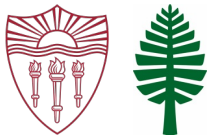
Evaluating Machine Learning-Based Binary Vulnerability Detectors: Lessons From the Trenches

Sima Arasteh · Jelena Mirkovic
Mukund Raghothaman · Christophe Hauser



University of Southern California
Dartmouth College

Why Binary Vulnerability Discovery?

- Lots of research on detecting vulnerabilities in source code
- **But (#1):** Source code is frequently unavailable
- **But (#2):** Source code might not reflect binary semantics

Why Machine Learning based Approaches ?

- Signature based approaches rely on hand-crafted Rules

Challenges:

- Composing signatures is labor-intensive
- Sensitive to compiler flags and source code modification

- Machine learning performs well across various domains

Challenges:

- Neural Network works with vectors
- How to transform vulnerable program to vector embeddings?
- How to make the classifier aware of vulnerability instead of function semantic ?

Why am I giving this talk?

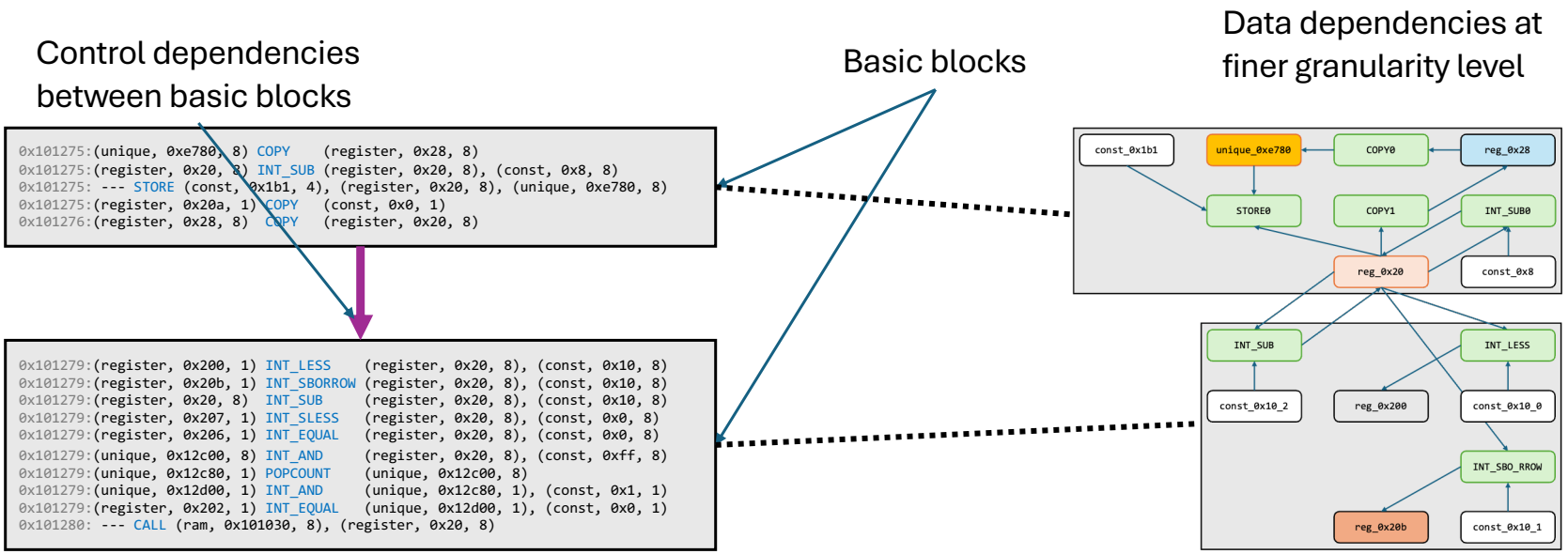
- Bin2vec: learning representations of binary executable programs for security tasks. (Arakelyan et al, Cybersecurity journal 2021)

- My talk on Friday: BinHunter**:** A Fine-Grained Graph Representation for Localizing Vulnerabilities in Binary Executables. (ACSAC 2024)

Session: Application Security
Time: 8:45-9 AM Friday 13 Dec

How BinHunter works?

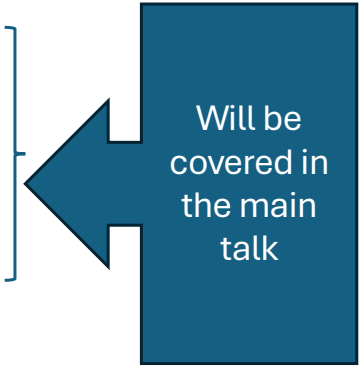
- Automatic vulnerability Discovery tool using Deep Learning
- Represent vulnerable code segments of a function as a graph Representation
- Localize vulnerability within functions



Machine learning based vulnerability discovery

Challenges:

- Requires appropriate dataset
- Requires a good representation learning
- Choose an appropriate training algorithm
- Baseline Selection and Evaluation



Experimental Challenges

The rest of this talk:

- **Challenges of Collecting Appropriate Dataset**
- Challenges of Baseline Selection and Evaluation

Existing real-world datasets

- Researchers propose different Datasets
- Examples :
  - Limitations:
    - Many of them are not publicly available
    - Contains limited number of programs in the training (lack of diversity)
    - Specific for binary similarity detection
  - ,
  - Jtrans (Wang et al., ISSTA 2022)

Features of an ideal dataset

- Training needs many vulnerable/non vulnerable functions
- It should cover large number for each vulnerability type
- Dataset should reflect real world software
- Contain of diverse programs (lead to overfit the training process)

Question: How to assemble such a dataset?

Easy Answer

CWE121_Stack_Based_Buffer_Overflow_12

Shows the level of data/control complexity

Juliet C / C++ Test Suite
(NSA Center for Assured Software, 2012)

Features:

- Contains lar
  patched C,C
- Contains 118
- Diverse progr
- Include both
  vulnerabilitie
- Include func
  and control f
- Buggy/patch
  macros
- Vulnerable/patched lines define with code comments

Advantages for training:

- Enough samples
- Labeled data with ground truth
- Diversity
- Cover different types of vulnerabilities

```
#ifndef OMITBAD
void
CWE121_Stack_Based_Buffer_Overflow
run_memcpy_12_bad()
                                    of the "if"
{
  ....
}
#endif /* OMITGOOD */
```

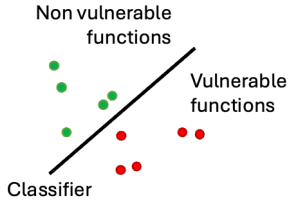
Challenges of training using Juliet

- Functions are small size
- Vulnerabilities are artificial and handmade

Issues with Existing approaches

- Use dataset for function similarity detection
- Examples:
  - Genuis (Feng et al., CCS 2016)
  - Asm2vec (Ding et al., SP 2019)
  - Jtrans (Wang et al., ISSTA 2022)

Granularity of Detection



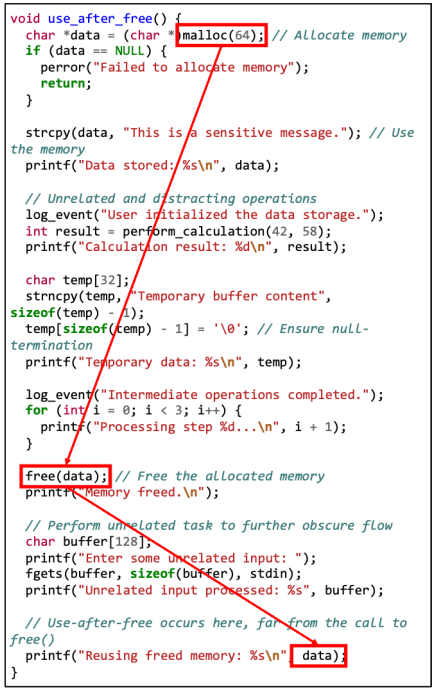
- **Challenge 1:** How to make classifier focus on the vulnerability, rather than on function semantics?
- Function similarity detection causes issues with “large” functions

```c
void user_authentication(const char *input) {
  char buffer[16]; // Fixed size buffer
  strcpy(buffer, input); // NO BOUNDS CHECK!!!
  printf("Authenticating user %s\n", buffer);
  ...
  if (...) {
    printf("Access granted.\n");
  } else {
    printf("Access denied.\n");
  }
}
```

```c
void process_file(const char *fname) {
  char temp_buffer[16]; // Fixed size buffer
  strcpy(buffer, fname); // NO BOUNDS CHECK!!!
  printf("Processing file...\n");
  ...
}
```

Granularity of Detection

- **Challenge 2:** Simply identifying vulnerable functions does not help with triage or patching
- Problem amplified when function body is large



```c
void use_after_free() {
  char *data = (char *)malloc(64); // Allocate memory
  if (data == NULL) {
    perror("Failed to allocate memory");
    return;
  }

  strcpy(data, "This is a sensitive message."); // Use
the memory
  printf("Data stored: %s\n", data);

  // Unrelated and distracting operations
  log_event("User initialized the data storage.");
  int result = perform_calculation(42, 58);
  printf("Calculation result: %d\n", result);

  char temp[32];
  strncpy(temp, "Temporary buffer content",
sizeof(temp) - 1);
  temp[sizeof(temp) - 1] = '\0'; // Ensure null-
termination
  printf("Temporary data: %s\n", temp);

  log_event("Intermediate operations completed.");
  for (int i = 0; i < 3; i++) {
    printf("Processing step %d...\n", i + 1);
  }

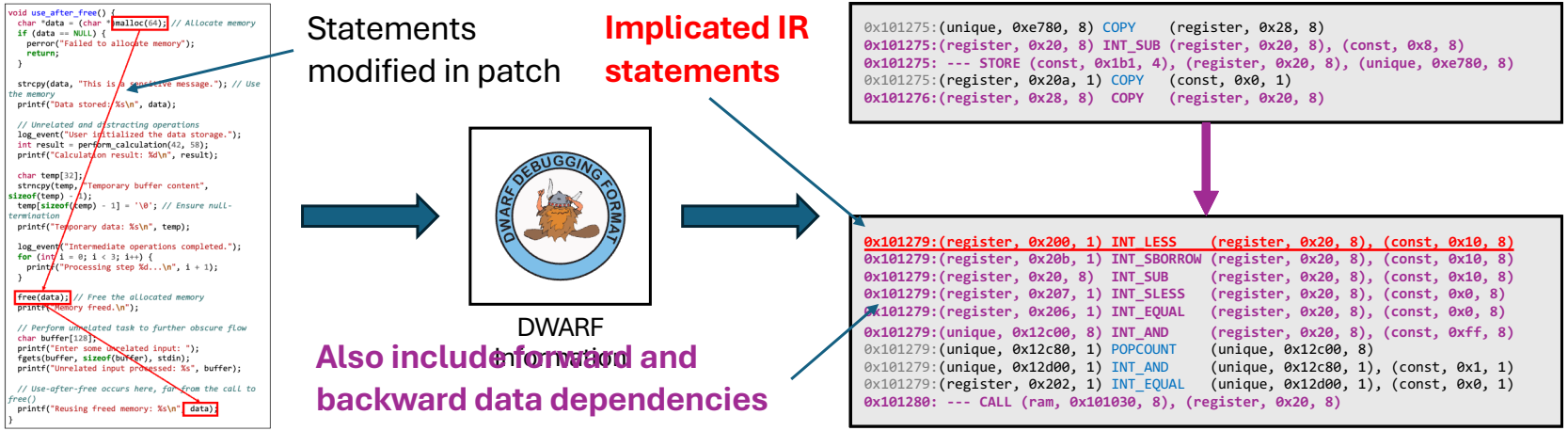
  free(data); // Free the allocated memory
  printf("Memory freed.\n");

  // Perform unrelated task to further obscure flow
  char buffer[128];
  printf("Enter some unrelated input: ");
  fgets(buffer, sizeof(buffer), stdin);
  printf("Unrelated input processed: %s", buffer);

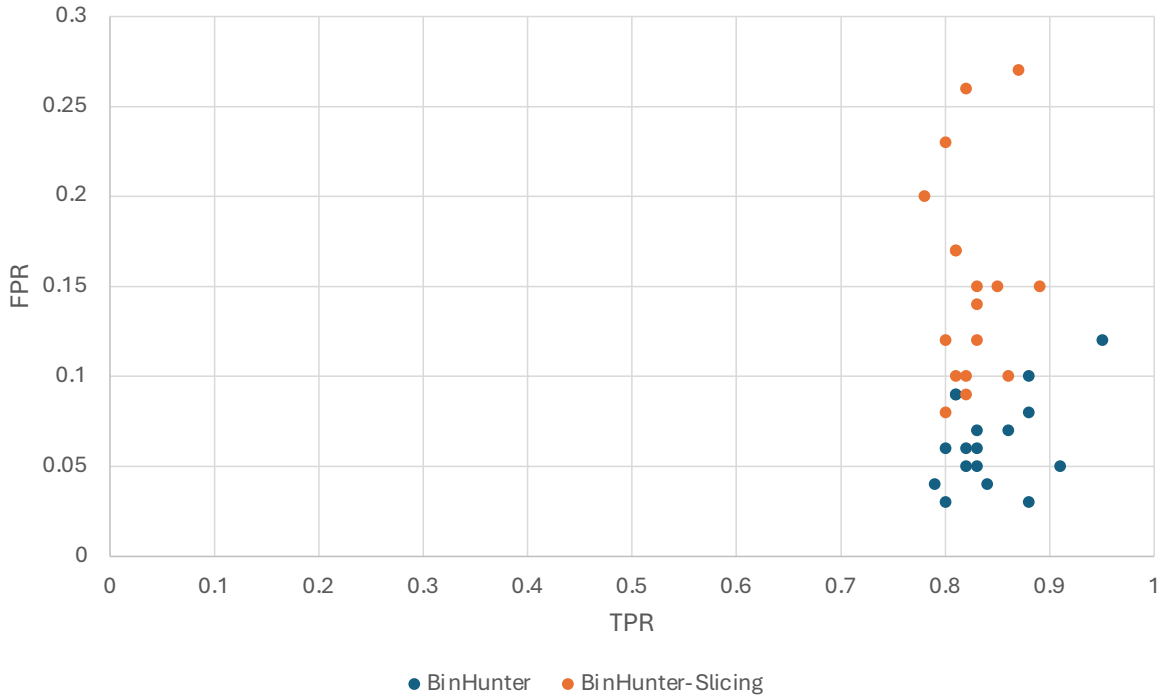
  // Use-after-free occurs here, far from the call to
free()
  printf("Reusing freed memory: %s\n", data);
}
```

Solution

- Question?
  - Can we train and test at small scale?



Ablation study on Juliet

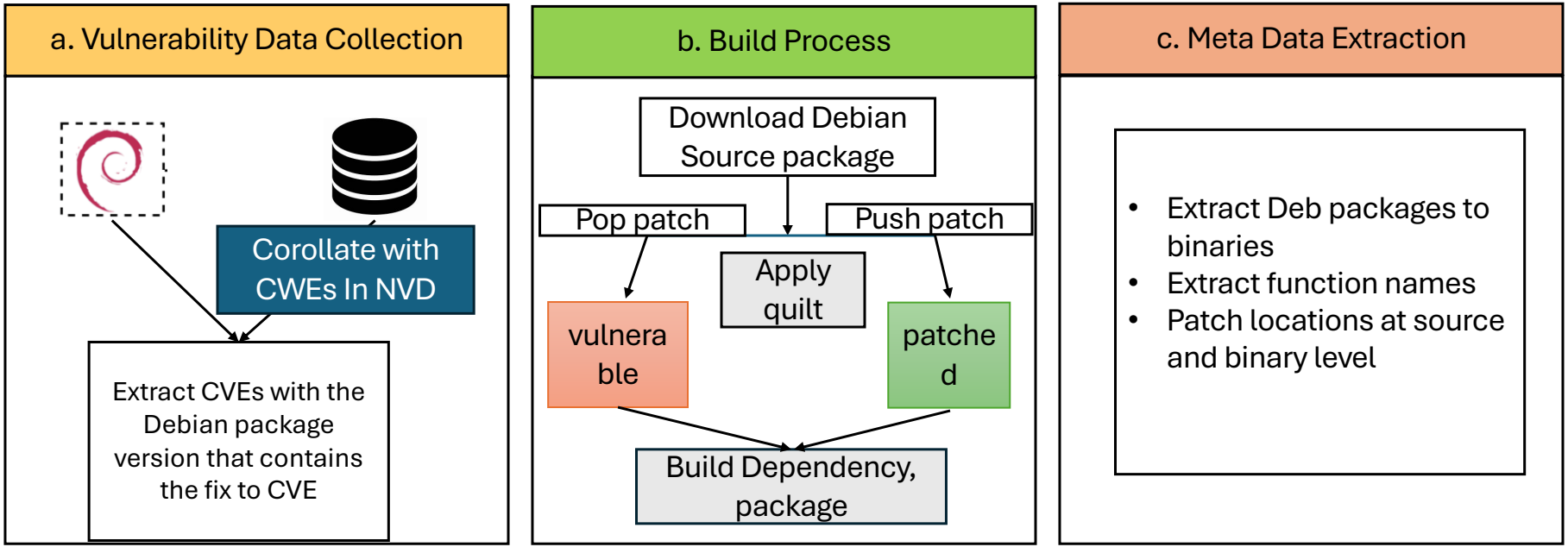


Objections:

- Not enough to judge if the method works
- Functions small and does not reflect reality
- Need Evaluation on real-world software

How Do We Assemble a Real-World Dataset?

- Historical vulnerabilities in Debian Packages



Challenges of assembling our benchmark

- Each Debian package needs its own environment to get build
- The manual build process is time consuming and labor intensive
- Trouble shooting in case of failure

Can BinHunter even detect bugs in ideal situation?

Question?

- Can we even learn anything from Juliet?
- Is this knowledge transferable to real world ?

Answer:

- Conduct an experiment to first test on bug locations in real world software

Ablation study in Debian Packages

- BinHunter detects 17 out of 24 bugs
- BinHunter/slicing detects only 1bug

Experimental Challenges

The rest of this talk:

- Challenges of collect appropriate benchmarks
- Challenges of baseline selection and evaluation

Criteria of Baseline Selection

- Contains Experiments related to vulnerability Discovery
- Open Source
- Align with our problem

Difficulties in baselines

Challange1:

- Many approaches are bug search engines
  - Search a target program in a function repositories
  - Evaluation: rank functions similar to the target function

  Compute the evaluation metrics for n top functions
- Our approach : trains a separate binary classifier for each vulnerability type
  - Evaluation: compare with ground truth labels

Challange2:

- Some baselines use tools that require license and not free
- Examples: Jtrans, Genuis, Asm2vec

Challange3:

- Sometime code modification is required to adjust the baseline on our dataset

  Example: Jtrans

How to align a fair comparison?

- Use the logic behind the classification
- Align this logic with ranking based approaches
  - Example:
    - If more than half of returned functions @n top ranked functions

    Are similar with the target, the detection label is correct

Lessons I learned

1. Care needed while selecting benchmarks
    a. Need to be relevant to technique being developed
    b. Large amounts of training data needed (Synthetic data is great, but how to generalize?)
    c. Evaluation should show real-world efficacy (Historical CVEs, new bugs, etc.)
2. Care needed while deploying baselines
    a. Many baselines not freely available or require proprietary licenses (We can at least make our own artifacts public)
    b. Modifications needed to handle small changes in problem setting (Function similarity vs. bug detection, etc.)
    c. Figuring out why something works / does not work is hard