Using Graph-Based Machine Learning Algorithms for Software Analysis

ITEA Cybersecurity Workshop

Michael D. Brown

31 August 2023

Software Analysis

Goal: Automatically determine facts about a program's properties and behaviors.

Used extensively in compilers, security, and reverse engineering.

Must make trade-offs - impossible to collect a complete set of program facts in general / non-trivial cases.

Software Analysis

Many core problems cannot be solved deterministically:

- Phase ordering
- Precise binary decompilation
- Declaring software vulnerability free

SoTA tools employ heuristics and / or rely on humans

Meaningful gains are few and far between despite sizable research investments.

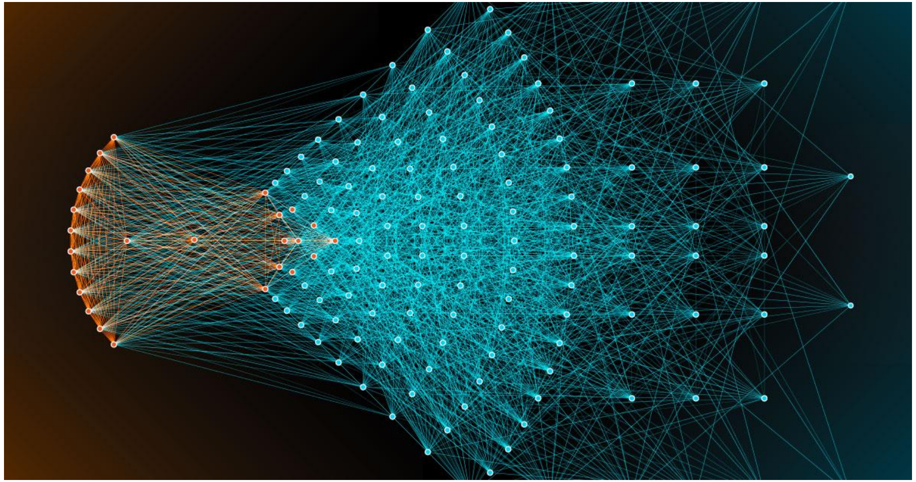
Using ML Techniques for Software Analysis

Advances can be made via AI/ML:

- AI/ML not bound by the constraints of traditional software analysis

- Approximates human probl

  solving on fuzzy tasks



Using ML Techniques for Software Analysis

Challenges:

- How do we represent software in a way that AI/ML techniques can ingest?
- What is the right program representation to use?

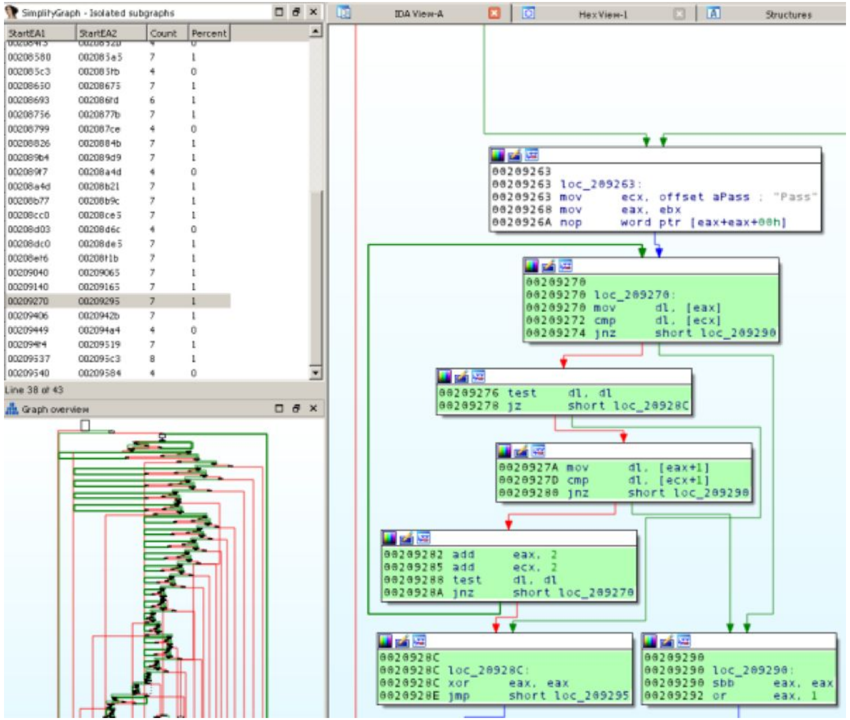
.... and Pitfalls:

- Easy to apply ML to unsuitable problems (soundness)
- Can we get enough data?

Key Insights

1) How do we represent software in a way that these techniques can ingest?

Programs are inherently graph-like, so use existing graph-based ML algorithms



Key Insights

2) What is the right program representation to use?

Depends on the application!

We can use compiler / decompiler tools to convert software to the right representation for our problem.

Key Insights

3) What problems are suitable for ML-based software analysis?

ML systems cannot be expected to be 100% accurate:
<u>DON'T</u> **use them when soundness is required!**

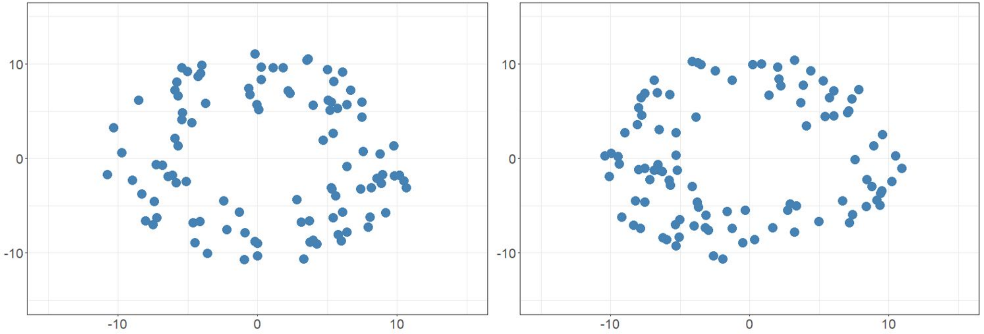
Useful for many security and reversing applications – tolerant of false positives.

Applications

4) Can we get enough data?

Real world data is hard to find in volume, but...

New automated program generation tools and benchmarking datasets makes creating quality <u>synthetic datasets</u> realistic.



Original data Synthetic data

The synthetic data retains the structure of the original data but is not the same

Applications

Two recent successes using graph-based ML over the last several years

1. VulChecker: Scans source code for vulnerabilities

2. CORBIN: Recover symbolic mathematics from binaries

Both tools developed under funding from DARPA I2O

VulChecker

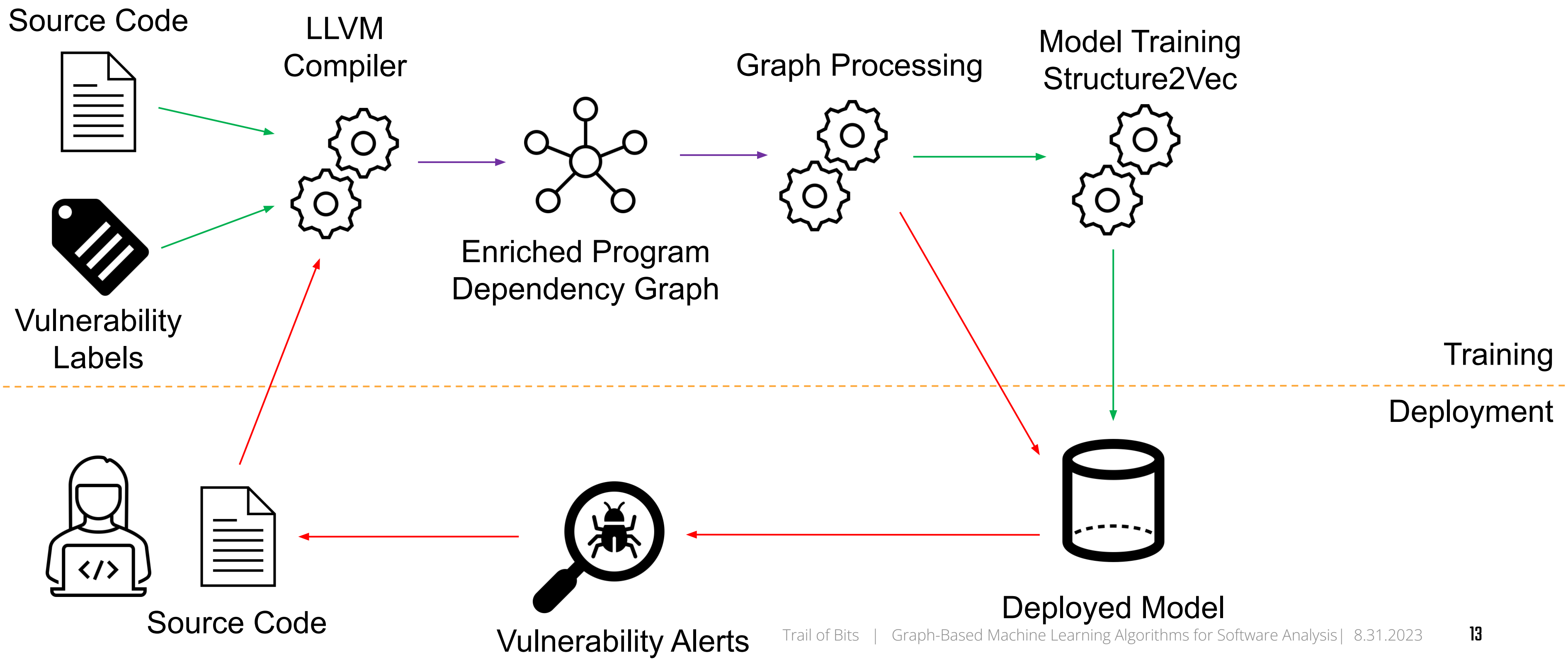
VulChecker

Problem: Certain types of high-risk vulnerabilities are difficult for traditional code scanners to discover. Can ML-based systems do better?

Is this a good problem for ML? – Yes

- No requirement for soundness – existing code scanning workflows produce false positives
- Use existing benchmark datasets as training data

VulChecker Overview



VulChecker Data Strategy

Bootstrap model with NIST Juliet dataset, supplement with as many real-world samples as possible.

Juliet Dataset

- Low Fidelity – Programs are synthetic "toys"
- Low Effort – Programs are labelled with in-line comments, very straightforward to harvest
- High Contrast – Malicious and benign versions of each example
- High Volume - Thousands / CWE

Samples from CVE database

- High Fidelity – Real-world samples in complex programs
- High Effort – Engineering required to localize and scrape samples
- Low Contrast – CVE databases don't include references to patched code
- Low Volume – Dozens at best

VulChecker Data Strategy

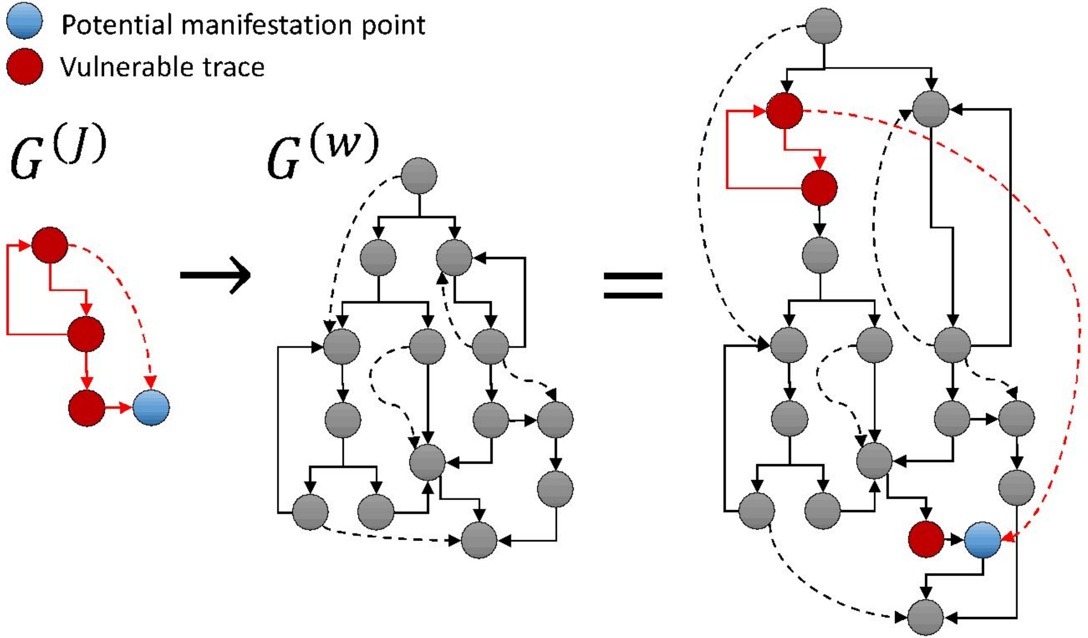
Improve synthetic sample fidelity via augmentation with real-world structures

<u>**Augmentation procedure**</u>

1. Inject nodes from synthetic samples into benign code

2. Adjust edges to maintain control-flow and data-flow integrity

<u>Note</u>: Properties of the graph processing ensure non-interference for dataflows

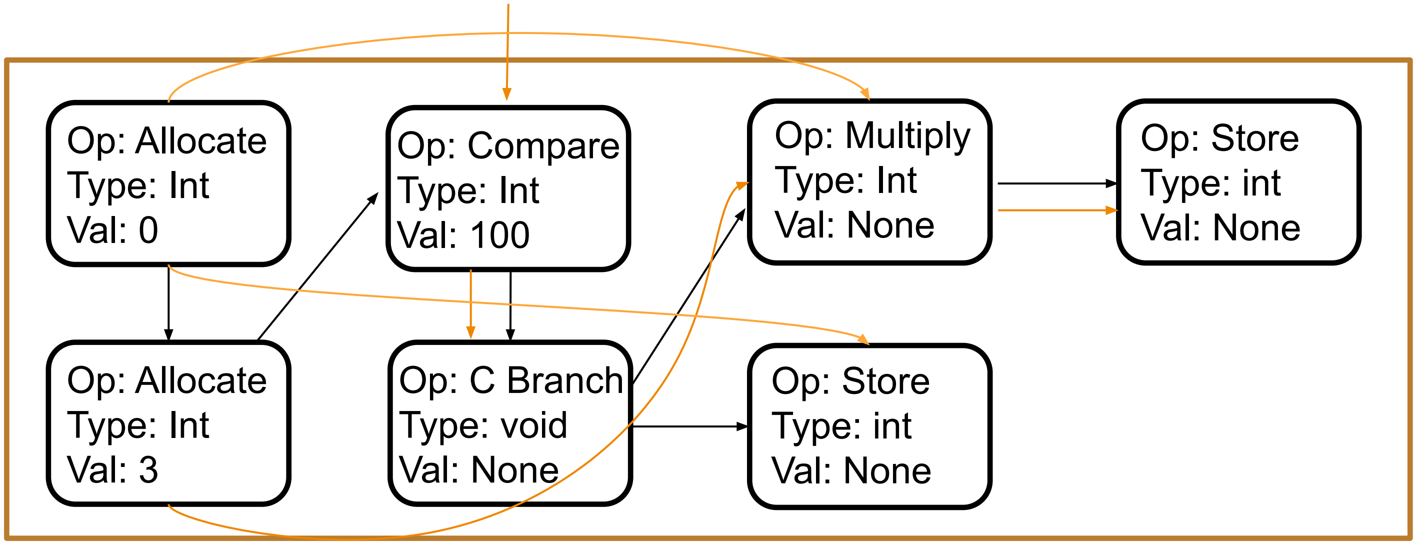


VulChecker Data Processing

Use compiler infrastructure to convert source code to simplified enriched graph representation (ePDG)

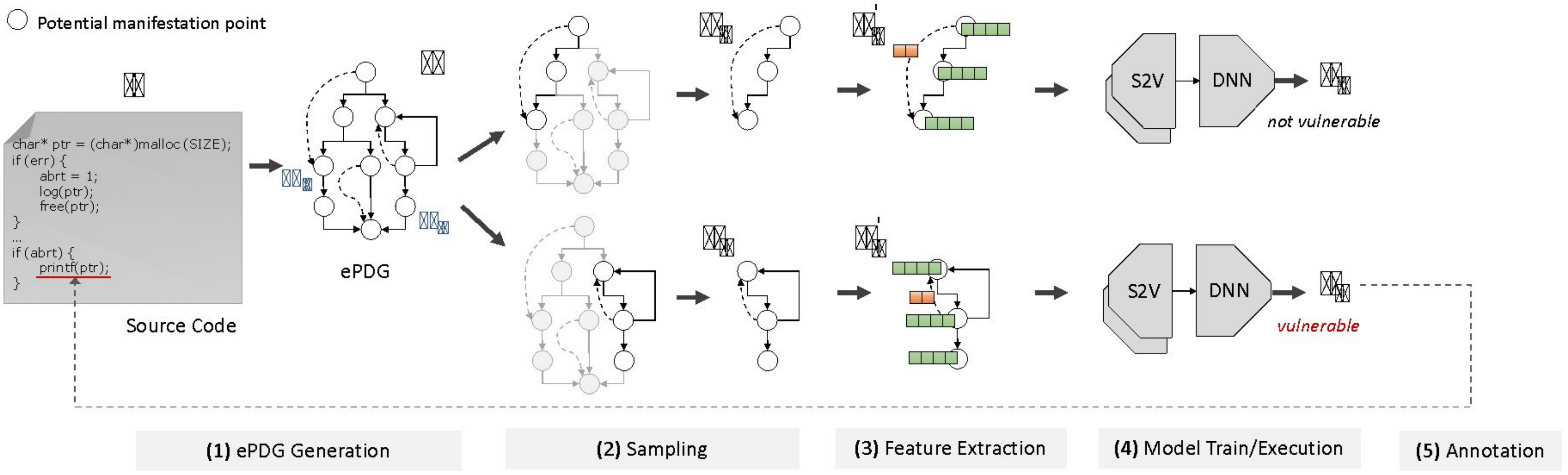
```
int a = 0;
int b = 3;

if(c > 100){
    b = b * a;
} else {
    b = a;
}
```



VulChecker Training and Deployment

From larger graph structure, extract sub paths and classify as vulnerable or not vulnerable.

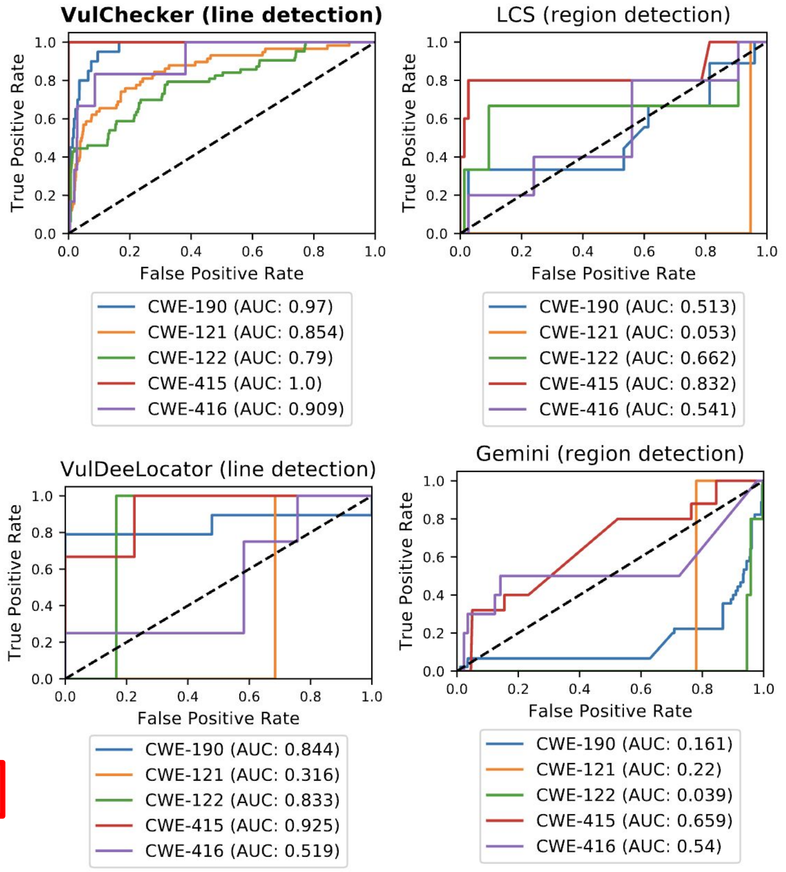


VulChecker Evaluation

Compared VulChecker against 4 other ML tools and commercial SAST tool across 5 CWEs

(Train – Augmented data, Test – RW data)

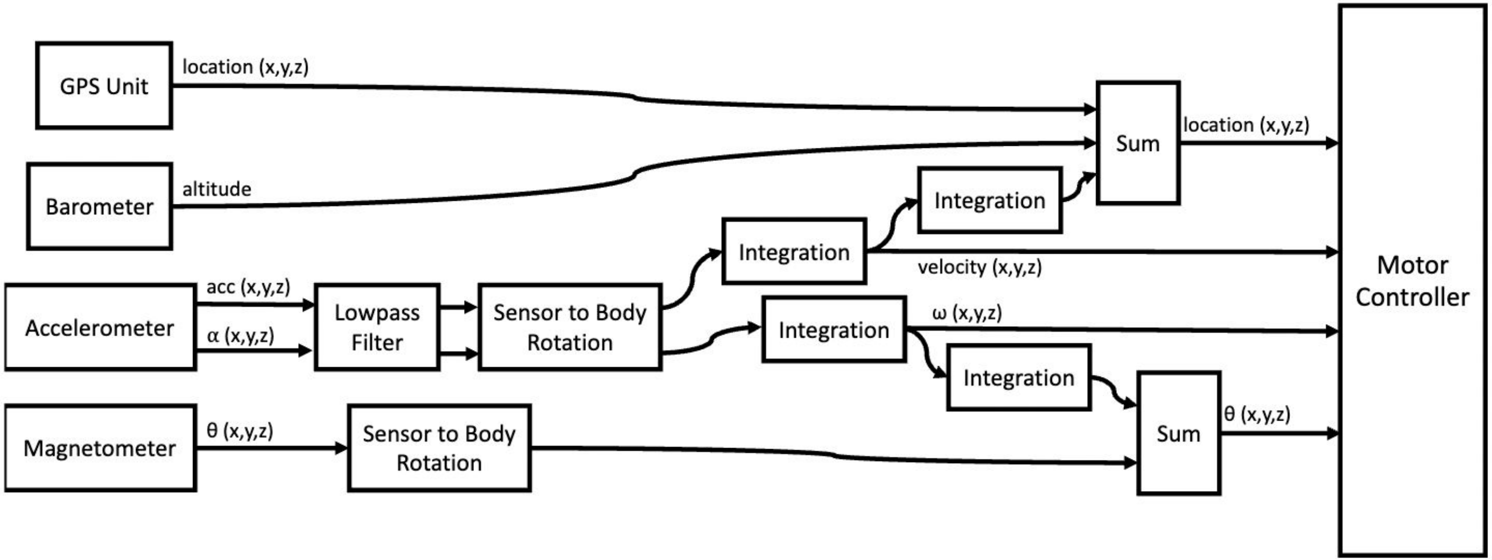
CWE	VulChecker @ FPR 0.05 Lines TP	VulChecker @ FPR 0.05 Lines FP	VulChecker @ FPR 0.05 CVEs TP	VulChecker @ FPR 0.1 Lines TP	VulChecker @ FPR 0.1 Lines FP	VulChecker @ FPR 0.1 CVEs TP	Helix QAC Lines TP	Helix QAC Lines FP	Helix QAC CVEs TP
190	9	55	3	12	112	6	1	2	1
121	7	33	7	9	112	9	4	230	1
122	1	6	1	1	6	1	4	241	1
415	3	0	2	3	0	2	0	5	0
416	4	6	4	6	228	6	0	0	1
Total	**24**	**100**	**17**	**31**	**458**	**24**	**9**	**478**	**4**



CORBIN

CORBIN

<u>Problem</u>: Legacy CPS need updates to improve performance or safety. Source code not available to patch. If we can recover control loops we can re-implement firmware easily.



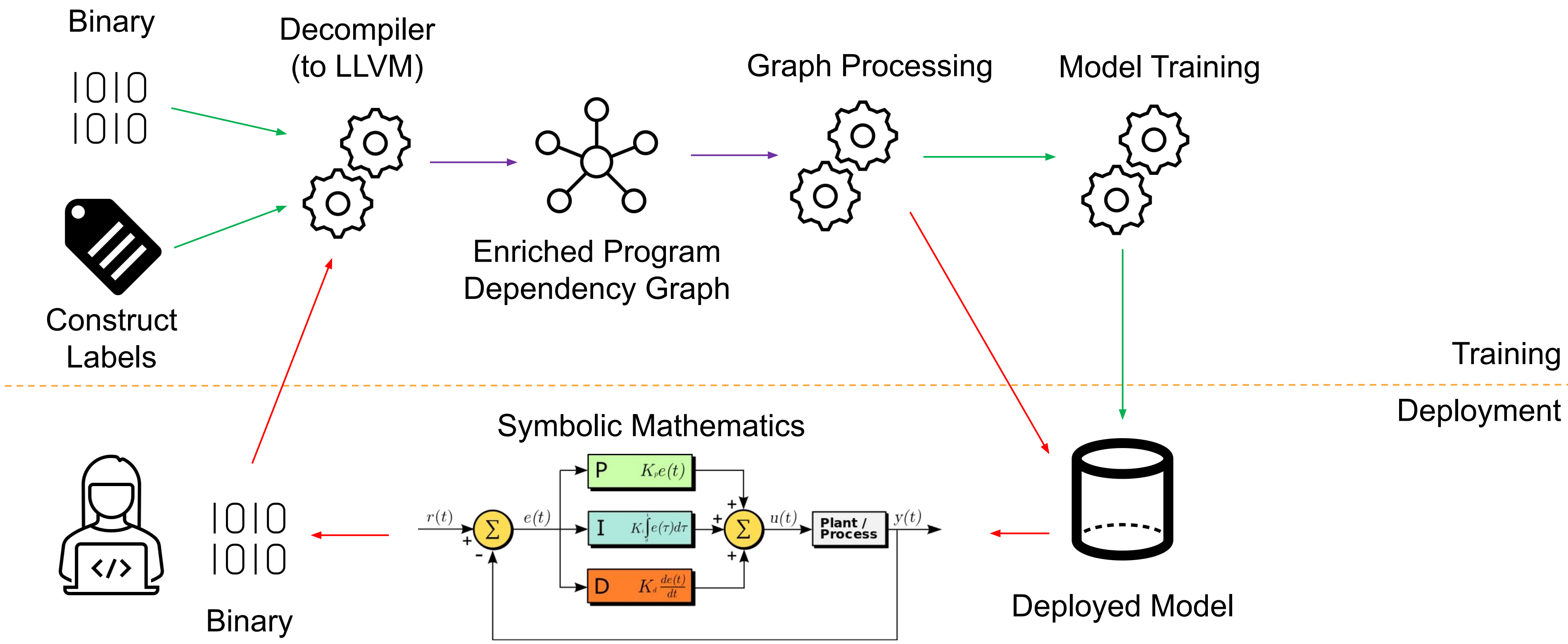
CORBIN

Is this a good problem for ML? – Yes

- **No requirement for soundness – reverse engineering workflows are tolerant of errors, recovered code won't be used blindly.**

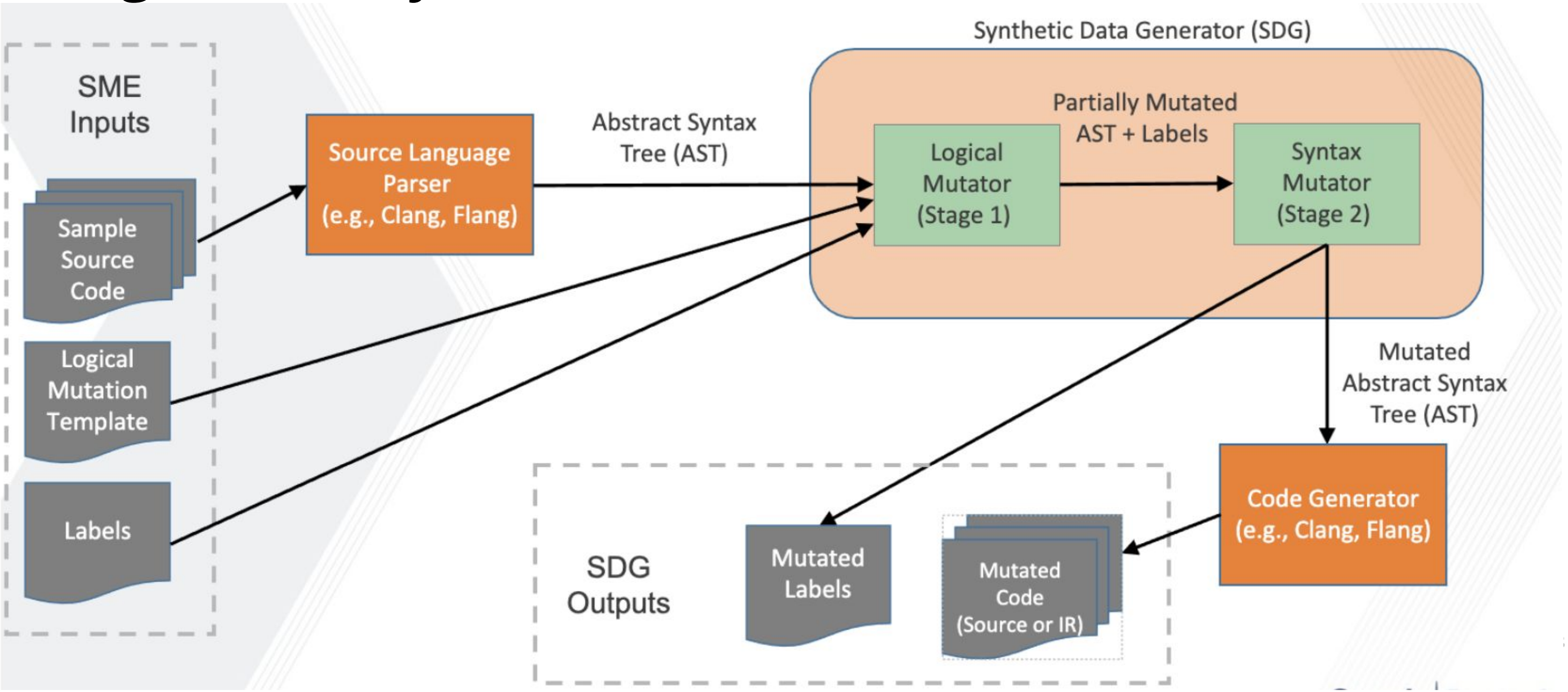
- **Can generate synthetic data sets for math constructs easily**
  - Diversity and complexity are open problems, however

CORBIN Overview



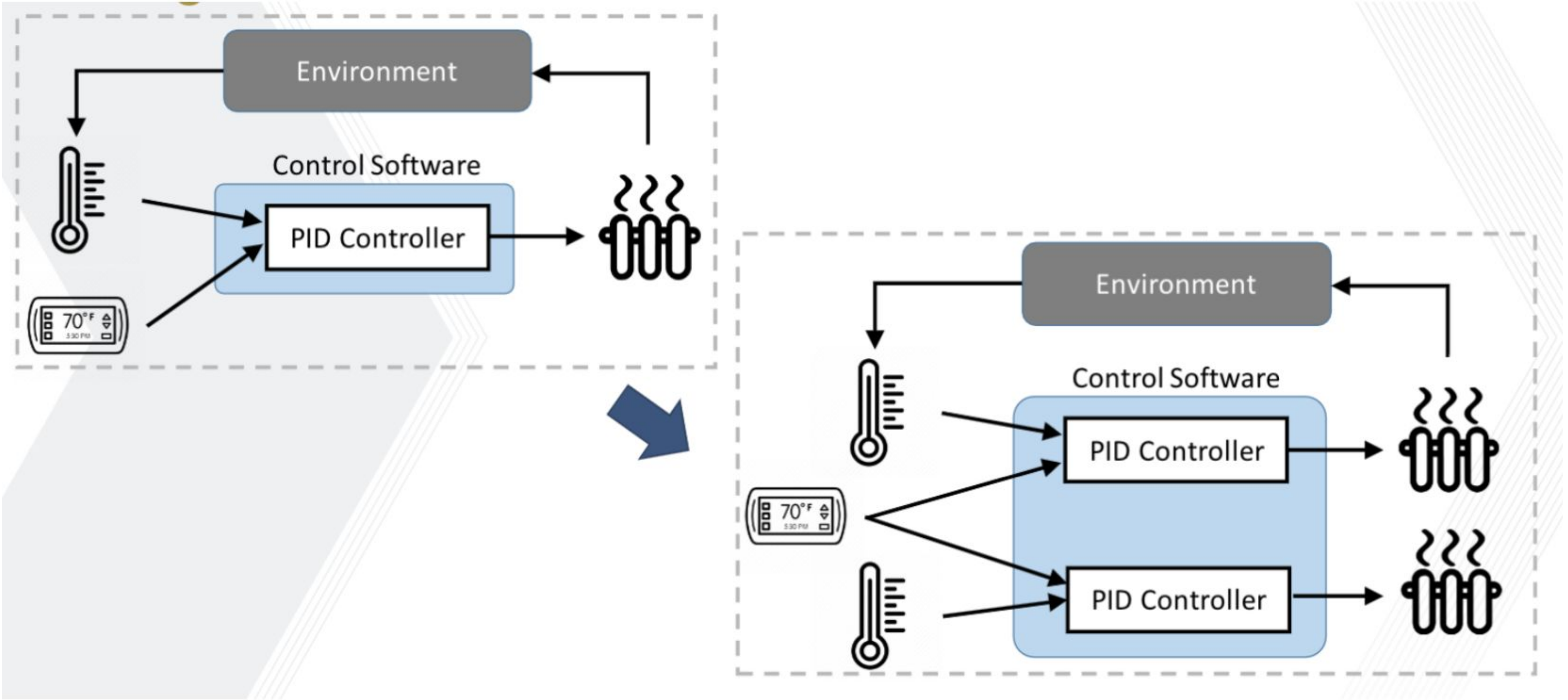
CORBIN Data Strategy

Amplify small volume of real-world / SME derived samples with logical and syntactical mutations



CORBIN Data Strategy

Logically mutate single zone to multi-zone controller



CORBIN Data Strategy

Syntactic mutation: capture programmer induced variance

```
for (int i=0; i<10; ++i){

// Do Something

}
```

```
int i = 0;
while (i<10){

// Do Something

++i;
}
```

```
if(a > b){
    // Do Something
}
else{
    // Do Something Else
}
```

```
if(b <= a){
    // Do Something Else
}
else if(a > b){
    // Do Something
}
```

CORBIN Data Processing

Uses same base approach as VulChecker, with domain-specific improvements.

Many complex mathematical functions are handled by libraries:

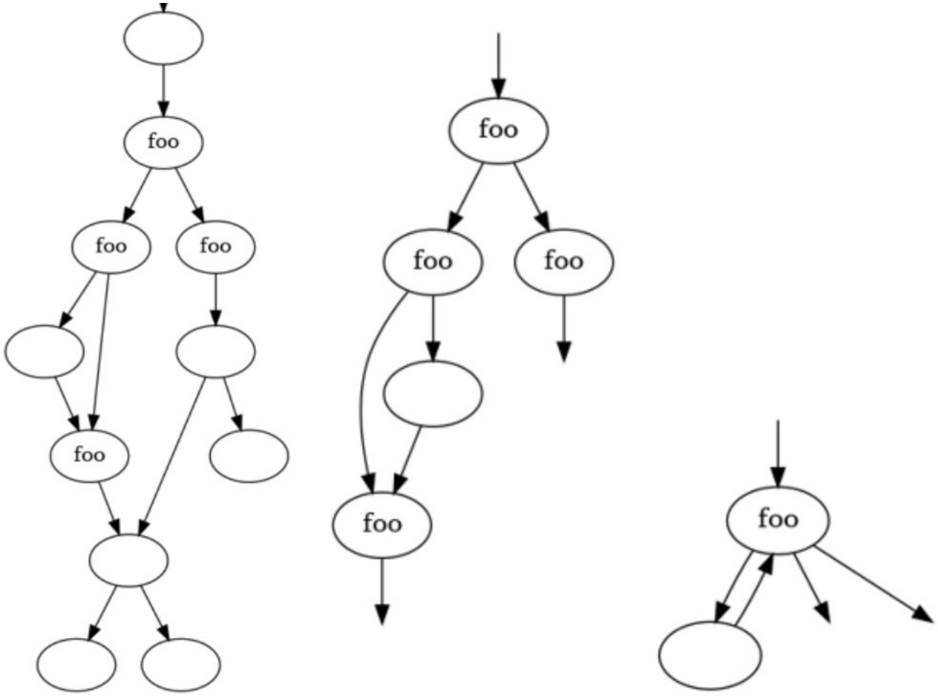
- Libm
- Lapack

In ePDG these are function call nodes – we reduce them to atomic math operations

CORBIN Training and Deployment

Graph to Graph approach

- match subgraphs / nodes corresponding to mathematical constructs
- condense to symbolic representation



CORBIN Results

On synthetic data holdout set:

- **Strong performance across all trained constructs**
  - To be expected – same production procedure from source

On autopilot software:

- **Many misclassifications, some limited success**
  - Limitations largely due to imprecise binary to LLVM IR lifting

Conclusion: Approach is viable – but training on source code derived samples did not transfer to binary derived samples.

Key Takeaways

Key Takeaways

1. **Problem formulation is important - our successes relied on focused feature selection.**
   - Unlikely to find success directly applying models (including LLMs!)

2. **ML approaches supplement, not replace, traditional (i.e., algorithmic) approaches.**
   - Prioritize problems that rely on human expertise

3. **Make synthetic data as real as possible for good results!**

Contact

Michael D. Brown

Principal Security Engineer

michael.brown@trailofbits.com

References/Links

VulChecker Paper

https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky

VulChecker @ Github

https://github.com/ymirsky/VulChecker