Proposed Methodology

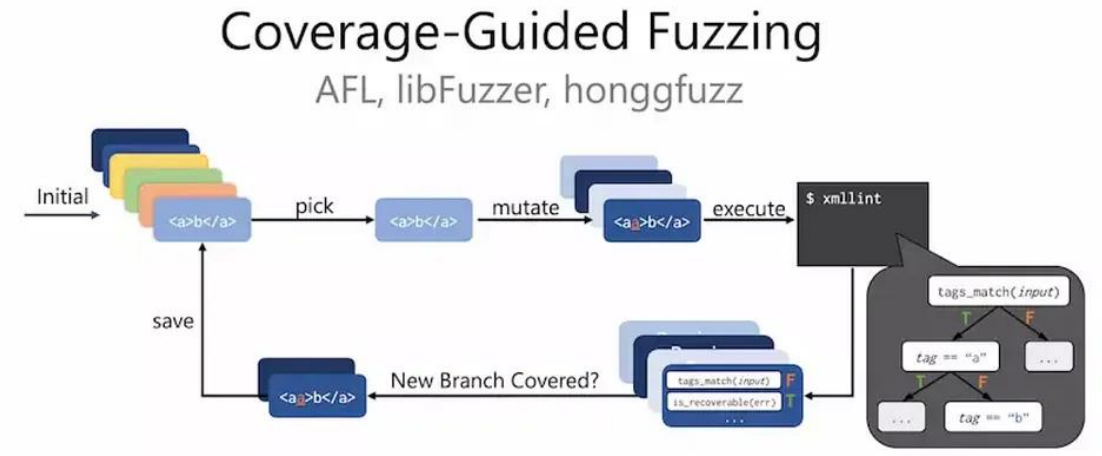
(Credits: Gaby's message)

1. Code Acquisition:

- Our objective is to look for vulnerabilities in a code so we obviously need to acquire the source code to analyze somewhere.
- We either get it from GitHub at a specific commit hash(which is just an identifier for a code at a point in time) or by ingesting a local firmware (which in simple terms means we directly get the code that was written into a hardware).
- Every one of those code files would have the metadata like if it was meant for x86 or ARM arch systems, file path, etc.

2. Fuzzing:

- Gaby said in parallel but I doubt that. So, I'll assert that it's semi-parallel. Anyway, we know that fuzzing is feeding random inputs to test a code. Coverage guided fuzzing is like that except we take feedback. For eg: If I input "ab" and I figure that a specific block is being executed wrongly, coverage guided fuzzing mutates "ab" -> "abc" and tries again to find more. It's like permutations in DSA in a way



- We need to use a library for coverage guided fuzzing which is where AFL++ and libFuzzer comes in.

Note: libFuzzer is more suited for our application. Let's stick to libFuzzer to cut down on the noise.

1. AFL (American Fuzzy Lop)

- Uses compile-time instrumentation
- Known for exceptional speed and mutation strategies

2. libFuzzer

- In-process fuzzer
- Works with LLVM Sanitizers
- Extremely effective for C/C++

3. Sanitization:

- C/C++ compilers such as GCC, Clang, LLVM, etc. has a powerful feature called sanitization which detects memory access errors. When we compile our code with instrumentation flags, these are the intentions:

- **fsanitize = address:** Detects bugs that pose threat to memory safety (eg: buffer overflows, use-after-free, invalid free, memory leaks).
- **-fsanitize-coverage = trace-pc:** Records which code paths execute by tracing program counters**.** Very useful for coverage guided fuzzing

We did all this by feeding inputs to the code and that context should obviously be recorded. So for each input we executed during fuzzing, we store coverage bitmap (what code it covered) and runtime crash signature (what crashed). This directly maps to the instrumentation flags we would've used earlier.

4. Intermediate Representation:

clang -emit-llvm -S -O0

- If we type this in the compiler, it produces LLVM IR which is the best choice for understanding memory access and control flows in C/C++ code. LLVM IR is produced as a .ll file.

Note: https://shorturl.at/Eu49g **-** More detail on why LLVM is optimal for our use case

opt -dot-cfg program.ll

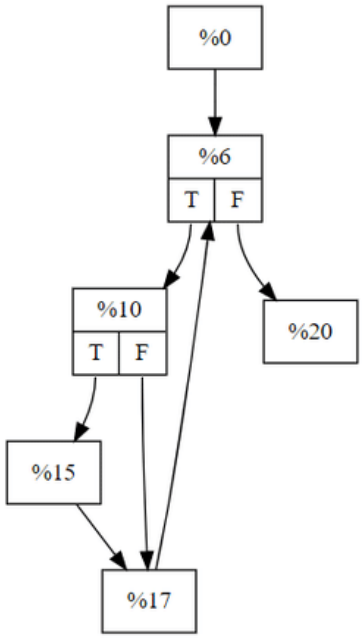
- This command is used to model a CFG graph where nodes = basic block and edge = control flow

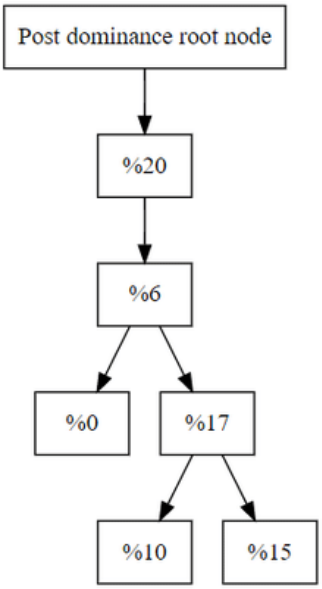
```
int findmax()
{
        int i=0, n;
        std::cin>>n;

        int max=0;
        while(i<n)
        {
                int a;
                std::cin>>a;
                if(a>max)
                {
                        max=a;
                }
                i++;

        }

        return max;
}
```





Control Flow Graph

- We load this graph into NetworkX to preserve it

5. Static Analysis:

- In CVD-MAX, static analysis is done on the CFG graphs from the above step
- We simply the CFG graph by merging linear blocks and removing unreachable blocks. This reduces the graph's depth and complexity while still preserving the necessary information
- The remaining nodes are annotated with numeric features such as memory read/writes, pointer arithmetic, function calls, etc.
- These nodes are fed into GCN or GAT via pytorch arithmetic

Note: GAT is better suited for our needs because GCN treats all nodes equally. In order to find vulnerabilities, we need to distinguish critical blocks/branches. Hence Graph Attention Network(GAT)

- Function here means a program function (eg: main). For each function, the GAT produces a fixed length vector embedding that contains all its control characteristics we obtained from the previous steps (such as what memory was accessed, where read/written, dangerous paths, etc.)

6. Dynamic Analysis:

- While static analysis tells us what could happen, dynamic analysis actually shows us what happens in a runtime.
- For that we use a sandboxed system
- We employ fuzzing here too and feed the code meaningless, mutated inputs
- Real time behavior of the code, such as memory access and control flow, is recorded through Dynamic Binary Instrumentation (DBI)

- Dynamic Binary Instrumentation (DBI): To put it simply, it injects custom code into the execution stream (note that it does not inject it into the source code), without modifying the source code to analyze and monitor how the program behaves during runtime. It captures signals such as:

    - Memory allocations/deallocations
    - Memory read/write addresses
    - Function entry/exit
    - System calls

- It produces a chronological log of the runtime events where each event is structured as (type + parameter + sequence index)

- Valgrin, Intel PIN, DynamoRIO are DBI tools

Note: Again, let's cut down on the noise and too much ambiguity in the tools. My suggestion is Docker + QEMU. Reason being firmware ingestion is too difficult and has greater overhead. So it's easier to opt for Docker.

Docker vs QEMU

Feature	Docker	QEMU
Technology Type	Containerization (OS-level virtualization)	Full system emulator and virtualizer
Resource Usage	Lightweight, shares host kernel, efficient resource use	Heavier, runs a full guest OS, requires more resources
Boot Time	Very fast (seconds or less)	Slower (minutes for a full OS boot)
Isolation Level	Process isolation (less secure than VMs)	Hardware-level isolation (more secure)
Cross-Architecture	Requires QEMU internally or specific host support for non-native builds	Can emulate different CPU architectures natively
Primary Goal	Package and deploy applications consistently and efficiently	Run full operating systems on different hardware or architectures

Valgrind vs Intel PIN vs DynamoRIO

- **Choose Valgrind** if you want to find memory vulnerabilities in a program **today**.
- **Choose** Intel Pin if you want to **write your own logic** to monitor program behavior with the easiest API.
- **Choose** DynamoRIO if you want to integrate with **fuzzers** or build a tool that requires **maximum speed**.

Feature	Valgrind	Intel Pin	DynamoRIO
Primary Goal	Rich, in-depth analysis (e.g., memory errors)	Flexible, high-performance custom analysis	Efficient, transparent runtime code manipulation
Performance	Slower (typically 4x–20x slowdown)	Faster (minimal overhead, often a few percent)	Faster (minimal to medium overhead)
Tool Development	Easier API, built-in tools (Memcheck, Callgrind, etc.)	High-level API with callbacks	Lower-level API, powerful but more complex
Platform Support	Primarily Linux, some macOS support	Linux, Windows, macOS (x86/x64)	Linux, Windows, macOS (x86/x64, ARM)
Instrumentation	Runs code on a synthetic CPU (simulated)	Uses dynamic recompilation (JIT) and a code cache	Uses dynamic recompilation (JIT) and a code cache
Open Source?	Yes, open source	No, proprietary (developed by Intel)	Yes, open source (BSD license)

- Dynamic Feature extraction just converts these traces we got from the earlier step and encodes them
- Since the data obtained from dynamic analysis is chronological and because runtime sequence matters, we need a temporal learning model that captures the vulnerability patterns that depend on the event ordering

Note: 1D-CNN is a better choice than LSTM because the latter is unnecessarily hard and complex. We also don't have a long term dependency in our use case.

Yeah, LSTMs are just a hellish nightmare to train. Not only are they large and slow, they're also unstable. Yay. CNNs are just significantly easier to train. Even when you "need" an LSTM, its often wise to use Conv layers to reduce sequence length so the LSTM portion can be as lightweight as possible.

1D Convolutional Neural Networks (1D-CNN) and Long Short-Term Memory (LSTM) networks excel at sequential data, but **1D-CNNs are generally faster and better at extracting local, short-term, or noise-sensitive features**. LSTMs excel at capturing long-term temporal dependencies. For optimal results, a hybrid CNN-LSTM model is often used, where 1D-CNN extracts features and LSTM models dependencies.

7. Multi-head Attention:

- Static analysis gives structural evidence; Dynamic analysis provides behavioral evidence.

Why multi-head attention?

- If we give fixed weights to both like, structure matters 50% and behavior matters 50%, it might lead to false positives. Because in cases like buffer overflow, CFG looks normal, but runtime behavior is suspicious. So, the model learns where to focus on, right now, instead of fixing it overall.
- This outputs a feature vector that captures everything we've learned so far

8. Classification:

- The feature vector is passed to an FCN classifier
- It answers two questions:
    - Is this code vulnerable? **Yes/No ->** Binary output
    - What known vulnerability does it map to? **CWE ID ->** Categorial output