recompiled binaries.

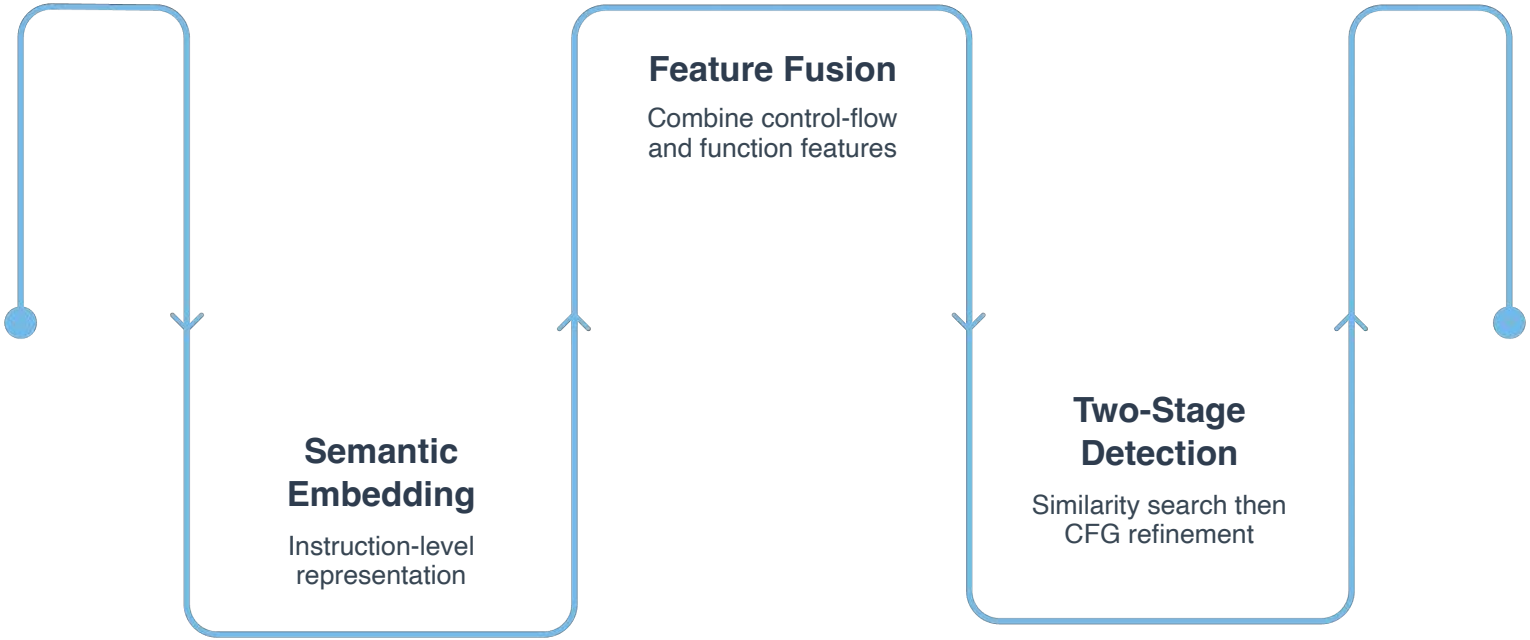
## Semantic Equivalence Is Lost

Even when two functions are semantically identical, their assembly representations diverge across architectures. Detecting this equivalence requires learned, semantic-aware representations.

Framework Overview

# Introducing AMMF

**AMMF** Attention-driven Multi-feature fusion is a neural architecture designed to detect vulnerable functions in compiled binaries, even when the target architecture, compiler, or optimization setting differs from the reference. It operates by learning *what a function means*, not just what it looks like.




AMMF's design separates semantic understanding from structural analysis, then unifies both through attention mechanisms to produce robust, discriminative function embeddings suitable for large-scale binary analysis.

Architecture Layer 1

# Instruction Semantics via Embeddings

## Learning What Instructions Mean

AMMF learns embedding representations that capture the semantic intent of each instruction, independent of ISA syntax. Equivalent operations, such as x86 MOV, ARM LDR, and MIPS LW, receive similar embeddings across architectures.

### Token Normalization

Operands and registers are normalized to reduce vocabulary fragmentation across ISAs.

### Semantic Grouping

Instructions are grouped by functional category to preserve semantic signal.

### Cross-ISA Alignment

Embeddings align semantically equivalent instructions from different architectures.

Architecture Layer 2

# GRU + Self-Attention for Contextual Modeling

AMMF uses a GRU to capture sequential instruction dependencies within basic blocks. Hidden states then pass through self-attention to weight instructions by relevance to vulnerability patterns.

## GRU Temporal Modeling

Captures instruction ordering and flow within a basic block.

## Self-Attention Weighting

Learns which instructions are most discriminative for vulnerability detection.

## Long-Range Dependencies

Attention spans multiple basic blocks to capture distributed vulnerability logic.

Architecture Layer 3

# Attention-Augmented Convolutional Fusion

## The Fusion Challenge

Semantic embeddings alone are insufficient. Real-world vulnerabilities often manifest through **structural patterns** specific control-flow paths, function call patterns, and graph-level properties. AMMF fuses semantic and structural features through an attention-augmented convolutional architecture that learns which feature types are most informative for each function.

## Feature Streams Being Fused

01

### Semantic Stream

GRU + attention output from instruction-level analysis at the basic block level

02

### Control-Flow Stream

Structural attributes of the CFG node count, edge topology, loop depth, dominance relationships

03

### Function-Level Stream

Higher-order features: call graph position, argument count, return behavior, size metrics

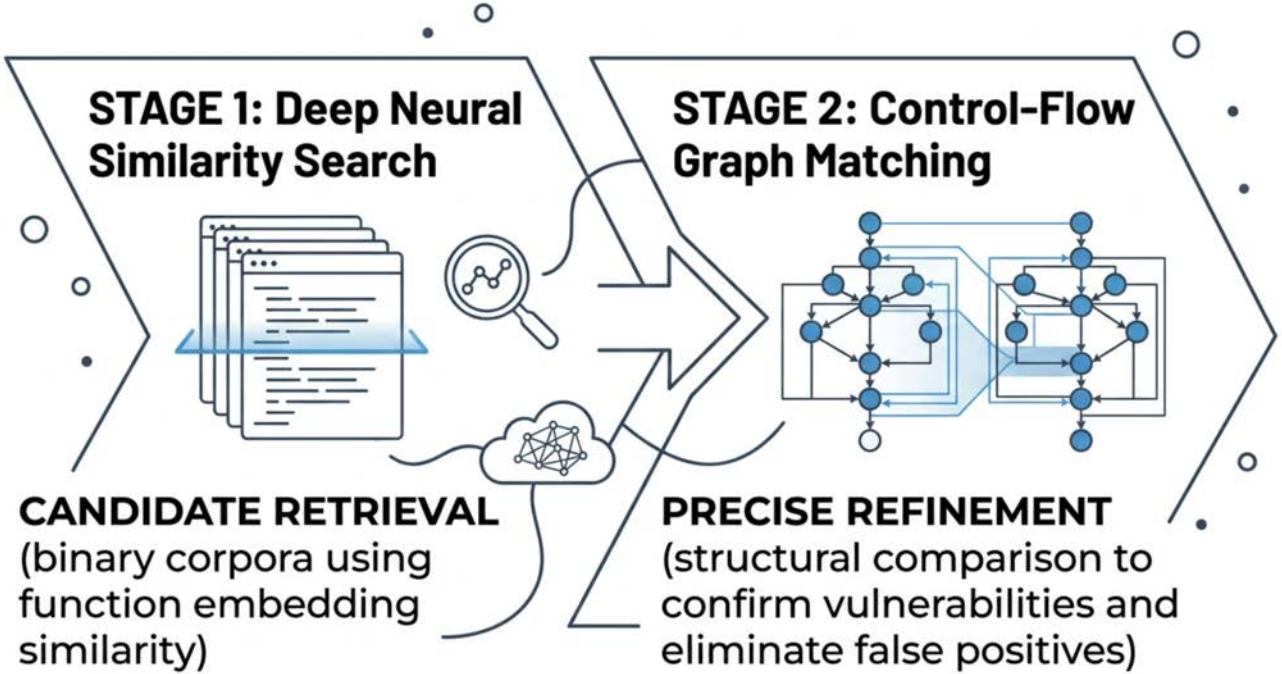
04

### Unified Embedding

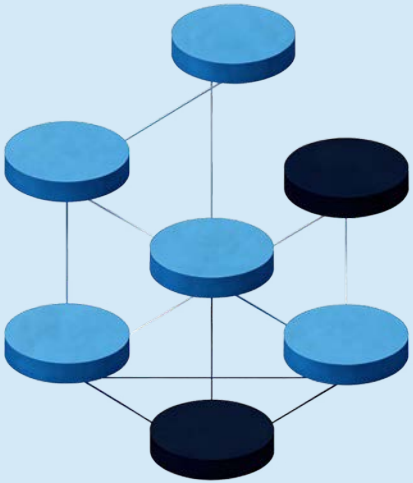
Attention weights determine how much each stream contributes to the final function representation

Detection Strategy

# Two-Stage Vulnerability Detection




Stage 1 uses approximate nearest-neighbor search over learned embeddings to quickly narrow a large binary corpus to a candidate set. Stage 2 applies precise CFG matching only to these candidates, reducing cost while improving precision. This coarse-to-fine design helps AMMF scale to real-world binary analysis.



Stage 1 Deep Dive

# Neural Similarity Search at Scale

## Embedding-Based Retrieval

AMMF encodes every function in the target binary corpus as a fixed-length embedding vector. Candidate vulnerable functions are identified by computing **similarity scores** between query embeddings (from known-vulnerable reference functions) and the corpus embeddings. Functions with high similarity are flagged for Stage 2 analysis.

## Why This Scales

- Embedding inference is parallelizable across functions
- Approximate nearest-neighbor search is sub-linear in corpus size
- No pairwise comparison needed dramatically reduces complexity
- Robust to architecture differences through learned semantic alignment

Stage 2 Deep Dive

# CFG Matching for Precision Refinement

## What Is CFG Matching?

Control-Flow Graph matching compares the structural topology of candidate functions against reference vulnerable functions examining basic block sequences, branching patterns, and edge relationships to confirm semantic equivalence.

## Why It Reduces False Positives

Embedding similarity may conflate structurally distinct functions that share surface-level semantics. CFG matching acts as a structural verifier, rejecting candidates that lack the specific control-flow signatures associated with the vulnerability.

## Efficiency Through Filtering

Because CFG matching is applied only to Stage 1 candidates not the entire corpus the computational overhead is bounded and practical for real-world deployment.

## The Precision-Recall Balance

Stage 1 is calibrated for **high recall** it is better to surface more candidates than to miss a true vulnerability. Stage 2 is calibrated for **high precision** it aggressively filters false positives to deliver actionable results to analysts. Together, the two stages achieve detection quality neither could accomplish alone, forming a complementary pipeline that mirrors how expert analysts perform manual binary review.

### Go Ecosystem Relevance

# Why AMMF Matters for Go-Based Systems

Go's first-class cross-compilation support `GOARCH`, `GOOS` makes it a natural fit for systems that must run on diverse hardware. But this portability creates a security analysis gap: a vulnerability in a Go library may be compiled into dozens of target binaries, each looking different at the assembly level. AMMF is purpose-built for exactly this threat model.

## Single Source, Many Targets

One Go module can produce x86, ARM64, MIPS, and RISC-V binaries from a single build command

## Transitive Dependency Risk

Go module dependencies carrying known CVEs appear in multiple compiled binaries across a deployment fleet

## Source-Free Analysis

AMMF operates purely at the binary level no source code required, enabling analysis of third-party and vendor binaries

Experimental Results

# Robustness Across Real-World Scenarios

AMMF was evaluated across three challenging cross-architecture scenarios that reflect real deployment conditions. Results demonstrate consistent detection capability across all axes of variation.

## Cross-Architecture

Reference functions compiled for x86-64 matched against ARM, MIPS, and other target architectures. AMMF's semantic embeddings successfully bridge ISA differences, correctly identifying functionally equivalent vulnerable functions.

## Cross-Compiler

Functions compiled with GCC vs. Clang produce distinct binary outputs. AMMF remains robust to compiler-induced structural variations by grounding detection in semantic rather than syntactic features.

## Cross-Optimization

Optimization levels from O0 through O3 dramatically restructure control flow and inline functions. AMMF's fusion of semantic and structural features provides stability across all optimization settings.

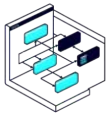
Strong accuracy and recall were achieved across all three evaluation dimensions, with meaningful reduction in false positives compared to prior approaches without relying on source code.

Key Takeaways

# What AMMF Means for Practitioners

## Attention Is the Unifying Force

Self-attention and attention-augmented convolution are not just performance improvements they are the mechanism that allows semantic and structural signals to be jointly weighted, making AMMF fundamentally more robust than single-feature approaches.



## Two Stages Beat One

Separating coarse similarity retrieval from precise CFG-based verification is the practical key to scaling binary vulnerability detection. High recall in Stage 1, high precision in Stage 2 each stage doing what it does best.

## Cross-Platform Languages Demand Cross-Architecture Security

As Go and similar languages power more infrastructure, binary-level security analysis must evolve to match. AMMF provides a concrete, deployable architecture for closing this gap in both internal audits and M&A due diligence contexts.

AMMF is available for further discussion explore integration into CI/CD pipelines, binary SCA workflows, and pre-acquisition security assessments.

# Thank you!!