# Where's the Length? Inferring Size Parameters in Function Interfaces

Jiajun Cheng
May 29, 2025

UC RIVERSIDE

# Problem definition

**Goal**

Detect whether a *pointer-length* relationship exists between two arguments in a function call.

**Input**

- A function **f(…)**
- Two arguments of **f(…)**, one of them is **a pointer**, and the other is **an integer**.
- A call site to **f(…)**
- Little or no assumption about **f**'s internal implementation

```
char *buf = malloc(32);
size_t n = 32;
f(buf, n);    // Is n the size of buf?
```

UC RIVERSIDE

# Problem definition

## What's under-constrained symbolic execution?

### Project purpose

Augment **under-constrained symbolic execution** with extra knowledge about black-box library calls.

### Symbolic execution

**Symbolic execution,** similar to **abstract interpretation** (but not using lattices), is a program analysis technique that explores program paths using symbolic inputs instead of concrete values, generating path conditions to describe feasible execution paths.

### Problem: Scalability

Path explosion: $|\text{paths}| \sim 2^{|\text{if-statements}|}$

```
int foo(int x) {
 if (x)
   return x/10;
 else
   // in order to reach this branch
   // x == 0
   return 10/x;
}
```

# Problem definition

## What's under-constrained symbolic execution?

**Project purpose**
Augment **under-constrained symbolic execution** with extra knowledge about black-box library calls.

**Under-constrained symbolic execution**
It extends the idea of symbolic execution to analyze isolated functions or fragments without full calling context, using partial inputs and assumptions to simulate how the code might behave in real scenarios.

**Problem: call contract**
We do not know if b is the size of a, if no calling contexts are given.

```
int sum(int b, int* a)
{
  // Is b the size of a?
  int r = 0;
  for(int i = 0; i < b; i++)
  {
    r += a[i];
  }
  return r;
}
```

```
int last(int b, int* a)
{
  // Is b the size of a?
  return a[b-1];
}
```

UC RIVERSIDE

# Problem definition

## Why we don’t care how f is implemented?

### Project purpose

Augment **under-constrained symbolic execution** with extra knowledge about black-box library calls.

### The chicken-and-egg problem

- If we peek inside **f** to “discover” a pointer-length rule, we must already assume that such a rule exists.
- That circular assumption undermines soundness.
- Therefore, we must **infer the contract without inspecting f’s body.**

### Analyzing f might be hard

Considering the last function, we cannot figure out the semantics of f by itself.

```
int sum(int b, int* a)
{
  // Is b the size of a?
  int r = 0;
  for(int i = 0; i < b; i++)
  {
    r += a[i];
  }
  return r;
}
```

```
int last(int b, int* a)
{
  // Is b the size of a?
  return a[b-1];
}
```

UC RIVERSIDE

# Existing Work

- **Array Length Inference for C Library Bindings**
- Given a C function, without looking at call sites, inferring pointer-length rules.
- Provide the opposite information to the analysis.


- Alisa J. Maas, Henrique Nazaré, and Ben Liblit. 2016. Array length inference for C library bindings. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16). Association for Computing Machinery, New York, NY, USA, 461–471. https://doi.org/10.1145/2970276.2970310


```
int symbolicLoop(int *array, int y)
{
    int *end = array + y;
    int sum = 0;
    while (array < end) {
        sum += *array;
        array++;
    }
    return sum;
}
```

**Listing 4: Example function with an argument of symbolic length containing a specialized loop**

```
guint g_str_hash (gconstpointer v)
{
    const signed char *p;
    guint32 h = 5381;

    for (p = v; *p != '\0'; p++)
        h = (h << 5) + h + *p;

    return h;
}
```

**Listing 5: Real-world function with an argument sentinel-terminated by NUL, taken from glib**

UC RIVERSIDE

# Constraint System

## Modeling Pointers as Tuples

We model each pointer as a tuple:

$$ptr = (address, length, capacity)$$

- address: the memory location it points to
- capacity: size of the underlying buffer (i.e., how much memory is allocated)
- length: meaningful content size (e.g., strlen(ptr) for strings)

We naturally have these constraints:

$$0 \leq ptr.address$$
$$0 \leq ptr.length \leq ptr.capacity$$

# Constraint System

- For pointer arithmetic (adding a number):

$$ptr' = ptr + offset \Rightarrow \begin{cases} ptr'.address = ptr.address + offset \\ ptr'.length = ptr.length - offset \\ ptr'.capacity = ptr.capacity - offset \end{cases}$$

- For pointer arithmetic (pointer difference):

$$offset = ptr' - ptr \Rightarrow offset = ptr'.address - ptr.address$$

- For malloc and array initialization:

$$ptr = malloc(n) \Rightarrow ptr = (unconstrainted, unconstrainted, n)$$

- For realloc:

$$ptr' = realloc(ptr, n) \Rightarrow ptr' = (unconstrainted, unconstrainted, n)$$

- For strlen:

$$len = strlen(ptr) \Rightarrow len = ptr.length$$

# Constraint System

- For more complex functions (like strchr):

$$ptr' = strchr(ptr, ch) \Rightarrow \begin{cases} ptr' = NULL \quad or \\ \exists\delta.\, 0 \le \delta < ptr.length, ptr' = ptr + \delta \end{cases}$$

```c
int main (){
  char str[] = "This is a sample string";
  printf ("Looking for the 's' character in \"%s\"...\n",str);
  char * pch=strchr(str,'s');
  while (pch!=NULL){
    printf ("found at %d\n",pch-str+1);
    pch=strchr(pch+1,'s');
  }
  return 0;
}
```

```
Output:

Looking for the 's' character in "This is a sample string"...
found at 4
found at 7
found at 11
found at 18
```

# Solving the Constraint System

## SMT Solver

### What is an SMT Solver?

- Satisfiability Modulo Theories Solver
- Generalizes SAT solving: Boolean satisfiability + rich background theories
- Answers SAT / UNSAT / UNKNOWN (timeout)
- SMT solving is **NP-hard**.
- Keep formulas small and simple to solve the system efficiently.

### Why Z3?

- High-performance solver from Microsoft Research
- Incremental solving with push/pop for repeated queries
- Battle-tested, widely used in compilers and verification tools

# Solving the Constraint System

## Step 1: Identify Candidate Relations

- Given a function $f(ptr_1, \dots, ptr_n, int_1, \dots, int_m)$
- Collect all **pointer arguments** and **integer arguments**
- For each pair $(ptr_i, int_j)$, test whether these properties hold:
    - $int_j = ptr_i.length$
    - $int_j \leq ptr_i.length$
    - $int_j = ptr_i.capacity$
    - $int_j \leq ptr_i.capacity$

# Solving the Constraint System

## Step 2: Use Z3 to Prove Relationships

To check if $int_j = ptr_i.capacity$ always holds

1. Add all generated constraints into a Z3 context
2. Check satisfiability
   - If UNSAT → unreachable
   - If SAT → continue
3. Add the negation: $int_j \neq ptr_i.capacity$
4. Re-check SAT:
   - If UNSAT, the original equality must always hold
   - If SAT, the equality is not guaranteed

We can use push/pop commands in Z3 to speed up solving multiple related checks, since only the predicate being tested changes—while the base constraint system remains the same.

# Solving the Constraint System

## Step 3: Score calculation

- Compute score for each $(ptr_i, int_j)$ pair:

$$\text{score} = \frac{\#\{\text{checks that always held}\}}{\#\{\text{total checks}\}}$$

- Why a check might "not hold":
  1. False assumption: the relation genuinely fails at some call site
  2. Bounded context: limited to k-length call strings (may miss deeper contexts)
  3. Framework unknown: our analyses are unable to proof the assumption
- Using scores in under-constrained symbolic execution
  - Feed these confidence scores into the under-constrained symbolic executor
  - Prioritize or prune paths based on high-confidence pointer-length facts

# Future work

## Stochastic Branch-Path Scoring

### Idea

Estimate each $(ptr_i, int_j)$ confidence score by sampling execution paths instead of merging with conservative upper bounds.

### Stochastic Path Procedure

To check if $int_j = ptr_i.capacity$ always holds

1. Seed Z3 with all base constraints for the current call context.
2. Check satisfiability
   - If UNSAT → path unreachable ⇒ discard sample
   - If SAT → keep path
3. Add the negation: $int_j \neq ptr_i.capacity$
4. Re-check SAT:
   - If UNSAT, the original equality must always hold
   - If SAT, the equality is not guaranteed
     - Option A: stop and record result
     - Option B: recursively extend context with another caller level

# Future work

## Stochastic Branch-Path Scoring

### Idea

Estimate each $(ptr_i, int_j)$ confidence score by sampling execution paths instead of merging with conservative upper bounds.

### Score calculation

Compute score for each $(ptr_i, int_j)$ pair:

$$score = \frac{\#\{\text{sampled paths where relation always holds}\}}{\#\{\text{sampled reachable paths}\}}$$

### Benefits

- Simpler constraints: avoids expensive least upper bound joins.
- Efficiency: dramatically lighter Z3 queries per path.
- Parallelism: independent path samples can run concurrently across cores or a compute cluster.

# Completed Milestones

**LLVM instruction support:**

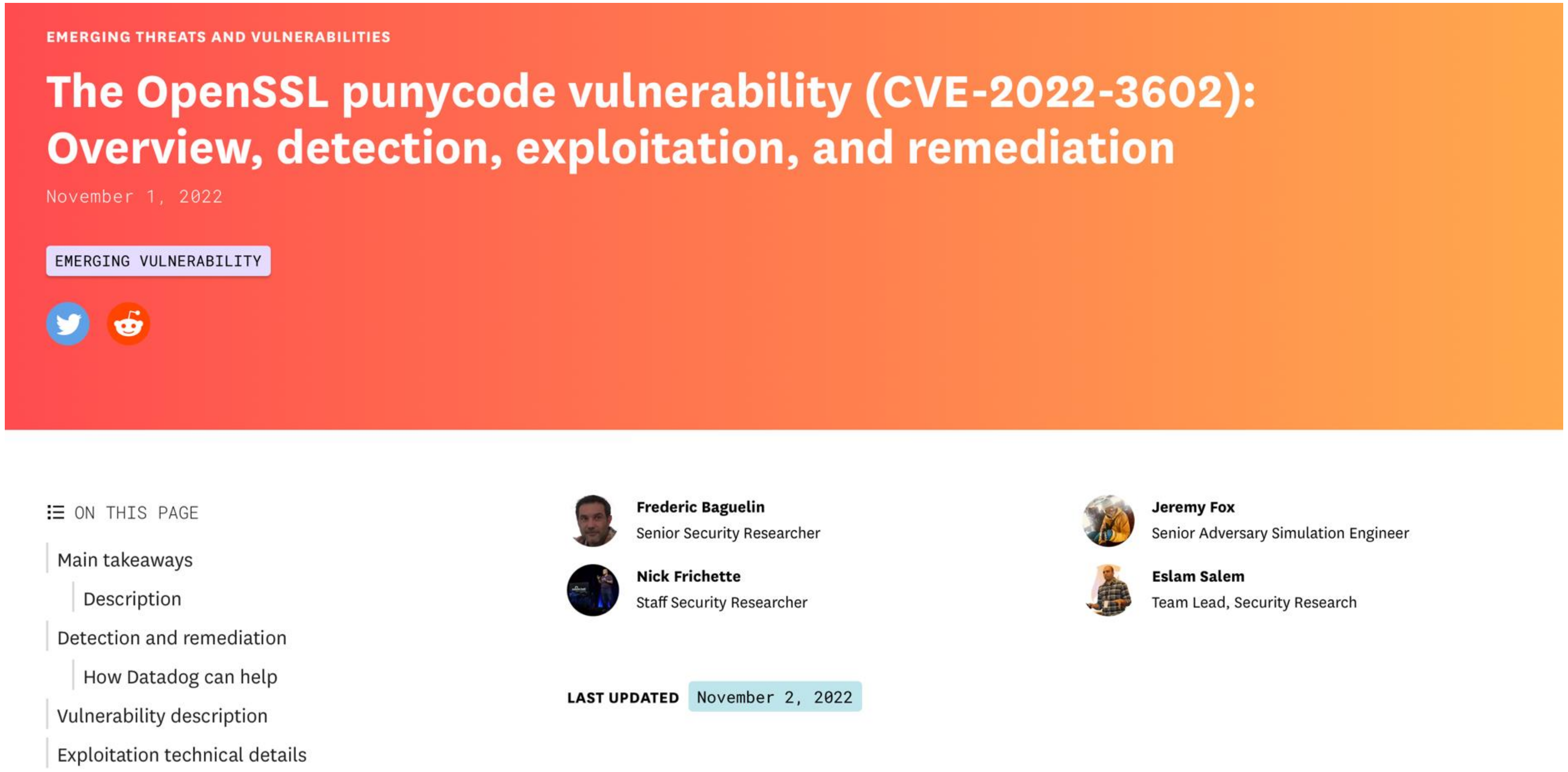
- GetElementPtrInst
- BinaryOperator
- TruncInst
- PtrToIntInst
- CallInst for strlen and strchr
- ICmpInst
- Partial support for PHINode

**Produce correctly results on some simple programs**

- Demo in the next slide

# Demo

## Check if our analysis pass can find the pointer-length rules for ossl_punycode_decode




UC RIVERSIDE

# Q&A