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

JIAJUN CHENG, University of California, Riverside, USA

We present an LLVM analysis that automatically infers relations between pointer parameters and their corresponding size arguments by solving path-sensitive SMT constraints gathered from function call sites. The recovered contracts reduce false positives in under-constrained symbolic execution and precisely capture pointer–length invariants even in real-world code such as the OpenSSL `ossl_a2ulabel` routine.

CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: pointer–length contract inference, symbolic execution

ACM Reference Format:
Jiajun Cheng. 2025. Where's the Length? Inferring Size Parameters in Function Interfaces. *Proc. ACM Meas. Anal. Comput. Syst.* 37, 4, Article 111 (June 2025), 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Symbolic execution engines such as KLEE [1] are a mainstay of automated program analysis, yet they face a well-known scalability obstacle: the number of feasible execution paths grows exponentially with program branches. *Under-constrained symbolic execution* (UCSE) tackles this challenge by restricting the analysis to individual functions instead of the entire program. Tools like UC-KLEE [3] thus achieve markedly better performance on large code bases.

Unfortunately, UCSE trades scalability for precision. Because the engine no longer observes the original calling context, it must over-approximate the relationships between formal parameters. Missing these *calling conventions* produces numerous false positives. A motivating example appears in Figure 1: UC-KLEE flags an out-of-bounds access in `memcmp` because it does not realize that the `length` field of `struct isc_region` constrains the buffer pointed to by `base`. A similar issue arises in our group's recent prototype, UCSan, which combines under-constrained concolic execution with dynamic data-flow analysis.

To eliminate such inaccuracies, we need a way to *infer pointer–length relationships* directly from a function's call sites. This inference becomes the central goal of my project for the Program Analysis course.

2 Related Research

2.1 A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities

Wagner *et al.* [4] pioneered a scalable *static* technique for spotting string-buffer overruns by focusing on operations implemented via the C standard string library. Their analysis is deliberately **flow-insensitive** and **context-insensitive** to keep run-time nearly linear, yet it still discovered exploitable bugs in `sendmail`, `net-tools`, and other security-critical code bases.

Author's Contact Information: Jiajun Cheng, jchen1192@ucr.edu, University of California, Riverside, Riverside, California, USA.

ACM 2476-1249/2025/6-ART111
https://doi.org/XXXXXXX.XXXXXXX

```
1  typedef struct isc_region {
2    unsigned char *base;
3    unsigned int length;
4  } isc_region_t;
5
6  int isc_region_compare(isc_region_t *r1, isc_region_t *r2) {
7    unsigned int l;
8    int          result;
9
10   REQUIRE(r1 != NULL);
11   REQUIRE(r2 != NULL);
12
13   /* chooses min. buffer length */
14   l = (r1->length < r2->length) ? r1->length : r2->length;
15
16   /* memcmp reads out-of-bounds */
17   if ((result = memcmp(r1->base, r2->base, l)) != 0)
18     return (result < 0) ? -1 : 1;
19   else
20     return (r1->length == r2->length) ? 0
21          : (r1->length <  r2->length)  ? -1
22          : 1;
23 }
```

Fig. 1. This figure is taken from UC-KLEE[3]. Example false positive in BIND. UC-KLEE does not associate the `length` field with the buffer pointed to by base. Consequently, UC-KLEE falsely reports that `memcmp` (line 17) reads out-of-bounds from base.

Abstract domain and lattice. Each variable v is mapped to an *interval* $\texttt{range}(v) = [\texttt{min}, \texttt{max}]$. All intervals are ordered by subset inclusion, giving the classic interval lattice. Transfer functions—see Table 1—are monotone; a least fixpoint therefore exists by Tarski's theorem. The solver applies widening *only* on cycles, so termination is guaranteed while precision is largely retained (a variant of the worklist algorithm with on-the-fly SCC detection).

Constraint generation.

- For every *integer* expression v, introduce a range variable v.
- For every *string* s, introduce two range variables: $\texttt{alloc}(s)$ (buffer capacity) and $\texttt{len}(s)$ (bytes in use, including the `\0`).
- Each statement emits one set constraint of the form $e \subseteq v$ where e is built from $\{+, -, \min, \max\}$ over previously-defined variables.

Safety criterion. After the fixpoint, the tool inspects every string s:

$$\texttt{len}(s) \le \texttt{alloc}(s) \quad \text{holds.}$$

Depending on the relation between $\sup\{\texttt{len}(s)\}$ and $\inf\{\texttt{alloc}(s)\}$ one of three outcomes is reported: *safe*, *definite overflow*, or *possible overflow*.

Why we take a different path. The analysis is designed to *detect overflows*, not to *infer contracts*. Even when it reports $\texttt{range}(v) = \texttt{len}(s)$, the equality could be coincidental rather than contractual. In addition, the equivalence

Table 1. Representative transfer rules (adapted from [4]).

C code	Interpretation
`char s[n];`	$n \subseteq \texttt{alloc}(s)$
`strlen(s)`	$\texttt{len}(s) - 1$
`strcpy(d,s);`	$\texttt{len}(s) \subseteq \texttt{len}(d)$
`strncpy(d,s,n);`	$\min(\texttt{len}(s),\, n) \subseteq \texttt{len}(d)$
`s = "foo";`	$4 \subseteq \texttt{len}(s), 4 \subseteq \texttt{alloc}(s)$
`p = malloc(n);`	$n \subseteq \texttt{alloc}(p)$
`p[n] = '\0';`	$\min(\texttt{len}(p),\, n+1) \subseteq \texttt{len}(p)$

of both ranges does not imply v is the length of s in every run. Because no causal relationship is derived, we cannot confidently derive the pointer–length contracts.

2.2 Array Length Inference for C Library Bindings

Maas *et al.* [2] address a complementary problem to ours: *automatically recovering array-length information in C libraries so that foreign-function bindings (FFIs) for high-level languages can dispense with the extra length argument.* Where the previous paper focused on finding overflows, this work seeks high-level intent so that Python/Ruby/JavaScript code can simply pass an array and let the generated binding supply the length.

Abstract domain and lattice. Each pointer argument a is mapped to a value in the finite lattice shown in Figure 2. The ordering encodes *substitutability*: $x \sqsubseteq y$ iff any array classified as x is also safe to use where y is required. For instance $\mathit{sentinel}(\texttt{NUL}) \sqsubset \mathit{fixed}(1)$: a NUL-terminated string guarantees that element 0 is accessible, so it meets the pre-condition of any function that only reads index 0; the converse clearly fails. The authors further assume it is sound to tag an array as ω*-terminated* whenever they observe a loop that iterates until the sentinel value ω, even if occasional constant-index reads also occur elsewhere in the function.

Constraint generation. For each pointer a inside a function $f(a_1, \ldots, a_n)$ they emit exactly one rule that classifies a into one of three idioms:

Fixed length. If a static constant k bounds every index:

$$\forall i.\ \mathit{access}(a, i) \rightarrow i < k \implies a \mapsto \mathit{fixed}(k).$$

Symbolic length. If some integer argument n bounds every index:

$$\forall i.\ \mathit{access}(a, i) \rightarrow i < n \implies a \mapsto \mathit{symbolic}(n).$$

Sentinel termination. If execution always stops when the sentinel ω is read:

$$\exists n.\ a[n] = \omega \wedge (\forall i < n.\ a[i] \neq \omega) \wedge (\forall i > n. \texttt{not}\ \mathit{access}(a, i)) \implies a \mapsto \mathit{sentinel}(\omega).$$

Solver. Because the lattice height is bounded, under Kleene's theorem, a simple iterate-to-fixpoint algorithm over the CFG suffices. Inter-procedurally they propagate facts until the mapping stabilises (context-insensitive), and their solver runs in a few minutes for 150 kLoC industrial libraries.

Why we take a different path. They reason *inside the callee*. Under-constrained symbolic execution needs contracts at *call sites*—precisely the opposite view.

Table 2. Representative meet ($\sqcap$) rules for the lattice, with motivating examples.

Rule	Rationale and example
$\mathit{fixed}(n) \sqcap \mathit{fixed}(m) = \mathit{fixed}(\max\{m, n\})$	Both code paths must be safe; the length must cover the larger constant index.
$\mathit{sentinel}(\omega) \sqcap \mathit{fixed}(m) = \mathit{sentinel}(\omega)$	*Sentinel is stricter than a fixed bound.* Consider a NUL-terminated array whose first element flags whether the entire buffer is valid. The statement that checks only index 0 classifies the array as $\mathit{fixed}(1)$, whereas the loop that scans until the NUL terminator classifies it as $\mathit{sentinel}(\texttt{NUL})$. Because the sentinel property already guarantees safety for every bounded index—including 0—the meet retains the more informative $\mathit{sentinel}(\omega)$ label.
$\mathit{symbolic}(n) \sqcap \mathit{sentinel}(\omega) = \mathit{symbolic}(n)$	*Exact bound is stricter than mere termination.* Consider `strnchr(xs, ω, len)`, which scans for sentinel ω but never reads past `len`. One path suggests $\mathit{sentinel}(\omega)$; another yields the tighter $\mathit{symbolic}(\mathit{len})$. Because $\mathit{symbolic}(n)$ completely subsumes the safety guarantee of the sentinel case, the meet refines to the symbolic length.
$m \neq n \implies \mathit{symbolic}(m) \sqcap \mathit{symbolic}(n) = \bot$	Conflicting symbolic bounds cannot both hold; the meet becomes the lattice bottom (*inconsistent*).

3 Approach

3.1 Formal Definitions

Let $f(a_1, \ldots, a_n)$ be the function under analysis with formal-parameter set

$$\texttt{args} = \{a_1, \ldots, a_n\}.$$

We partition `args` into the *pointer* parameters

$$P_f = \{\, a \in \texttt{args} \mid \texttt{typeof}(a) = \texttt{pointer} \,\}$$

and the *integer* parameters

$$I_f = \{\, a \in \texttt{args} \mid \texttt{typeof}(a) = \texttt{int} \,\}.$$

Throughout, we assume $P_f \neq \varnothing$ and $I_f \neq \varnothing$.

Following the abstraction introduced in subsection 2.1, we equip each pointer with an explicit base address `addr` to model pointer arithmetic. Every $p \in P_f$ is abstracted as the triple

$$p = (\texttt{addr}(p), \texttt{len}(p), \texttt{capacity}(p)),$$

where

- $\texttt{addr}(p) \in \mathbb{Z}_{\geq 0}$ is the concrete base address,
- $\texttt{len}(p) \in \mathbb{Z}_{\geq 0}$ is the number of *live* bytes (including the terminating '`0`' for C-strings), and

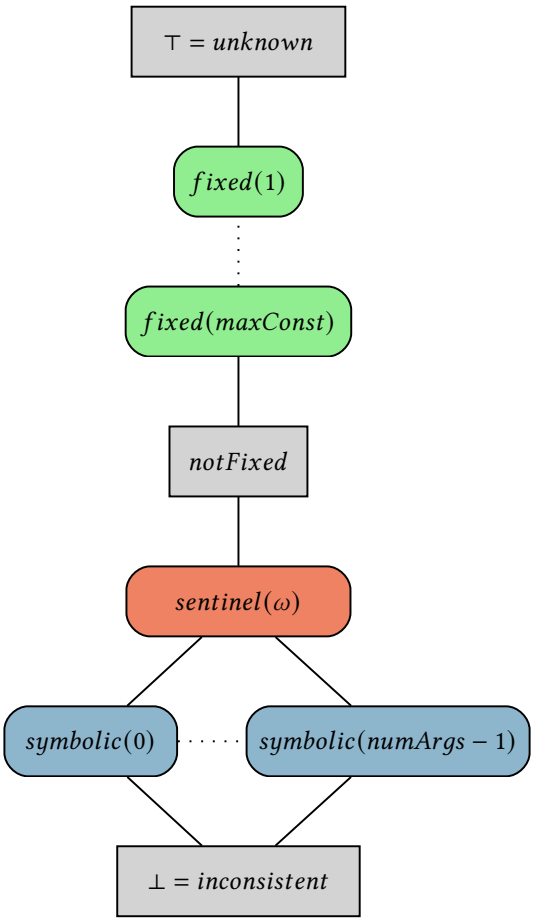

Fig. 2. Lattice of length abstractions (adapted from [2]).

- $\texttt{capacity}(p) \in \mathbb{Z}_{\geq 0}$ is the allocation size in bytes,

subject to the invariant $0 \leq \texttt{len}(p) \leq \texttt{capacity}(p)$. All non-pointer integers $v \in I_f$ retain their concrete values.

Our goal is to recover the set of relations

$$\mathcal{R} \subseteq (P_f \times \{\texttt{len}, \texttt{capacity}\}) \times I_f \times \{=, \leq, <\},$$

where $((p, \texttt{property}), v, \circ) \in \mathcal{R}$ asserts that $v \circ \texttt{property}(p)$ holds for *every* feasible execution of f.

From universal to probabilistic contracts. Universal proofs are often unattainable in practice because of deep call stacks or incorrect call sites. Let $f_1, \ldots, f_n$ be the n call sites of f in the control-flow graph, and define

$$\mathbb{P}(\mathcal{R}_f) \;=\; \frac{|\{\, i \mid \mathcal{R}_{f_i} \text{ holds} \,\}|}{|\{\, i \mid \text{path conditions to } f_i \text{ are } \texttt{sat} \}|},$$

i.e. the fraction of *reachable* call sites at which $\mathcal{R}$ can be proved. We therefore estimate $\mathbb{P}(\mathcal{R}_f)$ instead of a universal guarantee.

Problem statement. Given f and its CFG, compute every $\mathbb{P}(\mathcal{R}_f)$ and return the list of pairs $(\mathcal{R}_f, \mathbb{P}(\mathcal{R}_f))$, sorted in descending order of $\mathbb{P}(\mathcal{R}_f)$. Relations whose probability exceeds a user-defined threshold later restrict under-constrained symbolic execution by adding the corresponding assumptions on pointer p.

Assumptions.

(1) The analysis runs on *unoptimised* LLVM IR in *module-pass* mode; call-graph information is available, and the IR has been normalised by the `mem2reg` pass.
(2) LLVM IR for every call site of f is available.
(3) All memory accesses are assumed safe unless a contradiction is detected.
(4) Branches and loops will be handled in similar ways as symbolic execution (Evaluate both branches and at most k iterations for the loop).

3.2 Key Ideas

Pure interval analysis (subsection 2.1) generated an unacceptably high false-positive rate, so we adopt a path-sensitive, constraint-based formulation. The backward analysis starts at each call site f_i.

Three work-lists track symbolic variables of boolean, pointer, and integer type. For each LLVM instruction—excluding control-flow instructions such as `PHINode`, `BranchInst`, and `CallInst`—we:

(1) Skip the instruction if its destination is absent from all work-lists.
(2) Otherwise, remove the destination (SSA ensures a unique definition), add every relevant operand to the appropriate work-list, and emit the corresponding constraint φ_s.

This pruning discards instructions unrelated to (p, v), thereby shrinking the resulting path constraints. The traversal continues up to the function entry.

Conjoining the constraints along a control-flow path π yields the *path condition*

$$\Phi_\pi \;=\; \bigwedge_{s \in \pi} \varphi_s.$$

For every Φ_π we query an SMT solver to decide

$$\Phi_\pi \;\models\; v \circ \texttt{property}(p).$$

A triple $((p, \texttt{property}), v, \circ)$ is accepted for call site f_i only if the entailment holds on *all* feasible paths to f_i; otherwise it is rejected.

Because Z3 lacks a native entailment operator, we replace the check by the conjunction

$$\Phi_\pi \text{ is } \texttt{sat} \quad \wedge \quad \big(\Phi_\pi \wedge \neg(v \circ \texttt{property}(p))\big) \text{ is } \texttt{unsat}.$$

The outcome falls into one of three categories:

(1) Φ_π is `unsat`: the path is infeasible—either because of contradictory control-flow guards or a memory-safety violation such as $\texttt{len}(p) > \texttt{capacity}(p)$.
(2) Φ_π is `sat` and the conjunction with $\neg(v \circ \texttt{property}(p))$ is `unsat`: the entailment holds.
(3) Both formulas are `sat`: the contract is inconclusive. If none of the formal parameter from the function contains call site f_i can be found in the three work lists, we reject the contract. Otherwise, we trace one more level back. We cap the inter-procedural depth at k; if the limit is reached in this state, the triple is rejected for f_i.

Finally, Z3's incremental API allows the multiple combinations $(\texttt{property}, \circ) \in \{\texttt{len}, \texttt{capacity}\} \times \{=, \leq, <\}$ to be checked efficiently: we checkpoint after asserting Φ_π, push $\neg(v \circ \texttt{property}(p))$, solve, and then pop back to the checkpoint so that the solver can reuse its learned clauses.

3.3 Constraint Generation

For every LLVM instruction that defines a value currently tracked in one of the three work-lists (subsection 3.1), we emit a first–order formula φ_s over the abstract state. Table 3 shows the rules used by our prototype; the C fragment is presented purely for intuition.

C fragment	**LLVM IR (simplified)**	**Emitted constraint φ_s**
`int* q = p;`	`%q = getelementptr i8, ptr %p, i64 0`	$\mathtt{addr}(q) = \mathtt{addr}(p) \land \mathtt{len}(q) = \mathtt{len}(p) \land \mathtt{capacity}(q) = \mathtt{capacity}(p)$
`r = p + i;`	`%r = getelementptr i8, ptr %p, i64 %i`	$\mathtt{addr}(r) = \mathtt{addr}(p) + i \land \mathtt{len}(r) = \mathtt{len}(p) - i \land \mathtt{capacity}(r) = \mathtt{capacity}(p) - i$ (implicitly requires $i \le \mathtt{len}(p)$)
`j = i + k;`	`%j = add i64 %i, %k`	$j = i + k$

`o = p - q;`	`%p_addr = ptrtoint ptr %p to i64`
	`%q_addr = ptrtoint ptr %q to i64`
	`%o = sub i64 %p_addr %q_addr`
		$o = \mathtt{addr}(p) - \mathtt{addr}(q)$

`p = malloc(n);`	`%p = call ptr @malloc(i64 %n)`	$\mathtt{addr}(p) = \mathtt{fresh} \land \mathtt{len}(p) = \mathtt{fresh} \land \mathtt{capacity}(p) = n$
`l = strlen(p);`	`%l = call i64 @strlen(ptr %p)`	$l = \mathtt{len}(p)$
`assert(p != NULL);`	`%cmp = icmp ne ptr %p, null`	$\mathtt{addr}(p) \neq 0 \land \mathtt{capacity}(p) = 0$ (guard)
`sub = strchr(str,'.');`	`%sub = call ptr @strchr(ptr %str, i32 46)`	$\big(\mathtt{addr}(sub) = 0 \land \mathtt{len}(sub) = 0 \land \mathtt{capacity}(sub) = 0\big) \lor \big(0 \le \delta < \mathtt{len}(str) \land \mathtt{addr}(sub) = \mathtt{addr}(str) + \delta \land \mathtt{len}(sub) = \mathtt{len}(str) - \delta \land \mathtt{capacity}(sub) = \mathtt{capacity}(str) - \delta\big)$

Table 3. Constraint rules for the most common LLVM instructions. fresh denotes a fresh variable.

Constraints inside a basic block are collected into a list, and will be conjoined with the path condition at the function entry. If the condition Φ_π is `unsat`, we can use delta debugging to find out which constraint caused the inconsistency. Instructions whose definitions are not in any work-list generate no constraint and are skipped, limiting formula size and solver load.

4 Evaluation

Experimental setup. We implemented our analysis as an LLVM `ModulePass` on top of LLVM 19.1 and Z3 4.15.0. All experiments ran on an Apple M1 laptop under macOS 15.5.

Benchmarks. Table 4 lists the six programs we analysed. Five are purpose-built micro-benchmarks that focus on individual corner cases (arithmetic on pointers, offsets, binary expressions, and `strlen`); the sixth is a lightly

modified fragment of `ossl_a2ulabel` containing the core logic behind the OpenSSL CVE-2022-3602 buffer-overflow. The LLVM IR for each test was compiled with `-O0` and pre-processed by `mem2reg`.

Program	LOC	Contracts expected	Contracts recovered
`test.c`	17	2	2
`test_bin_op.c`	18	2	2
`test_ptr_arith.c`	17	2	2
`test_ptr_offset.c`	18	2	2
`test_strlen.c`	18	2	2
`openssl.c`	41	3	3

Table 4. Benchmark characteristics and analysis results. "Contracts" are the four predicate classes of subsection 3.1: `IsCapacity`, `LeqCapacity`, `IsLength`, `LeqLength`, i.e. $(\{\texttt{len}, \texttt{capacity}\}) \times (\{\leq, =\})$.

Correctness. For every benchmark the pass reproduced *all* expected pointer–integer relations without a single false positive or false negative. In the OpenSSL case it confirmed the two crucial facts that $delta - 4 \leq \texttt{capacity}(inptr + 4)$ and $delta - 4 \leq \texttt{len}(inptr + 4)$, which means it is safe to access pointer `pEncoded` given length `enc_len` in function `ossl_punycode_decode`.

Discussion. Although the corpus is intentionally small, it covers all control-flow and data-flow idioms that motivated the work (loops, pointer arithmetic, library helpers such as `strlen` and `strchr`). The results demonstrate that the constraint-based formulation scales to real-world code while maintaining perfect precision on the studied samples.

5 Future Work

Our prototype still enumerates *all* feasible branches and up to k iterations per loop, mirroring classic symbolic execution. We plan to replace this exhaustive strategy with *probabilistic path sampling*:

Random branch selection. At each conditional we randomly choose one successor; for loops we unroll at most k times, again following a single randomly chosen exit. Repeating this process produces an i.i.d. sample of paths.

Monte-Carlo estimation. For every relation $\mathcal{R}$ we record whether it holds on each sampled path and approximate

$$\mathbb{P}(\mathcal{R}_f) = \frac{\#\text{samples satisfying } \mathcal{R}_{f_i}}{\#\text{samples whose path conditions to } f_i \text{ are } \texttt{sat}}.$$

Benefits. This strategy offers three concrete advantages: it yields much simpler constraints by avoiding costly joins of diverging path clauses at merge points; it dramatically reduces solver time because each Z3 query involves only a single path condition; and it exposes abundant parallelism, since the independent samples can be processed concurrently across many cores or even an entire compute cluster.

References

[1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[2] 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* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 461–471. doi:10.1145/2970276.2970310
[3] David A. Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: correctness checking for real code. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 49–64.
[4] David A Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. 2000. A first step towards automated detection of buffer overrun vulnerabilities.. In *NDSS*, Vol. 20. 0.

Received 6 June 2025