# K-path CodeBERT as a Foundation Model for Vulnerability Detection

Abhinav Jain, aj70 (Group Name: Neurafix)

# Objective – Vulnerability Detection

- **Vulnerabilities**: points in the code where inadvertent programming errors or oversights create vulnerabilities that could be exploited by malicious actors

| Vulnerability types | Description |
|---|---|
| Memory Disclosure | A structure is copied to user space and not all fields or padding bytes are properly initialized resulting in memory disclosure. |
| Buffer Overflows | A length field involved in a copy operation is not checked resulting in buffer overflows. |
| Resource Leaks | A function creates a resource, but it is not destroyed on all error paths. |
| Design Errors | Program design does not sufficiently implement security policies. |
| Null Pointer Dereference | A pointer controlled by an attacker is dereferenced without checking whether it is null. |
| Missing Permission Checks | A security sensitive operation can be accessed without undergoing a prior permission check. |
| Race Conditions | Concurrently running processes cause various types of vulnerabilities. |
| Integer Overflows | A length field involved in a copy operation is checked, but the check is insufficient as integer overflows are not accounted for. |
| Division by Zero | An attacker-controlled value is a denominator in a division and it is allowed to be zero. |
| Use After Free | An allocated block of memory is used after being freed by the allocator. |
| Integer Type Vulnerabilities | A length field involved in a copy operation is checked, but the check is insufficient as the length field is a signed integer. |
| Insecure Arguments | Passing arguments to a function results in an implicit, insecure type cast. |

Vulnerabilities discovered in the Linux kernel

# Objective – Vulnerability Detection

- **Vulnerabilities**: points in the code where inadvertent programming errors or oversights create vulnerabilities that could be exploited by malicious actors
- Large Language Models (LLMs) are becoming increasingly popular for variety of code understanding and code analysis tasks

# Can you identify the vulnerability?

```
module locked_register( input [15:0] Data_in,
input clk, resetn, write, lock_status, debug_unlocked,
outputreg [15:0] Data_out ) ;
always @( posedge clk or negedge resetn ) begin
    if ( ~ resetn ) begin
        Data_out <= 16'h0000 ;
    end
    else if ( write &(~ lock_status | debug_unlocked )
        ) begin
        Data_out <= Data_in ;
    end
    else if (~write) begin
        Data_out <= Data_out ;
    end
end
endmodule
```

Verilog Code

# What does GPT-4 model has to say about this?

```verilog
module locked_register( input [15:0] Data_in,
input clk, resetn, write, lock_status, debug_unlocked,
outputreg [15:0] Data_out ) ;
always @( posedge clk or negedge resetn ) begin
    if ( ~ resetn ) begin
        Data_out <= 16'h0000 ;
    end
    else if ( write &(~ lock_status | debug_unlocked )
        ) begin
        Data_out <= Data_in ;
    end
    else if (~write) begin
        Data_out <= Data_out ;
    end
end
endmodule
```

Verilog Code

verilog Copy code

```verilog
else if (write & (~lock_status | debug_unlocked)) begin
    Data_out <= Data_in;
end
```

Here's the issue:

The condition `(~lock_status | debug_unlocked)` means that you can write to `Data_out` in two scenarios:

1. When the register is not locked (`~lock_status` is TRUE).
2. When `debug_unlocked` is TRUE, regardless of the value of `lock_status`.

This is a security concern because if the `debug_unlocked` signal is ever inadvertently or maliciously set to TRUE in a non-debug environment, the `locked_register` will effectively become unlocked, bypassing the `lock_status`. This can potentially allow unauthorized writes to the register.

**Possible Fix:**

To mitigate this issue, consider adding another debug-mode indicator in your system that must be TRUE for `debug_unlocked` to have any effect. Alternatively, you can remove the `debug_unlocked` feature altogether if it poses too high a security risk in the context of your application. If you do need a debug mode, ensure that it can't be enabled accidentally or without proper authority in the production environment.

# Drawbacks

- Currently, calling GPT based models is expensive to be used as Automatic Program Repair
- Other small-scale LLMs like CodeBERT can be used but they treat code as Natural Language
- Interesting code properties become implicit

# Approach

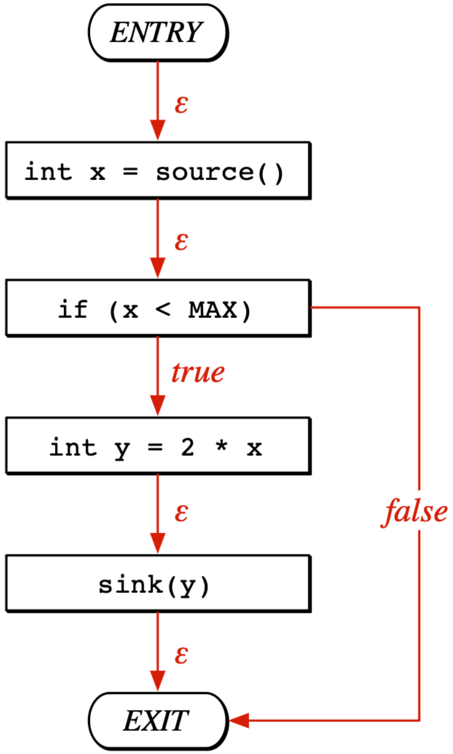
- Make code properties explicit to the LLM
- Which properties?
- Leverage widely available static analysis tools

# Approach

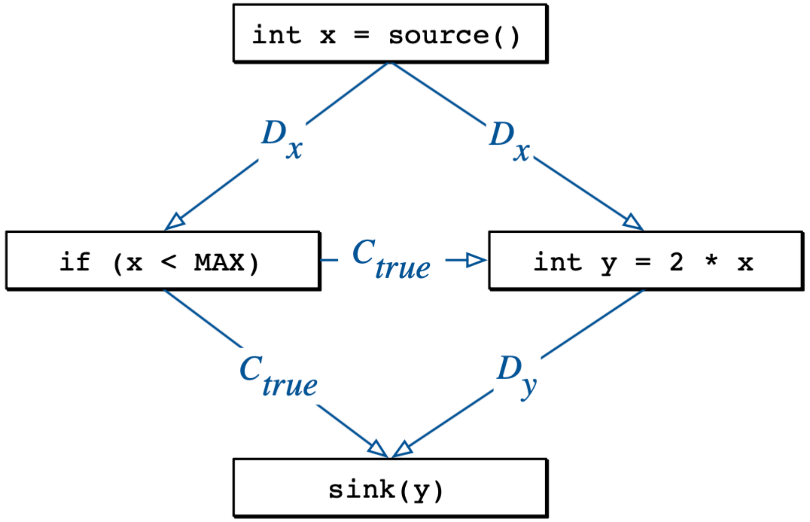
- Make code properties explicit to the model

```
void foo()
{
    int x = source();
    if (x < MAX)
    {
        int y = 2 * x;
        sink(y);
    }
}
```

Source Code




Control Flow Graph




Program Dependence Graph

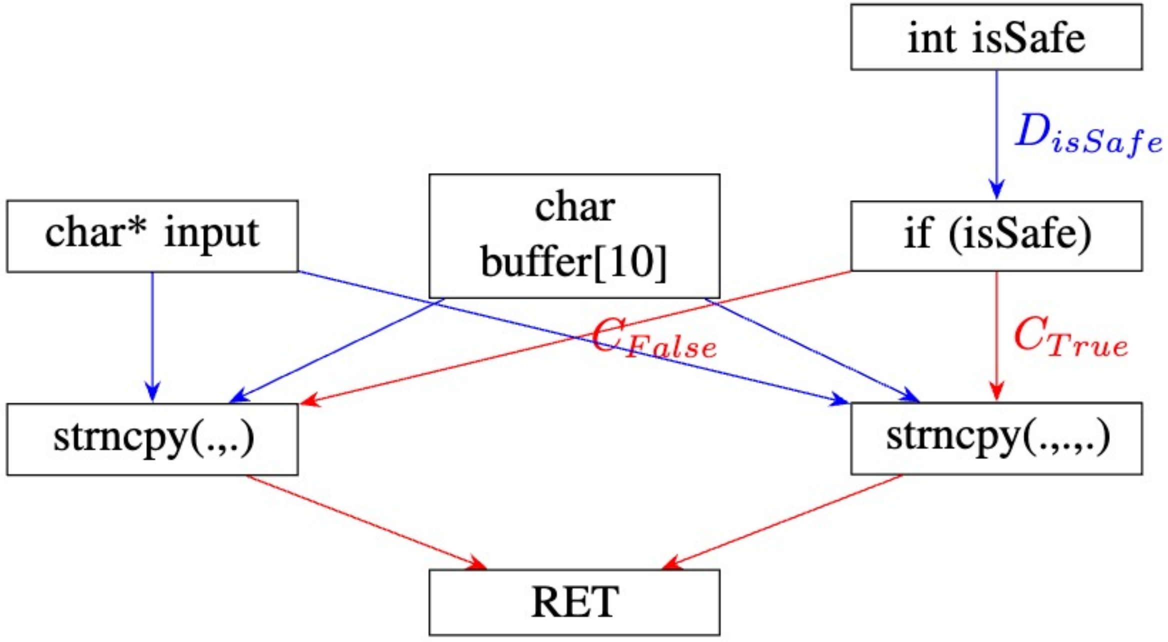
# Can code property graphs expose the underlying vulnerability?

## Can code property graphs expose the underlying vulnerability?

- In most cases, yes! Do path traversals.

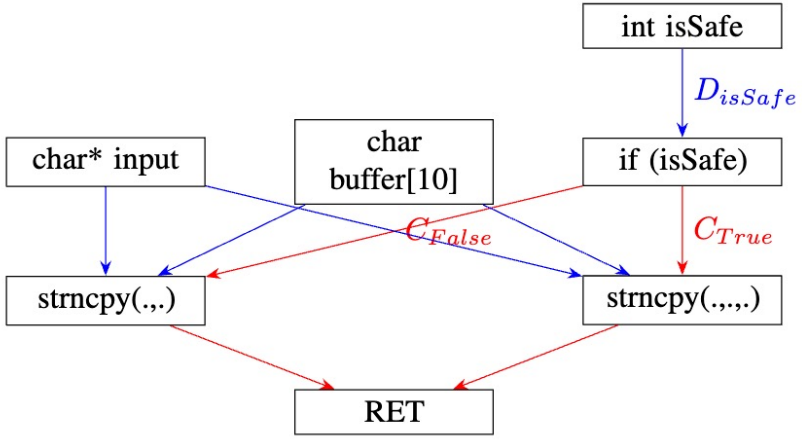
```c
void copyFunction(const char* input, int isSafe) {
    char buffer[10];

    if (isSafe) {
        // Safely copy input to buffer with bounds
            checking
        strncpy(buffer, input, sizeof(buffer) - 1);
        buffer[sizeof(buffer) - 1] = '\0';
        printf("Safe Copy: %s\n", buffer);
    } else {
        // No bounds checking, potential buffer
            overflow vulnerability
        strcpy(buffer, input);
        printf("Unsafe Copy: %s\n", buffer);
    }
}
```

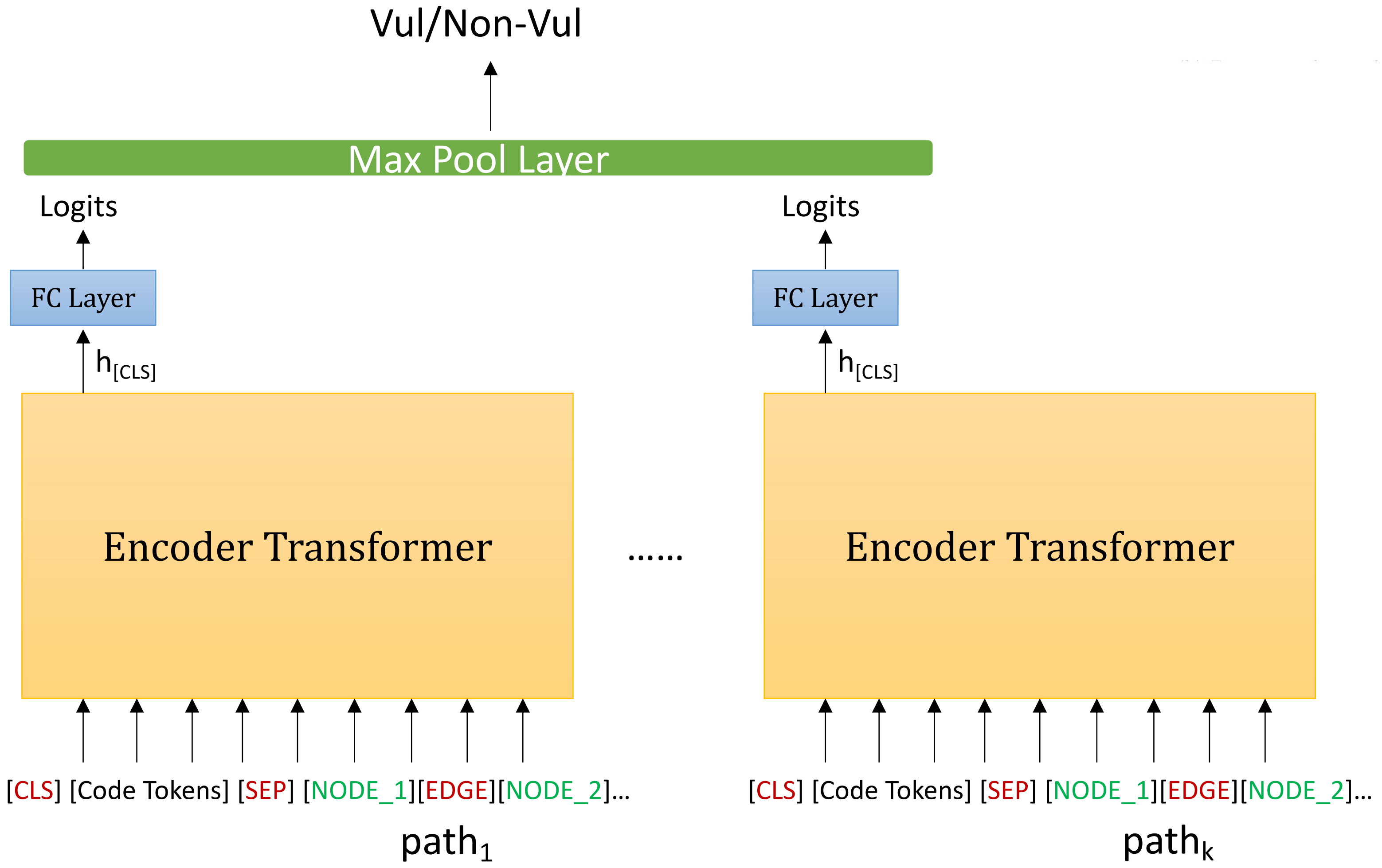
Vulnerable block of function
with buffer overflow




Program Dependence Graph

# K-Path CodeBERT



# Results: Baseline Comparison on Devign Dataset

| Model | Accuracy |
| --- | --- |
| CodeBERT | 61.79 |
| CodeBERT + CFG$(k = 1)$ | 62.19 |
| CodeBERT + CFG$(k = 2)$ | **63.18** |
| CodeBERT + PDG$(k = 1)$ | 62.66 |
| CodeBERT + PDG$(k = 2)$ | **62.81** |

# Results: Baseline Comparison on Devign Dataset

| Model | Accuracy |
|---|---|
| CodeBERT | 61.79 |
| CodeBERT + CFG$(k = 1)$ | 62.19 |
| CodeBERT + CFG$(k = 2)$ | **63.18** |
| CodeBERT + PDG$(k = 1)$ | 62.66 |
| CodeBERT + PDG$(k = 2)$ | **62.81** |

**True Positives Profile** (total vulnerable samples in the test set 1255):
- CFG: 53.23%
- PDG: 43.98%
- Agreements: 36.89%
- Unique to CFG: 16.33%
- Unique to PDG: 7.09%

Both models look at same source code but different code properties, end up detecting unique vulnerabilities.

# Results: Baseline Comparison on Devign Dataset

| Model | Accuracy |
|---|---|
| CodeBERT | 61.79 |
| CodeBERT + CFG$(k = 1)$ | 62.19 |
| CodeBERT + CFG$(k = 2)$ | **63.18** |
| CodeBERT + PDG$(k = 1)$ | 62.66 |
| CodeBERT + PDG$(k = 2)$ | **62.81** |

What about a **Mixture-Of-Expert** like model?
- Precision of CodeBERT + CFG (k=2) : 53.23
- Precision of CodeBERT + PDG (k=2) : 43.98
- Precision of MOE: 60.32

# Results: Zero-Shot Test with $k_{test}$ >= $k_{train}$ paths

| k-path Model | $k_{test} = 2$ | $k_{test} = 4$ | $k_{test} = 8$ |
|---|---|---|---|
| CFG ($k_{train} = 1$) | 62.52 | 62.59 | **62.81** |
| CFG ($k_{train} = 2$) | 63.18 | 63.32 | **63.51** |
| PDG ($k_{train} = 1$) | 62.30 | 62.26 | **62.45** |
| PDG ($k_{train} = 2$) | 62.81 | 62.81 | **63.14** |

# Ablation: Path-only Model?

| Code Property | $k_{train} = 1$ | $k_{train} = 2$ | $k_{train} = 4$ |
|---|---|---|---|
| CFG | 60.54 | 60.61 | **61.68** |
| PDG | 57.61 | 58.05 | **59.85** |

# Conclusion

- Proposed approach systematically encodes both code semantics and structural properties,
  - leveraging LLMs for nuanced code interpretation and
  - CPGs for capturing inherent code structures
- Varying path numbers during evaluation further revealed the model’s resilience
- Findings highlight the complementary strengths of LLMs and CPGs, underscoring the significance of a hybrid approach for comprehensive vulnerability detection in software systems

# Future Work

- Smarter path selection strategies, moving beyond the simplistic consideration of shorter paths to potentially predicting program slices as candidate proposals
- Extending our k-path CodeBERT as a foundation model for unsupervised clustering of paths, aiming to identify **nefarious** outliers that deviate from system-wide invariant properties

# Questions?