Blanket Execution:
Dynamic Similarity Testing for Program Binaries and Components

Manuel Egele, Maverick Woo,
Peter Chapman, and David Brumley
Carnegie Mellon University

Picture Yourself as an Analyst

You just identified a function of interest

Questions:

- Have I seen an equivalent or similar function before?
- How can I find binaries that contain similar functions?

Similar vs. Equivalent

```
static int strcmp_name(V a, V b) {
  return cmp_name(a, b, strcmp);
}

static inline int cmp_name (
  struct fileinfo const *a,
  struct fileinfo const *b,
  int (*cmp) (char const *,char const *)
)
{
  return cmp (a->name, b->name);
}
```

Similar vs. Equivalent (cont.)

```
407ab9 <strcmp_name>:
   ab9: push %rbp
...
   ad1: mov $0x402710,%edx
... PLT entry of strcmp
   ad6: mov %rcx,%rsi
   ad9: mov %rax,%rdi
   adc: callq 406fa1 <cmp_name>
   ae1: leaveq
   ae2: retq

406fa1 <cmp_name>:
   fa1: push %rbp
...
   fcd: callq *%rax
... call func. pointer (e.g., strcmp)
   fcf: leaveq
   fd0: retq
```

gcc –O0

```
4053e0 <strcmp_name>:
    e0: mov (%rsi),%rsi
    e3: mov (%rdi),%rdi
    e6: jmpq 402590
         <strcmp@plt>
```

Syntactic differences complicate static similarity analysis

gcc –O3

Function-Binary Similarity

Question with plenty security applications

- Patch analysis / patch-based exploit generation
  Which function has (not) been patched?
- Malware analysis
  Did I analyze similar code like this already?
- Higher-level concepts
  Function-binary search engine

Blanket Execution

Dynamic analysis

- Execute function ***f*** under a fixed environment
- Record side effects (features) of this execution
- Two functions ***f*** and ***g*** are similar if their side effects are similar

Limited coverage

- Execute ***f*** repeatedly starting from first un-executed instruction → full line coverage
- But: Natural meaning of function execution (i.e., start from beginning) is sacrificed

Execution Environment

- Provides concrete & consistent values for:
  - All registers
  - All memory locations
- Must be efficiently reproducible
- Blanket Execution-Run:
  1. Load target binary via OS loader
  2. Initialize execution environment
  3. Divert control from program entry point to the first un-executed instruction in ***f***

Implementation Considerations

- Compiled functions have dependencies
  - Global variables
  - Structure of passed arguments

- In blanket execution, functions are executed in randomized but fixed environment
  - Dependencies are likely not met → frequent accesses to unmapped memory

Argument Access

```
static inline int cmp_name (
  struct fileinfo const *a,
…
return cmp (a->name, b->name);

struct fileinfo {
char * name, … }

e0: mov (%rsi),%rsi
```

Implementation (cont.)

- Environment specifies dummy memory page
- Dummy page is mapped (on demand) at all unmapped addresses
  - Memory writes succeed
  - Memory reads — consistent and succeed
- Consistent values allow comparison

Side Effects & Feature Vectors

- Dynamically observable features (e.g., memory accesses, syscalls, etc.)
- Combine all side effects per function into a feature vector of length N (for N features)
- Coordinates: sets of observed feature values
- Similarity score for ***f*** and ***g***

$$sim_k(f,g) = \sum_{i=1}^{N} \left(w_i \times \frac{|v_i(f, env_k) \cap v_i(g, env_k)|}{|v_i(f, env_k) \cup v_i(g, env_k)|} \right) / \sum_{\ell=1}^{N} w_\ell$$

weighted   Jaccard indices   normalized

Features

- Memory reads/writes to the stack
- Memory reads/writes to the heap
- System calls
- Library calls via plt
- Function return value in `%rax`

Dataset

- GNU coreutils 8.13 (95 binaries)
- Three compilers:
  - GNU gcc 4.7.2
  - Intel icc 14.0.0
  - LLVM clang 3.0-6.2
- Four optimization levels each (-O{0,1,2,3})
- Result: 1,140 binaries, 195,560 functions
- Debug symbols → ground truth through function names

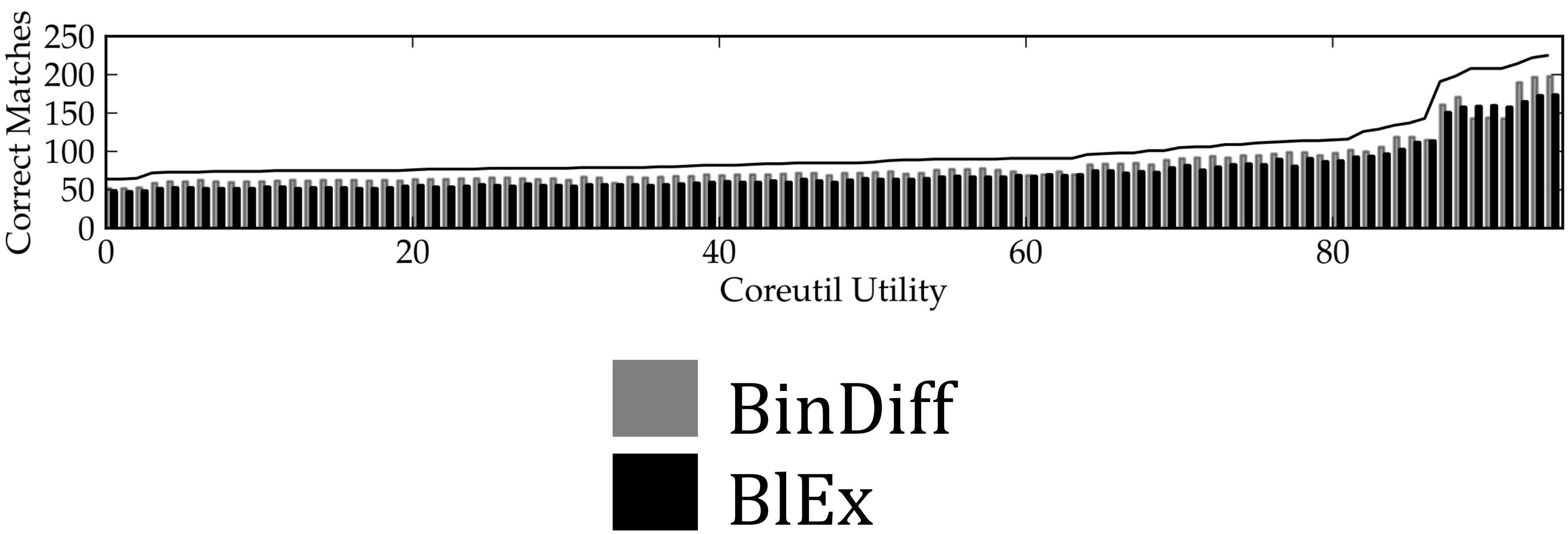
BlEx Performance

- Implemented with Intel's Pin
- 195,560 functions & 11 environments
- 1,590,773 BE-runs / environment
- 17,498,507 BE-runs →57 CPU days
- Two versions of `ls` ~ 30 CPU minutes
- Independent executions → embarrassingly parallel workload

Results vs. BinDiff

Proxy for (dis-)similarity: # optimizations

- O2 vs. O3 high similarity (9 optimizations)
- O0 vs. O3 high dissimilarity (66 optimizations)

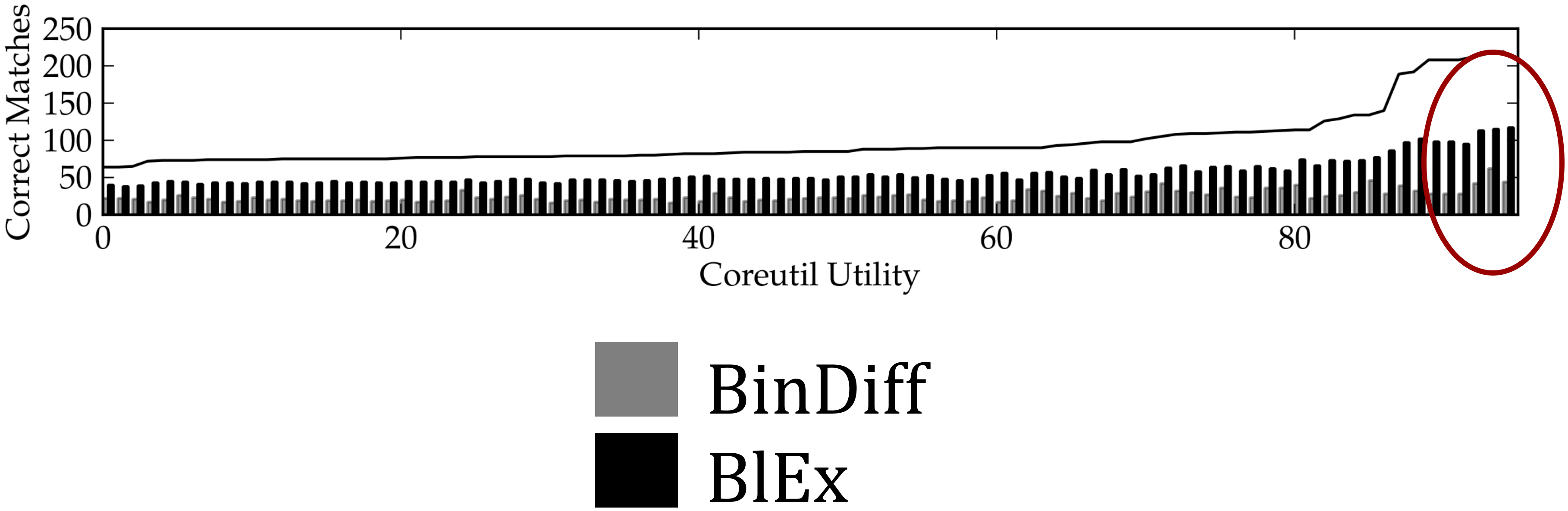


Results vs. BinDiff (cont.)

Large syntactic differences

– gcc –O0 vs. gcc –O3

– BlEx outperforms BinDiff 2x on avg. (up to 3.5x)



Binary Search Engine

Given:

- An indexed corpus C of function-binaries / feature vectors (v_1, …, v_n)
- A search query function ***f***

Result:

- Which feature vector $v_i \in C$ corresponds to the function ***g*** most similar to ***f***
- Sort results w.r.t. similarity with ***f***

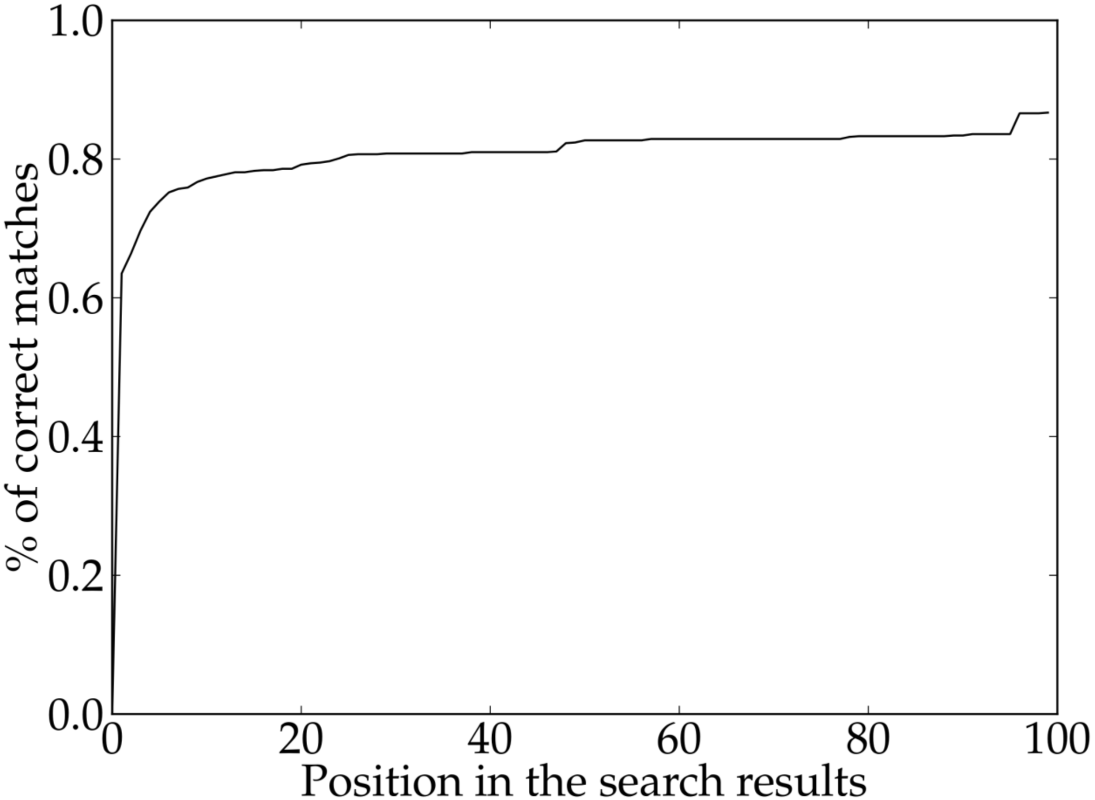
Binary Search Engine — Experiment

- Queries: $q_1, \ldots, q_{1,000}$ randomly selected functions from coreutils (gcc –O0)
- Corpus C: 29,015 remaining functions from coreutils (gcc –O1, gcc –O2, gcc –O3)
- Single search executes in < 1s

Binary Search Engine — Results

64% correct match at the top

77% correct match under top ten



Summary

- Function binary similarity is a challenge
  - Static approaches thwarted by syntactic differences (e.g., compiler or optimization)
- Blanket Execution: dynamic analysis to identify similar function-binaries
  - Coverage achieved by re-executing function
  - Functions are similar if their feature vectors are
  - Outperforms static systems for large syntactic differences
  - Blanket execution can be used as a building block for a binary search engine

END