

# Past-Sensitive Pointer Analysis for Symbolic Execution

Anonymous Author(s)

## ABSTRACT

We propose a novel fine-grained integration of pointer analysis with dynamic analysis, including dynamic symbolic execution. This is achieved via *past-sensitive pointer analysis*, an on-demand pointer analysis instantiated with an abstraction of the dynamic state on which it is invoked.

We evaluate our technique in three application scenarios: chopped symbolic execution, symbolic pointer resolution, and write integrity testing. Our preliminary results show that the approach can have a significant impact in these scenarios, by effectively improving the precision of standard pointer analysis with only a modest performance overhead.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**;

## KEYWORDS

Symbolic Execution, Pointer Analysis

### ACM Reference Format:

Anonymous Author(s). 2020. Past-Sensitive Pointer Analysis for Symbolic Execution. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. ACM, New York, NY, USA, 11 pages.

## 1 INTRODUCTION

We present a novel technique for increasing the precision of *pointer analysis* [27, 34, 37, 38] when used in the context of *dynamic analysis and dynamic symbolic execution*.<sup>1</sup> We show that the increased precision, which comes with a modest performance overhead, can benefit three applications: chopped symbolic execution, symbolic pointer resolution, and write integrity testing.

*Our Approach.* Existing dynamic analyses which employ static pointer analysis run the pointer analysis first and then utilize its results [2, 6, 12, 28, 41]. We propose a more fine-grained integration: invoke the pointer analysis on-demand whenever the symbolic exploration reaches a certain program point<sup>2</sup> where the points-to information is needed, and at that point instantiate the analysis with a *path-specific* abstraction of the current symbolic state. Specifically, our abstraction distinguishes objects already allocated in the past, which are assigned a unique allocation site, from objects allocated in the future, which can share allocation sites, to an extent

<sup>1</sup>We note that from the perspective of running a pointer analysis in a dynamic context, the dynamic analysis scenario is simply a particular case of the dynamic symbolic execution one in which a single path is explored and no symbolic data (in particular symbolic pointers) are present. Therefore, in the remaining of the paper we mainly discuss our technique in the more general context of symbolic execution.

<sup>2</sup>Technically, we only invoke pointer analysis at procedure call sites.

depending on the sensitivity of the pointer analysis. We call our analysis *past-sensitive pointer analysis* (PS<sub>PA</sub>).

*Applications.* Many dynamic analysis techniques, such as various forms of integrity enforcement [2, 6, 12] and different extensions of dynamic symbolic execution [28, 36, 41] rely on the results of a pointer analysis. In our paper, we specifically explore *Chopped Symbolic Execution (CSE)* [41], *Symbolic Pointer Resolution* [28], and *Write Integrity Testing (WIT)* [2].

*Chopped Symbolic Execution.* Dynamic symbolic execution (DSE) [11] computes an under-approximation of the program's behavior by systematically exploring *multiple*, but not all, program paths. Unfortunately, scaling symbolic execution to handle large programs is challenging due to the well-known state-explosion problem. Thus, attempts have been made to utilize information gained by pointer analysis during the symbolic exploration [28, 36, 41].

Chopped symbolic execution [41] is a DSE variant that can skip calls to functions that users deem as irrelevant to the code they want to analyse. To safely do so, the analysis relies on the results of a pointer analysis, which provides conservative information about the *side effects* of the skipped function calls, i.e. the memory locations that may be modified by their code. If those locations are later read by the analyzed code, a *recovery* process takes place, in which relevant parts of the skipped code are executed. Of course, the more precise the results of the pointer analysis, the fewer unnecessary recoveries take place and thus the more effective the technique. We show that by running the pointer analysis in the symbolic state just before skipped function calls, the number of recoveries can be significantly reduced (§6.2).

*Symbolic Pointer Resolution.* Symbolic pointers present a particular challenge for DSE [7, 28]. Modern DSE systems typically map each symbolic memory object into a different solver array. Queries involving symbolic memory objects are then easily translated into SMT constraints involving the corresponding SMT arrays. Since symbolic pointers can potentially refer to multiple memory objects, the DSE system first needs to find all the memory objects to which the pointer could refer to, so that the right SMT arrays can be referenced. In one of the most popular memory models, the *forking model* [7, 9, 32], the DSE system scans each memory object in turn, issuing solver queries to determine if the pointer can refer to that memory object. If a pointer analysis determines that the pointer cannot refer to an object, that object can be ignored, saving potentially expensive solver queries. In this paper, we show that our approach can significantly speed up symbolic pointer resolution, by eliminating a much larger number of solver queries compared to a standard static pointer analysis (§6.4).

*Write Integrity Testing (WIT).* WIT [2] is a well-known defense against certain classes of security attacks. At a high level, WIT uses a pointer analysis to determine which pointers are allowed to access which objects. For instance, the pointer analysis might

determine that a pointer  $p$  can only refer to objects  $a$  and  $b$ . Dynamic instrumentation is then added to enforce the results of the static analysis—e.g., if  $p$  is used to access another object  $c$  via a buffer overflow, execution is safely terminated. Of course, the precision of the pointer analysis has a direct impact on the effectiveness of the analysis. In our experiments with WIT [2], we show that computing the analysis in the dynamic context where the program has already finished its initialization can lead to significant improvements in precision and thus effectiveness (§6.3).

*Main contributions.* Our main results can be summarized as follows:

- (1) We propose *past-sensitivity*—a new form of sensitivity in pointer analysis which makes use of the dynamic context under which it is invoked.
- (2) We describe a technique for generating path-specific pointer abstractions in the context of dynamic analysis and symbolic execution.
- (3) We provide an implementation based on the state-of-the-art symbolic execution engine KLEE [7], which we make available as open source.<sup>3</sup>
- (4) We show the benefits of our technique in three different scenarios: chopped symbolic execution, symbolic pointer resolution, and write integrity testing.

## 2 OVERVIEW OF OUR APPROACH

We demonstrate our approach by applying it to the symbolic execution of the simple program shown in Figure 1.<sup>4</sup> First, the program allocates two hash tables ( $t1$  and  $t2$ ), and inserts even elements into  $t1$  and odd elements into  $t2$  (line 52). At line 55 it inserts a new element to table  $t1$  with the symbolic key  $k1$ . The insertion function uses `table_lookup` to check if the element already exists in the table, which in turn computes the hash of the input key, and iterates over the nodes of the relevant bucket to find the matching element.

Assume that a developer is interested in testing the part of the code operating on table  $t2$ . The developer could try to reduce path explosion (e.g., due to the while loop at lines 20–23) by skipping the invocation of function `table_insert` at line 55 using chopped symbolic execution. However, an attempt to skip the invocation at line 55 using the original technique, which relies on flow- and context-insensitive pointer analysis, will not be successful, since the analysis will report a false dependency between the side effects of `table_insert` at line 55 and the locations read by `table_lookup` at line 56. The relevant part of the abstraction computed by the (whole-program) pointer analysis is shown in Figure 2. Note that a context- (and flow-) sensitive pointer analysis will not solve our problem, since the elements of both tables are allocated in the same context (at line 30 of `table_insert`, which is called from line 52).

Leveraging the fact that we need the side effects information during the execution (in particular before the call at line 55), and not necessarily at its start, we aim to take advantage of the dynamic information at hand. In order to run pointer analysis from a dynamic

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define N 20
6 typedef struct elem_t
7     {unsigned k; char *v; struct elem_t *next;} elem_t;
8 typedef struct {unsigned n; elem_t **array;} table_t;
9
10 table_t *table_alloc(unsigned n) {
11     table_t *t = malloc(sizeof(table_t));
12     t->array = calloc(sizeof(elem_t *), n);
13     t->n = n;
14     return t;
15 }
16
17 elem_t *table_lookup(table_t *t, unsigned k) {
18     unsigned int hash = k % t->n;
19     elem_t *e = t->array[hash];
20     while (e) {
21         if (e->k == k) break;
22         e = e->next;
23     }
24     return e;
25 }
26
27 void table_insert(table_t *t, unsigned k, char *v) {
28     elem_t *e = table_lookup(t, k);
29     if (!e) {
30         e = malloc(sizeof(elem_t));
31         e->v = malloc(10);
32         e->k = k;
33         int hash = k % t->n;
34         e->next = t->array[hash];
35         t->array[hash] = e;
36     }
37     strcpy(e->v, v);
38 }
39
40 void run(table_t *t) {
41     while (...) {
42         // wait for key and data
43         table_insert(t, k, v);
44     }
45 }
46
47 void main() {
48     table_t *t1 = table_alloc(N);
49     table_t *t2 = table_alloc(N);
50     for (unsigned i = 0; i < N; i++) {
51         table_t *t = i % 2 == 0 ? t1 : t2;
52         table_insert(t, i, "...");
53     }
54     unsigned k1, k2; // symbolic
55     table_insert(t1, k1, "foo");
56     elem_t *e = table_lookup(t2, k2);
57     run(t2);
58 }

```

Figure 1: Motivating example.

context, we need the ability to compute the abstraction of the current symbolic state. A key observation is that we can benefit from the ability to distinguish between objects which were already allocated when the analysis is invoked, even if they have the same static allocation site.

<sup>3</sup>Link omitted for double-blind review.

<sup>4</sup>The example demonstrates code patterns that we encountered in real code: *libtasn1* creates a tree-like data structure where nodes are allocated inside a loop and *m4* utilizes multiple hash tables.

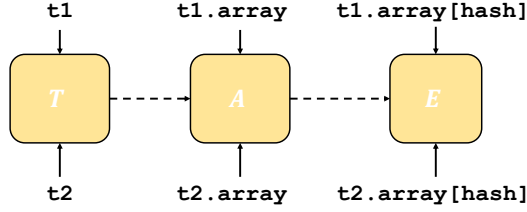


Figure 2: Abstraction with static pointer analysis.

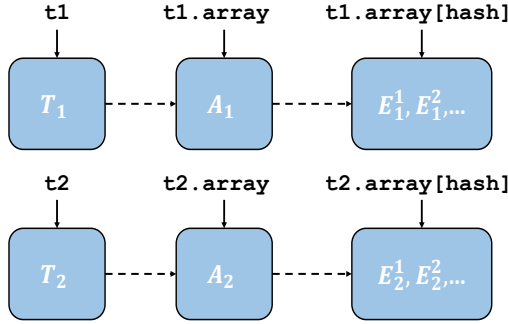


Figure 3: Abstraction with past-sensitive pointer analysis.

The abstraction computed using our PS<sub>PA</sub> approach, at the call-site of `table_insert` at line 55, is shown in Figure 3. For each of the objects allocated until that point at lines 11, 12 and 30, we assign a *unique allocation site* in the abstract state. Therefore, when function `table_insert` is called at line 55, we know from the dynamic abstraction that: `t1` points to  $T_1$ , the field `array` of `t1` points to  $A_1$ , and the pointers of that array point (index-insensitively) to  $\{E_1^1, E_1^2, \dots\}$  (the elements that were already allocated at line 30). Using this information, we can compute precise enough side effects for the function `table_insert`, which allows us to eliminate the spurious data dependencies.

Another challenge arising from our example is the process of symbolic pointer resolution, a well-known challenge in dynamic symbolic execution [7, 28]. Note that in our example, when at line 55 `table_insert` calls `table_lookup`, the value of the pointer `e` at line 19 is symbolic, since it depends on the symbolic hash value which is derived from `k1`. Therefore, at line 21, `e`→`k` dereferences a symbolic pointer.

Typically, symbolic executors create one SMT solver array for each symbolic memory object [7, 32]. This approach makes it difficult to map dereferences of symbolic pointers to solver arrays, as each symbolic pointer may refer to multiple objects. As discussed in the introduction, one of the most common approaches for handling symbolic pointers is the *forking model*, used by e.g., KLEE [7]: when a symbolic pointer is encountered, the entire memory is scanned to find all the objects to which the pointer could refer. Then, for each possible object, the pointer is constrained to refer to that object

only, making it straightforward to express SMT constraints involving that pointer. Determining whether the symbolic pointer can refer to a certain memory object is expensive, as it involves solver queries.<sup>5</sup>

We remark that the blind process of memory scanning can be improved if one has points-to information in hand. If the symbolic pointer *cannot* statically point to an object which has an allocation site AS, then objects whose allocation site is AS can be ignored in the scanning process, thus saving solver queries. Obviously, the success of this approach depends directly on the precision of the pointer analysis. Consider the symbolic pointer dereference `e`→`k` discussed above. With static pointer analysis, the scanned space will be reduced to the objects whose allocation site is E (Fig. 2), i.e. the elements allocated by both tables (`t1` and `t2`). With our PS<sub>PA</sub> approach, the scanned space will be further reduced to objects whose allocation sites is one of  $\{E_1^1, E_1^2, \dots\}$  (Fig. 3), thus reducing the number of scanned objects by a factor of 2.

So far we discussed the benefits of our dynamic approach in the context of symbolic execution. Now we will show the benefits of our approach in the context of write integrity testing, which we introduced in §1. Consider the execution of the `run` function, which waits for a key and a value (of an arbitrary size), and inserts them to the table `t2` using `table_insert`. Note that if our value `v` is long enough, then a buffer overflow will occur during the execution of `table_insert` at line 37.

When WIT is enforced with static pointer analysis, all the string buffers allocated at line 31 correspond to a single static allocation site, and WIT only enforces that the `strcpy` accesses one of the string buffers allocated in line 31. Therefore, when the `strcpy` function overrides the next object in memory (which could be another string buffer), a buffer overflow can be missed. With our dynamic approach, we can perform the pointer analysis just before the invocation of `run`, which enables us to distinguish between each of the already allocated values, and those that will be allocated in the future. In particular, we can distinguish between values allocated for `t1` and `t2`. Since `run` only references `t2` and we track each past-allocated value, we can distinguish the different values associated with `t1`. Thus under some allocation schemes, e.g., when all string buffers are allocated sequentially, these distinctions allow PS<sub>PA</sub> to detect potential `strcpy` buffer overflows that are missed when using a standard static pointer analysis.

### 3 SYMBOLIC STATE API

*Symbolic execution* (SE) [11] is an automated technique for evaluating the behavior of a program on arbitrary inputs. It has been successfully used for e.g., test generation [21], bug finding [9], and security vulnerability discovery [13]. The idea behind symbolic execution was introduced over 40 years ago [5, 17, 29]. However, in the last decade, symbolic execution gained new momentum due to the dramatic improvement in SMT solvers [19], and the invention of dynamic/concolic symbolic execution [8, 21].

Roughly speaking, SE engines such as KLEE [7] systematically explore program paths by maintaining *symbolic states*. Each such

<sup>5</sup>Note that this process can be optimized to scan only part of the memory space [10, 20], but tools like KLEE don't use these optimizations as they pose some implementation challenges. Other memory models can also work better in some scenarios. [28].

state records the values of variables and heap-allocated locations as path-dependent functions of a given *symbolic* input and a *path condition*. The latter is a conjunction of quantifier-free first-order formulas that tracks the sequence of branches the SE engine followed along that path. The path is feasible if and only if the resulting condition is satisfiable.

Modern SE engines like KLEE compute symbolic states which reflect the layout and the contents of the program's memory with bit-level accuracy: Every time the SE engine encounters a memory-allocation command, be it of a local variable, a global variable or a heap object, it adds to the symbolic state a fresh *memory object*  $mo$  spanning an appropriate number of bytes starting at a unique *base address*  $\text{Base}_s(mo) \in \mathbb{N}$ .<sup>6</sup>

To simplify the presentation, we assume that the SE engine represents every memory object as if it is comprised of a sequence of *primitive fields* containing *primitive values*, which are either integers or addresses. We represent the contents of each primitive field by a unique *symbolic expression*  $e$ .<sup>7</sup>

We further expect that every memory object  $mo$  in symbolic state  $s$  has a *type*  $t \in T$ , denoted by  $\text{Type}_s(mo)$ , which determines the number of its primitive fields and their types (integer or address). The set of possible types is defined as follows:

$$T := \text{int} \mid \text{ref} \mid \text{Array}(n, T) \mid \text{Struct}(\bar{T})$$

The types  $\text{int}$  and  $\text{ref}$  correspond to integer and pointer primitive values occupying a single field. The type  $\text{Array}(n, t)$  is of an array containing  $n \in \mathbb{N}^+$  elements of type  $t$ , and the type  $\text{Struct}(t_1, \dots, t_n)$  indicates a record type containing  $n$  elements where the  $i$ -th element, for  $1 \leq i \leq n$ , is of type  $t_i$ . We denote the number of fields in an object of type  $t$  by  $\text{Size}(t)$  and use  $\text{Size}_s(mo)$  as a shorthand for  $\text{Size}(\text{Type}_s(mo))$ .

We assume that the SE engine exposes the set of allocated memory objects in a symbolic state  $s$ , their properties, and the value of pointer expressions. Specifically, we expect that the SE engine supports the following operations on a given symbolic state  $s$ :

- $\text{MO}_s$  returns the set of allocated memory objects in  $s$ .
- $\text{Base}_s$ ,  $\text{Type}_s$  and  $\text{AS}_s$  return the base address, type, and allocation site<sup>8</sup> of  $mo \in \text{MO}_s$ , respectively.
- Given a memory object  $mo$ , and a field index  $0 \leq i < \text{Size}_s(mo)$ , the functions  $\text{Type}_s(mo, i)$ , and  $\text{E}_s(mo, i)$  return the type of the  $i$ -th primitive field of  $mo$  and the symbolic expression representing its value, respectively.
- Given a symbolic expression  $e$  and a constant value  $c$ , the function  $\text{mayBeTrue}_s(e = c)$  determines whether it is possible that  $e$  has the value  $c$  in  $s$ .

## 4 PAST-SENSITIVE POINTER ANALYSIS

In this section, we present our technique for determining the abstraction when invoking pointer analysis from dynamic contexts. The technique abstracts the *memory graph* induced by the symbolic state using a *past-sensitive abstract domain*. Thus, we describe these aspects first.

<sup>6</sup>If the size  $N$  of the allocation is symbolic, the engine concretizes  $N$  to an arbitrary *admissible* size, i.e., one which adheres to the accumulated path constraints.

<sup>7</sup>We note that our implementation is bit-accurate.

<sup>8</sup>We assume that every variable and heap object is allocated in a unique line of code, as it is the case in programs written in LLVM bitcode [30].

## 4.1 Memory Graphs

The *memory graph* pertaining to a symbolic state  $s$ , denoted by  $G_s$ , is a graph which conservatively represents the possible memory layout in  $s$ : the nodes of the graph are the primitive fields of the allocated memory objects and its edges record the possible points-to relations. Given the symbolic state API described in the previous section, constructing the graph is rather straightforward:

$G_s = (N, E)$ , where

$$N = \{(mo, i) \mid mo \in \text{MO}_s \wedge 0 \leq i < \text{Size}_s(mo)\}$$

$$E = \{((mo_1, i_1), (mo_2, i_2)) \mid$$

$$mo_1, mo_2 \in \text{MO}_s \wedge \text{Type}_s(mo_1, i_1) = \text{ref} \wedge$$

$$\text{mayBeTrue}_s(\text{E}_s(mo_1, i_1) = \text{Base}_s(mo_2) + i_2)\}$$

Given a symbolic expression  $e$  and a symbolic state  $s$ , we denote by  $\text{Targets}_s(e)$  the set of nodes in the memory graph of  $s$  corresponding to the locations (primitive fields) that  $e$  might point to:

$$\text{Targets}_s(e) = \{(mo, i) \mid mo \in \text{MO}_s \wedge 0 \leq i < \text{Size}_s(mo) \wedge \text{mayBeTrue}_s(e = \text{Base}_s(mo) + i)\}$$

Let  $R$  be a set of symbolic pointer expressions. We denote by  $G_s(R)$  the subgraph of the memory graph of  $s$  obtained by projecting  $G_s = (N, E)$  nodes of some expression in  $R$ :

$$G_s(R) = (N', E \cap (N' \times N')), \text{ where}$$

$$N' = \{n \in N \mid n \text{ is reachable in } (N, E)$$

$$\text{from a node in } \bigcup_{e \in R} \text{Targets}_s(e)\}.$$

Note that when a function is invoked, it can only operate on the part of the memory which is reachable from its formal parameters and global variables. Thus, assuming  $R$  contains the symbolic expressions corresponding to the addresses of the global variables and the values of the actual parameters in  $s$ , we can soundly analyze the invocation considering only part of the memory represented by  $G_s(R)$  instead of  $G_s$ .

For example, the memory graph of the symbolic state  $s$  computed at line 55 of Figure 1 right before the invocation of `table_insert` contains several nodes: local variables (`t1`, `t2`, ...), allocated tables, allocated arrays, etc. The part of the memory graph relevant to the invocation  $G_s(\{t1\})$ , contains only string "foo" and the elements and the string buffers in table `t1`.

## 4.2 Past-Sensitive Abstract Domain

Pointer analysis algorithms conservatively determine the target of every pointer variable and field in the program. The most common way to represent this information is to use a *points-to graph*. A points-to graph is comprised of nodes representing (fields) of memory objects and the edges which encode the possible contents of the pointer fields. In addition, there are several *roots*, representing the entry points to the graph—the addresses of variables and the pointer parameters passed to the program.

Intuitively, one can think of a points-to graph  $D$  as abstracting a memory graph by collapsing multiple nodes of the memory graph into single nodes which can represent one or more memory fields. To ensure the abstraction is bounded, and thus that the points-to algorithm terminates, there is a bounded a priori fixed partitioning of the nodes. For example, all the heap objects allocated at the same allocation site are often collapsed together into the same node. Different points-to abstractions alter the graph in different ways. For



example, they may refine the abstraction by differentiating between objects allocated at different calling contexts (object-sensitivity) or coarsen it by ignoring the distinction between the different fields of an object (field-insensitivity).

In PSpA, we also use an allocation-based pointer abstraction, but with a twist: As we know the exact dynamic context in which the analysis should operate, we distinguish between the objects which were allocated prior to the analyzed invocation and the ones the analysis should consider as being possibly allocated from this point on. Thus, *every dynamic context induces its own abstraction*. Technically, our abstract domain  $D$  is defined over *admissible abstract states* as formalized below:

$$\begin{aligned} \mathcal{AS} \in \mathcal{AS}_{static} &= \text{The set of static allocation sites in } P \\ \mathcal{AS}_{unique} &= \mathcal{AS}_{static} \times \mathbb{N} \\ as \in \mathcal{AS} &= \mathcal{AS}_{static} \cup \mathcal{AS}_{unique} \\ v \in \mathcal{V} &= \text{Roots} \\ o \in \mathcal{O} &= \mathcal{AS} \times (\mathbb{N} \cup \{*\}) \\ D \in \mathcal{D} &= 2^{\mathcal{V}} \times 2^{\mathcal{O}} \times 2^{(\mathcal{V} \cup \mathcal{O}) \times \mathcal{O}} \end{aligned}$$

An *abstract state* (aka *points-to graph*)  $D = (V, O, PT) \in \mathcal{D}$  is comprised of a set of *roots*  $V$ , the program's entry points into the memory; a set of *abstract nodes*  $O$  representing memory locations; and a set of *abstract edges*  $PT$  representing the points-to relations. A *root*  $v \in \mathcal{V}$  is the address of a global variable or the value of a pointer parameter. An *abstract node*  $o = (as, f) \in \mathcal{O}$  is comprised of an *allocation site*  $as \in \mathcal{AS}$  and a *field index*  $f$  for memory objects which are abstracted in a field-sensitive manner, or an allocation site and the special symbol  $*$  for those which are not.

Allocation sites are the lines of code in the program in which a memory object is allocated. As explained above, during the analysis, an allocation site can be either *static*  $as \in \mathcal{AS}_{static}$ , potentially representing multiple objects, or *unique*  $(as, n) \in \mathcal{AS}_{unique}$  representing a single object allocated at allocation site  $as$  and residing in address  $n$ . The abstraction of a symbolic state, however, only generates nodes with unique allocation sites for already allocated objects in the heap. For example, in the program at Figure 1, each object allocated at line 11 will have a unique allocation site, and therefore the pointer analysis algorithm will be able to distinguish between the elements in tables  $t_1$  and  $t_2$ .

Our analysis aims to be field-sensitive, i.e., differentiate between the pointer values of different fields. Thus, an abstract memory object  $(as, f)$ , where  $f \in \mathbb{N}$ , conservatively represents the contents of the  $f$ -th field of an object represented by allocation site  $as$ . In case, however, the analysis cannot distinguish between different fields of the objects represented by  $as$ , it degenerates into a field-insensitive abstraction which uses a single memory object  $(as, *)$  to represent all the fields of the relevant objects. Clearly, an abstract memory object can be represented only in either a sensitive or an insensitive way. Thus, an abstract state  $D = (V, O, PT)$  is *admissible* if  $\forall as, f. (as, f) \in O \wedge (as, *) \in O \implies f = *$ . In the rest of the paper, we assume that all abstract states are admissible unless explicitly stated otherwise.

To define our abstract domain, we need to induce an *approximation order*  $\sqsubseteq$  over it. An abstract memory state  $D_1$  is more precise than  $D_2$ , denoted by  $D_1 \sqsubseteq D_2$ , if  $D_2$  can be obtained from  $D_1$  by collapsing all the nodes representing fields of some memory objects into field-insensitive nodes, and, possibly, adding additional edges.

### 4.3 Abstraction Function

Past-sensitivity amounts to determining the initial state and the abstract domain for the pointer analysis by abstracting the memory graph of the symbolic state  $s$  on which the analysis is invoked into a points-to graph.

Essentially, every node in  $s$ 's memory graph is mapped to an abstract node and the edges of the memory graph induce the points-to relations, with the exception of nodes representing array elements: Often, the size of arrays is not known at compile time. Thus, points-to analyses are *index-insensitive*, i.e., they do not distinguish between different elements of an array. However, if the array elements are structures, the abstraction does not smash together all primitive fields; instead it distinguishes between the contents of the different fields of the structure. Another way to understand the resulting abstraction is to consider a memory graph and “squeeze” every array in every memory object to be of size one and add to that single element all the points-to edges emanating from the original array. Note that the points-to abstraction of the “squeezed” object can still be field-sensitive. For example, if the memory object is an array containing, say three `point_t` structures with fields `x` and `y`, and thus comprised of six primitive fields, the “squeezed” object would contain two fields: The first would represent all the primitive fields pertaining to the `x`-coordinate and the second to those representing the `y`-coordinate.

To formally define the abstraction function, we first introduce function  $\text{AbsFld}(t, f)$  which maps the  $f$ -th field of a memory object of type  $t$  to an appropriate field index of the abstract node:

$$\text{AbsFld}(t, f) = \begin{cases} 0 & f = 0 \wedge t \in \{\text{int}, \text{ref}\} \\ \text{AbsFld}(t', i) & t = \text{Array}(n, t') \wedge i = f \bmod \text{Size}(t') \\ \text{AbsFld}(t_k, i) + d & t = \text{Struct}(t_1, \dots, t_n) \\ & \wedge 0 \leq k < n \wedge c = \sum_{j=1}^k \text{Size}(t_j) \\ & \wedge c \leq f < c + \text{Size}(t_{k+1}) \\ & \wedge i = f - c \wedge d = \sum_{j=1}^k \text{AbsSize}(t_j) \end{cases}$$

where  $\text{AbsSize}$  is defined as follows:

$$\text{AbsSize}(t) = \begin{cases} 1 & t \in \{\text{int}, \text{ref}\} \\ \text{AbsSize}(t') & t = \text{Array}(n, t') \\ \sum_{i=1}^n \text{AbsSize}(t_i) & t = \text{Struct}(t_1, \dots, t_n) \end{cases}$$

Given a symbolic state  $s$ ,  $\text{AbsNode}$  maps every node  $(mo, i)$  in  $s$ 's memory graph to the abstract node which represents it:

$$\text{AbsNode}_s(mo, i) = (\text{AS}_s(mo), \text{AbsFld}(\text{Type}_s(mo), i)).$$

Given a symbolic state  $s$ , we use the symbolic state API to extract the memory graph of  $s$  and abstract it into an abstract state  $\text{Abs}(s) = (V, O, PT)$  (points-to graph). First, however, we remove irrelevant nodes: We assume that  $s$  occurs right before the invocation of a function. Thus, we remove any node in the memory graph which is not reachable from an actual parameter or a global variable  $g \in \text{Global}$ . Formally, we assume that the formal parameters of the invoked function are  $p_1, \dots, p_k$  and that the values of the actual arguments of the invocation are defined by the symbolic expressions

$e_{p_1}, \dots, e_{p_k}$ .<sup>9</sup> Similarly, we assume that the value of every global variable  $g \in \text{Global}$  is represented by the symbolic expression  $e_g$ . Thus, the abstraction is defined as follows:

$$\begin{aligned} \text{Abs}(s) &= (V, O, PT_V \cup PT_O), \text{ where} \\ V &= \{p_1, \dots, p_k\} \cup \text{Global} \\ O &= \{\text{AbsNode}_s(mo, i) \mid (mo, i) \in N\} \\ PT_V &= \{(v, \text{AbsNode}_s(mo, i)) \mid v \in V \wedge (mo, i) \in \text{Targets}_s(e_v)\} \\ PT_O &= \{(\text{AbsNode}_s(mo_1, i_1), \text{AbsNode}_s(mo_2, i_2)) \mid \\ &\quad ((mo_1, i_1), (mo_2, i_2)) \in E\} \\ (N, E) &= G_s(\{e_v \mid v \in V\}) \end{aligned}$$

Once the abstraction of the initial state  $s$  is computed, we run a standard pointer analysis algorithm to obtain points-to information which is sound with respect to any symbolic state that might arise during a symbolic execution of the analyzed function starting at  $s$ .

#### 4.4 Reusing Summaries

The number of function invocations during the symbolic exploration can be very high, and therefore re-running the pointer analysis for every invocation which needs to be analyzed might incur high performance overhead. We address this issue by creating points-to summaries and reusing them when possible.

The reuse mechanism is based on a greedy best-effort isomorphism-checking function `CHECKSTATEISOMORPHISM` which accepts a function  $f$ , two abstract states  $s_1$ , for which we have already executed the pointer analysis, and  $s_2$ , for which we need to determine the points-to graph resulting from invoking  $f$  on it. `CHECKSTATEISOMORPHISM` checks if the two states are isomorphic so that the results obtained for  $s_1$  can be used in the context of  $s_2$ . The check is done by finding matching pairs of abstract objects from the two states starting from the ones pointed to by the global variables and the function parameters, while taking into account the field-sensitivity of the investigated objects.

If the isomorphism check succeeds then `CHECKSTATEISOMORPHISM` returns an object-matching map which allows to translate the already computed points-to graph to the one needed. For example, in the context of chopped symbolic execution, this allows determining the side-effects (mod set) obtained when the analysis is invoked on  $s_2$  from the side effects computed for  $s_1$ . If it fails, we re-run the pointer analysis with  $s_2$  as the initial state and memoize the results for future isomorphism checks.

For space reasons, we do not further discuss the reuse mechanism, and refer the interested reader to the supplementary material.

### 5 IMPLEMENTATION

Our implementation is based on SVF [40] and KLEE [7]. SVF implements whole-program pointer analysis, thus we extend it to support local pointer analysis (from an arbitrary initial state), and unique allocation sites. We integrated the extended version of SVF on top of KLEE with support for LLVM 3.8. We make our implementation available as open-source (link omitted for double-blind review).

**Type Information.** In order to abstract symbolic states in a field-sensitive manner, we need to know the types of memory objects. Our implementation uses LLVM which often provides imprecise

<sup>9</sup>For simplicity, and without loss of generality, we assume that all the formal parameters are pointers. This assumption is justified as we ignore integer parameters.

**Table 1: Average size of mod-sets and ref-sets.**

	Mod-set		Ref-set	
	$S_{PA}$	$PS_{PA}$	$S_{PA}$	$PS_{PA}$
libosip	17.78	2.94	32.98	3.68
libtasn1	7.35	1.55	8.24	1.94
libtiff	140.16	12.95	126.63	17.44

type information at memory allocation sites. However, a program can set the type of an object using a cast instruction. Thus, we track type cast instructions during the symbolic execution in order to infer type information. If the type of an object cannot be determined, we conservatively represent it in a field-insensitive manner.

### 6 EVALUATION

In our experiments, we demonstrate the precision and scalability benefits of our approach in two ways: First, by examining the raw results of the pointer analysis (§6.1), and second, by evaluating the effectiveness of our approach in the context of three client analyses: chopped symbolic execution (§6.2), write integrity testing (§6.3), and symbolic pointers resolution (§6.4). We used an Intel i7-6700 machine with 8 cores and 32GB of RAM running Ubuntu 16.04. We disclose our data for peer review in the supplementary material.

#### 6.1 Precision

We first evaluate the added precision of our approach by examining its effect on the size of the computed points-to sets. In particular, we report the size of the *mod-sets* and the *ref-sets* for functions called during a symbolic execution run, i.e. how many objects the pointer analysis determines that the function and its callees can write to and read from, respectively.

As benchmarks, we used three popular libraries which parse various input formats: *GNU libosip* (11K SLOC), *GNU libtasn1* (19K SLOC) and *libtiff* (65K SLOC). *GNU libosip* is a library for parsing SIP messages, *GNU libtasn1* is a library for decoding and encoding data in the Abstract Syntax Notation One (ASN.1) format, and *libtiff* is a library for parsing TIFF images. We chose these libraries because we believe they represent programs which are a good fit for symbolic execution as they have rather complicated logic and require a relatively simple modeling of the environment.

We compare the precision of the analysis in two configurations: *static pointer analysis* ( $S_{PA}$ ), where we obtain the points-to information using a standard pointer analysis, and *past-sensitive pointer analysis* ( $PS_{PA}$ ), our approach for executing the pointer analysis from a dynamic context. In each configuration, we run our modified version of KLEE for a limited number of instructions with the DFS search heuristic (which is deterministic, and thus covers the same execution paths across configurations), and record the size of the mod-set and ref-set of called functions. We analyze only functions called from the application code, excluding the test driver and *uclibc* (KLEE’s version of the standard library) internal functions.

The precision improvement of  $PS_{PA}$  compared to  $S_{PA}$  is significant and can be seen in Table 1, which shows for each benchmark and mode the average size of the computed mod-sets and ref-sets. The sizes are expressed in terms of abstract objects; we strip away

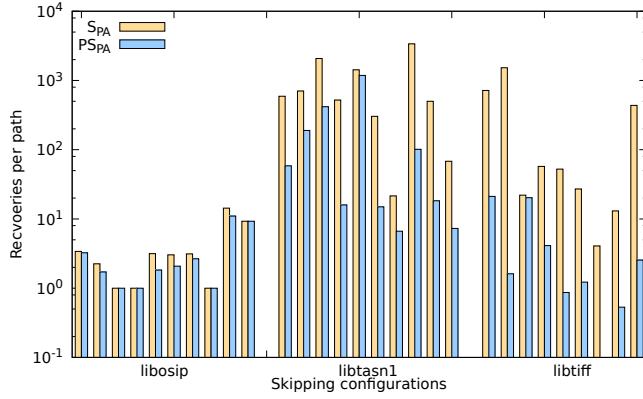


Figure 4: Recoveries per path (log scale).

the uniqueness of allocations sites, for the comparison with the original abstract domain to make sense.

## 6.2 Application: Chopped Symbolic Execution

To evaluate the impact of PS<sub>PA</sub> on chopped symbolic execution, we integrated the *Chopper* tool<sup>10</sup> into our code base.

Instead of computing the mod-set of a skipped function statically (as in the original technique), we use PS<sub>PA</sub> to compute the mod-set each time a skipped function is invoked. Remember from §1 that in chopped symbolic execution, whenever the program reads from the mod-set of a skipped function, a *recovery* procedure takes place to execute parts of the skipped function. An imprecise mod-set can lead to many unnecessary (and expensive) recoveries.

We performed several experiments. In §6.2.1, we show that our technique significantly reduces the number of recoveries needed in chopped symbolic execution due to an increase in the precision of pointer alias analysis. In §6.2.2, we then evaluate chopped symbolic execution in the context of test generation, showing that it achieves higher coverage when using PS<sub>PA</sub> instead of S<sub>PA</sub>. In §6.2.3 and §6.2.4 we show that PS<sub>PA</sub> can speed up chopped symbolic execution.

**6.2.1 Reducing the Number of Recoveries.** In this experiment, we show that past-sensitive pointer analysis can help reduce the number of recoveries during chopped symbolic execution.

We ran chopped symbolic execution with both S<sub>PA</sub> and PS<sub>PA</sub> mod-set computation. For each benchmark, we generate a list of called functions by running vanilla KLEE for ten minutes. We record only functions which are called from the main library API which the test driver invokes, and ignore *uclibc* internal functions. We then take ten random samples of ten functions to skip from that list, to generate ten different skipping configurations. For each skipping configuration, we ran chopped symbolic execution for ten minutes with the DFS search heuristic and count the number of recoveries.

Our random selection aims to remove the bias which may come from the selection of skipped functions. We pick functions from the ones already reached by KLEE, to make sure that the skipped functions we specify are indeed reached and skipped during execution.

<sup>10</sup><https://github.com/davidtr1037/chopper>

Table 2: Line coverage achieved by various configurations.<sup>11</sup>

	Search	S <sub>PA</sub>	PS <sub>PA</sub>	
			No reuse	Reuse
<i>libosip</i>	DFS	567	410	519
	Random	592	314	647
<i>libtasn1</i>	DFS	958	1079	1079
	Random	950	1019	1019
<i>libtiff</i>	DFS	669	673	673
	Random	647	677	1034

Figure 4 shows the number of recoveries per path executed for each skipped configuration (ten sets of ten randomly sampled functions for each of the three benchmarks). We can see that PS<sub>PA</sub> leads to a number of significant decreases in the number of recoveries. The largest decrease occurs in a configuration of *libtiff* where there are 43,344 recoveries with S<sub>PA</sub> and only 10 with PS<sub>PA</sub>, a reduction of 99%. The reduction for *libosip* ranges between 0% and 42%, for *libtasn1* between 17% and 99%, while for *libtiff* between 92% and 99%. The relatively poor performance of *libosip* only indicates that randomly skipping functions in *libosip* performs poorly compared to the other two benchmarks. However, PS<sub>PA</sub> is consistently outperforming S<sub>PA</sub> in each case.

**6.2.2 Improving Coverage.** In §6.2.1, we show that PS<sub>PA</sub> can help decrease the number of recoveries in chopped symbolic execution. As we don't expect to improve the structural coverage by randomly selecting the skipped functions, we ran an additional experiment where the function selection is done manually (note that [41] similarly performs the selection manually).

We then run chopped symbolic execution both with S<sub>PA</sub> and PS<sub>PA</sub>, and with PS<sub>PA</sub> without reuse to evaluate the impact of our reuse approach from §4.4. We use two heuristics: depth-first search (DFS) and random-path selection (Random) [7]. For each mode, we run chopped symbolic execution for one hour and compute the line coverage of the generated test suite using *gcov*.<sup>12</sup>

In *libosip*, we chose to skip functions which process different parts of the URI component of the SIP message. These functions are widely used in the library, and therefore the *context-insensitivity* of S<sub>PA</sub> leads to accumulation of side effects, which are propagated from all the call sites in the program. With PS<sub>PA</sub>, which starts the analysis from a much more precise initial state, we are able to locally analyze such functions, and avoid the noise which comes from considering other call sites.

In *libtasn1*, some of the skipped functions manipulate a tree data structure, which represents the structure of the parsed ASN.1 message. The nodes of this tree are dynamically allocated at the same allocation site, and therefore the analysis treats all the nodes in the tree as one node. Conversely, PS<sub>PA</sub> uses *unique allocation sites* for dynamically allocated objects, which allows us to distinguish the different nodes in the tree.

<sup>11</sup>The reader might notice the relatively small line coverage achieved in one hour. This is partly due to the large and challenging benchmarks, and partly because we only test a subset of the APIs in these libraries.

<sup>12</sup><https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>



**Table 3: The overhead of pointer analysis related computations. No R: without reuse, R: with reuse, RR: reuse ratio when run with reuse, N: number of static analysis invocations when run with reuse.**

	Search	No R	R	RR	N
<i>libosip</i>	DFS	97.87%	6.41%	99.99%	6,680,404
	Random	90.18%	2.66%	99.99%	919,326
<i>libtasn1</i>	DFS	3.06%	2.31%	98.86%	967
	Random	9.80%	3.54%	99.83%	6,800
<i>libtiff</i>	DFS	3.24%	0.04%	98.75%	401
	Random	95.74%	0.52%	99.99%	155,688

In *libtiff*, we skip the logging functions (warnings, errors, etc.) which receive some parts of the symbolic input and create many redundant forks. These functions are implemented using function pointers, and the imprecise *function pointer resolution* of  $SP_A$ , makes it impossible to skip these functions efficiently. With  $PS_{PA}$ , the precise initial state plays a critical role in the precise resolution of function pointers, which allows us to entirely skip these functions without triggering any recoveries.

The results are shown in Table 2. Chopped symbolic execution with  $PS_{PA}$  (and reuse) outperformed the static mode in five out of six cases, with an improvement between -9% and 60%. We manually investigated the case where  $SP_A$  achieved higher coverage, and found out that chopped symbolic execution with  $PS_{PA}$  had lower coverage in the skipped functions, but reached more than 100 lines in the relevant (non-skipped) code which were not reached with  $SP_A$ . Thus, the  $PS_{PA}$  mode had better results from the viewpoint of the application (generating test cases for non-skipped code).<sup>13</sup>

The effect of the reuse mechanism can be seen in Tables 2 and 3. Running without the reuse mechanism leads to higher overhead in all six cases, especially in *libosip*, where the number of analyzed functions is much higher. In the cases where the increase of the overhead was not significant, the coverage remained roughly the same. In *libosip*, where the increase of the overhead was high, the  $PS_{PA}$  mode without reuse was not able to improve upon the  $SP_A$  mode. Table 3 also shows the reuse ratio, that is the relative number of function invocations for which the result of the mod-analysis could be reused without re-running the pointer analysis algorithm. The high reuse ratio suggests that most of the called functions actually have the same points-to information (with respect to context and flow insensitive analysis). The SE engine creates states which have different data constraints, but in most of the states that doesn't imply different points-to information, therefore data and points-to information can be seen as mostly independent properties.

**6.2.3 CVE Reproduction.** In addition to the coverage experiments presented above, we wanted to understand the impact of  $PS_{PA}$  on the experiments presented in the paper introducing chopped symbolic execution [41]. The paper considers six vulnerable locations in *libtasn1*, manually selects a set of functions unrelated to the vulnerabilities to be skipped, and then reports the time taken to find

<sup>13</sup>It was difficult to automatically ignore the coverage of the skipped function calls, because some code is called from both a skipped and a non-skipped context.

an input that reaches the vulnerable locations with and without chopped symbolic execution.

We replicated the experiments using both  $SP_A$  and  $PS_{PA}$  in turn.<sup>14</sup> Our results are shown in the last two columns of Table 4. As can be seen,  $PS_{PA}$  leads to significant savings in some cases, e.g., from 04:23 to 00:37 for the third vulnerability when DFS is used; and from a timeout (set to one hour) to 10:25 for the fifth vulnerability when DFS is used. However, there are also some cases in which  $SP_A$  does slightly better, with the largest difference of 01:51 vs 02:03 for the second vulnerability when the random heuristic is used.

While trying to understand why  $SP_A$  does sometimes better, we realized that the search heuristic used in *Chopper* (which prioritizes non-recovery states, before invoking KLEE's underlying heuristic) has a large influence on the results. Columns 4 and 5 of Table 4 show the results when this heuristic is disabled. In this case,  $PS_{PA}$  consistently outperforms  $SP_A$  (with one small exception, 5 vs 4 seconds in one case), although the overall results are worse.

To remove the significant influence of search heuristics on execution time, we decided to design a series of experiments in which *Chopper* can exhaustively explore *all* the paths in the program (for a symbolic input of a given size). Since all the paths are explored, we can simply count the exploration time (with  $SP_A$  and  $PS_{PA}$  respectively), knowing that the search heuristic has no influence (other than the overhead of the search heuristic itself).

**6.2.4 All-path exploration.** As discussed above, we constructed test drivers for our benchmarks that ensure that all paths can be explored in under an hour with vanilla KLEE. We then run chopped symbolic execution with  $SP_A$  and  $PS_{PA}$  respectively, with a time limit of one hour. We skip the same functions as in §6.2.2.

The results are shown in Table 5. The  $SP_A$  mode times out for all benchmarks, due to a high number of recovery states, originating from the imprecision of whole-program static analysis. By contrast, the  $PS_{PA}$  mode achieves a significant speed-up relatively to vanilla KLEE: 7.9x in *libosip*, 18.9x in *libtasn1* and 3.6x in *libtiff*.

## 6.3 Application: WIT

As discussed in the introduction, Write Integrity Testing (WIT) [2] is a runtime security mitigation technique that aims to detect buffer overflows and other memory violations at runtime. WIT works by using a pointer analysis to assign colours to pointers and memory objects. A pointer and a memory object have the same colour if the pointer can alias the memory object. WIT then enforces—at runtime—that each pointer only writes to memory objects of its colour, otherwise it terminates the program.

The precision of pointer analysis is of utmost importance to WIT as it severely impacts how many buffer overflows it can detect: If two objects are assigned the same colour, a buffer overflow from one into the other cannot be detected.

The application of our technique to WIT comes from the observation that in a security context, we are not interested in monitoring with WIT all the paths of a program, just the ones that are attacker-controlled. In particular, the initialization code is typically

<sup>14</sup>We haven't managed to incorporate yet one of the optimizations from *Chopper*, the slicing optimization [41], but we are currently working on this. However, that optimization does not directly interact with the mod-ref computation, which is what  $PS_{PA}$  directly affects. We also found and fixed a bug in the SVF integration.



**Table 4: Replication of CVE reproduction experiments from [41], without the slicing optimization. Times are using the format *minutes:seconds* and the timeout (TO) is one hour.**

	CVE	Search	Chopping-aware heuristic			
			without		with	
			S <sub>PA</sub>	PS <sub>PA</sub>	S <sub>PA</sub>	PS <sub>PA</sub>
#1	2012-1569	Random	04:26	00:20	00:30	00:19
		DFS	04:57	01:46	00:11	00:06
		Coverage	03:54	00:19	00:23	00:23
#2	2014-3467 <sub>1</sub>	Random	TO	TO	01:51	02:03
		DFS	04:17	02:15	00:01	00:01
		Coverage	TO	TO	01:20	01:20
#3	2014-3467 <sub>2</sub>	Random	00:01	00:01	01:53	01:50
		DFS	TO	TO	04:23	00:37
		Coverage	00:01	00:01	01:51	02:02
#4	2014-3467 <sub>3</sub>	Random	TO	TO	02:30	02:12
		DFS	TO	TO	00:02	00:02
		Coverage	T.O	T.O	03:56	01:09
#5	2015-2806	Random	03:46	02:03	06:38	02:04
		DFS	TO	10:14	TO	10:25
		Coverage	02:20	01:00	03:27	01:00
#6	2015-3622	Random	00:03	00:03	00:03	00:02
		DFS	TO	07:25	07:16	06:33
		Coverage	00:04	00:05	00:03	00:03

**Table 5: The total execution time (in *minutes:seconds*), with a timeout of one hour.**

	Vanilla	S <sub>PA</sub>	PS <sub>PA</sub>
<i>libosip</i>	33:30	Timeout	04:16
<i>libtasn1</i>	41:29	Timeout	02:12
<i>libtiff</i>	32:40	Timeout	10:02

not controlled by the attacker, so one could run the pointer analysis after the initialization completes. This increases the precision of the analysis, especially since at that point various configuration options are fixed by the program. The disadvantage is that the instrumentation (see §1) needs to be finalized at runtime, which increases the initialization time.

An implementation of WIT is not publicly available and re-implementing WIT from scratch is difficult. Therefore, we decided to conduct our evaluation using two indirect measures: number of computed colours (a static measure), and number of transitions between the colours of the buffers allocated in memory (computed dynamically on a certain workload, assuming a sequential allocator). The number of colour transitions indirectly measures how many sequential overflows WIT could prevent. We restrict our measurements to heap-allocated objects.

More precisely, WIT can detect a sequential buffer overflow if adjacent buffers have different colours. Therefore, if the colour of

**Table 6: The number of WIT heap colours and transitions computed with different pointer analysis techniques.**

	Paths	Colours		Transitions	
		S <sub>PA</sub>	PS <sub>PA</sub>	S <sub>PA</sub>	PS <sub>PA</sub>
<i>libosip</i>	12,084,552	70	277	108,532,593	302,069,717
<i>libtasn1</i>	90,290	157	645	8,848,420	39,456,716
<i>libtiff</i>	300	1047	1101	1,938	1,938

an object being allocated is different from the colour of the previous object, we increment the number of colour transitions.

We compute colours both with S<sub>PA</sub> and PS<sub>PA</sub>, starting in the execution state after the initialization completes (we manually annotated the end of the initialization section, but one could also automate the process by checking when a user input is first read). We consider each field of a structure as a separate object with its own colour when counting the colour transitions.

We built this transition analysis inside KLEE. We run KLEE for one hour on each benchmark and record the number of paths KLEE explored under DFS. Then we run KLEE again up to that number of paths under DFS in both the S<sub>PA</sub> and PS<sub>PA</sub> modes. This ensures that both runs used the same paths to count the number of transitions. We report the sum of transitions across all the explored paths.

As shown in Table 6, PS<sub>PA</sub> was able to compute around 4 times more colours for the heap objects in *libosip* and *libtasn1*. For *libtiff* the increase in the number of colours is modest—the reason is that *libtiff* does not have any setup or initialization code, so our technique analyses almost the whole program.

For *libosip* and *libtasn1* the number of transitions increases by a factor of 2.8x and 4.5x respectively when using PS<sub>PA</sub> instead of S<sub>PA</sub>. This shows that PS<sub>PA</sub> can significantly increase WIT’s ability to detect sequential heap buffer overflows on these two benchmarks, as it now has the potential to detect several times as many overflows. For *libtiff* there is no difference in the number of transitions, stemming from the small difference in the number of colours. Note that for *libtiff* only 300 paths were executed in one hour, as its paths are significantly more complex for symbolic execution.

## 6.4 Application: Symbolic Pointer Resolution

In this section, we show how PS<sub>PA</sub> can be used to optimize the forking model for symbolic pointer resolution. As explained in §1 and §2, on a symbolic pointer dereference, the forking model scans each memory object in turn, issuing solver queries to determine if the pointer can refer to that memory object. If a pointer analysis determines that the pointer cannot refer to an object, that object can be ignored, saving potentially expensive solver queries.

Dynamically computing the points-to set of a symbolic pointer operand requires a snapshot of the symbolic state at the beginning of the current function. However, taking a snapshot at each function call is overly expensive. Instead, we rely on the observation—tested empirically on our benchmarks—that symbolic pointer dereferences only occur in a few functions. Therefore, we design a lazy snapshot mechanism that decides which functions to snapshot at runtime: The first time we encounter a symbolic pointer, we use the standard resolution algorithm and remember the called function. Next time

**Table 7: Symbolic pointer resolution experiments on *m4*, *make* and *SQLite*. Q: queries, RT: resolution time (%), ET: execution time (minutes), SA: static analysis time (%).**

		Q	RT	ET	SA
<i>m4</i>	Baseline	1,902	56%	49'	-
	SP <sub>A</sub>	1,836	55%	47'	0.15%
	PS <sub>PA</sub>	960	38%	34'	0.54%
<i>make</i>	Baseline	21,832	56%	65'	-
	SP <sub>A</sub>	18,872	52%	60'	0.16%
	PS <sub>PA</sub>	6,222	30%	41'	1.22%
<i>SQLite</i>	Baseline	7,726	28%	43'	-
	SP <sub>A</sub>	7,726	23%	51'	14.23%
	PS <sub>PA</sub>	1,166	5%	33'	0.51%

the same function is called, we take a snapshot at its beginning. Note that this approach has the advantage of having no overhead for programs that don't have symbolic pointer dereferences.

To evaluate our approach, we selected programs where symbolic pointers can be encountered. We selected *m4* (78K SLOC), a popular implementation of the *m4* macro processor included in most UNIX-based operating systems; GNU *make* (28K SLOC), one of the most popular build systems; and *SQLite* (127K SLOC), a well-known SQL database engine library. Specifically, these programs make extensive use of hash tables which are a prolific source for symbolic pointers. Each benchmark was run in three configurations: baseline, SP<sub>A</sub> and PS<sub>PA</sub>. For each run, we measured the following parameters: number of resolution queries, fraction of time spent in pointer resolution, total execution time and overhead of pointer analysis.

As can be seen in Table 7, SP<sub>A</sub> only slightly reduced the number of queries in the case of *m4* and *make*, therefore there was no significant reduction in resolution time and total execution time. Moreover, in the case of *SQLite*, the execution time of SP<sub>A</sub> was higher than the baseline, since static pointer analysis took almost 10 minutes. With PS<sub>PA</sub>, we reduced the number of queries up to a factor of 6.6x (in *SQLite*), which also results in a significant decrease in resolution time and execution time, while keeping the computation overhead of pointer analysis low (under 1.22%).

## 7 RELATED WORK

Pointer analysis is a core static analysis technique, with numerous applications [1]. As such, many research projects have focused on improving the precision of the technique by making it flow-sensitive [26], context-sensitive [42], object-sensitive [31], path-sensitive [44], as well as using hybrid combinations of the above techniques [39]. Previous analyses have also tried to make the analysis sensitive to the context in which objects are allocated [33]. However, as far as we know, we are the first to add dynamic context sensitivity to pointer analysis which allows much finer abstraction of the memory state at the call site than other methods. We also note that PS<sub>PA</sub> can use and enhance any underlying points-to analysis.

Procedure summaries are a standard technique for performing interprocedural analyses [35], and have been used to obtain modular pointer analysis, e.g., [14, 15]. Our reuse algorithm is similar in

spirit to these approaches. However, instead of using procedure summaries during the analysis we employ it only to top-level calls from which the analysis starts.

Wüstholtz and Christakis [43] propose a novel technique for targeted greybox fuzzing with online static analysis, which is used to guide the fuzzer toward recently modified parts of the program. In their online static analysis, the initial abstract state is computed by re-executing the path prefix in the abstract domain, which results in an over-approximation of all possible executions of that prefix. In our case, the initial abstraction state is computed directly from the current symbolic state, which is more precise. In addition, we extend our abstract domain with *unique allocation sites*, which brings context sensitivity into the static analysis.

Grech et al. [24] describe a hybrid dynamic/static pointer analysis for Java programs where static analysis is utilized to over-approximate the values of stack variables while the information about the structure of the heap is obtained dynamically. Specifically, the heap is abstracted using multiple heap snapshots obtained by profiling the program's heap [23] during the execution of selected test cases. The dynamic information allows to sharpen the results of the analysis and overcome challenging issues such as the handling of reflection, native code, and lambdas. However, the resulting analysis is not sound as the heap information is under-approximated. In contrast, our analysis produces sound results for all executions starting at the abstracted dynamic/symbolic state. Furthermore, Grech et al. [24] utilise an a priori fixed heap abstraction, whereas our analysis employs an input-state specific heap abstraction.

More broadly, there is a lot of work on combining static and dynamic analysis, including static analysis and dynamic symbolic execution. For instance, Csallner and Smaragdakis [18] combine a static checker with a test input generator to guide the latter toward the errors reported by the former, while Christakis et al. [16] combine partial static verification with dynamic symbolic execution by guiding the latter to check the unverified program executions from the former. Tighter integrations of static and dynamic analysis also exist, which work by alternating may and must analyses to simultaneously prove program properties and search for bugs [4, 22, 25]. Our work is similar in spirit to some of these combinations, but its main distinguishing feature is related to the need for a fine-grained connection between the concrete memory layout and the abstraction used by the pointer analysis.

Anand et al. [3] abstract symbolic states to reduce the search space for programs with recursive data structures, and Yorsh et al. [45] abstract concrete states to determine whether to terminate the analysis. We use a similar technique but for a different purpose: determine the abstract domain of the static pointer analysis.

## 8 CONCLUSIONS

We have presented past-sensitive pointer analysis, a new design point for pointer analysis that takes into account the dynamic context in which the analysis is invoked. A different pointer abstraction is computed for each dynamic context, with the abstraction being constructed just before the pointer analysis is needed.

We show that this novel design point offers significant benefits in three important application domains: chopped symbolic execution, write integrity testing (WIT) and symbolic pointer resolution.

## REFERENCES

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley.
- [2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'08)*.
- [3] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2009. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer* 11, 1 (01 Feb 2009), 53–67. DOI: <http://dx.doi.org/10.1007/s10009-008-0090-1>
- [4] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. 2008. Proofs from tests. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'08)*.
- [5] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT – A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices* 10, 6 (1975), 234–245.
- [6] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. 2008. *Data Randomization*. Technical Report MSR-TR-2008-120. Microsoft Research.
- [7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*.
- [8] Cristian Cadar and Dawson Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*.
- [9] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*.
- [10] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 1–38. DOI: <http://dx.doi.org/10.1145/1455518.1455522>
- [11] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)* 56, 2 (2013), 82–90.
- [12] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*.
- [13] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'12)*.
- [14] R. Chatterjee, B.G. Ryder, and W. Landi. 1999. Relevant Context Inference. In *Principles of Programming Languages (POPL)*.
- [15] Ben-Chung Cheng and Wen-Mei W. Hwu. 2000. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'00)*.
- [16] Maria Christakis, Peter Müller, and Valentin Wüstholtz. 2006. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *Proc. of the 28th International Conference on Software Engineering (ICSE'06)*.
- [17] Lori A. Clarke. 1976. A Program Testing System. In *Proc. of the 1976 Annual Conference (ACM'76)*.
- [18] Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' Crash: Combining static checking and testing. In *Proc. of the 27th International Conference on Software Engineering (ICSE'05)*.
- [19] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Communications of the Association for Computing Machinery (CACM)* 54, 9 (Sept. 2011), 69–77.
- [20] Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. 2009. Precise Pointer Reasoning for Dynamic Test Generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)*.
- [21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'05)*.
- [22] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. 2010. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *Proc. of the 37th ACM Symposium on the Principles of Programming Languages (POPL'10)*.
- [23] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps Don't Lie: Countering Unsoundness with Heap Snapshots. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 68 (Oct. 2017), 27 pages. DOI: <http://dx.doi.org/10.1145/3133892>
- [24] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the Heap: Ultra-scalable Static Analysis with Heap Snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 198–208. DOI: <http://dx.doi.org/10.1145/3213846.3213860>
- [25] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. 2006. Synergy: A New Algorithm for Property Checking. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'06)*.
- [26] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *Proc. of the International Symposium on Code Generation and Optimization (CGO'11)*.
- [27] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proc. of the 2nd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*.
- [28] Timotej Kapus and Cristian Cadar. 2019. A segmented memory model for symbolic execution. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*.
- [29] James C. King. 1975. A New Approach to Program Testing. In *Proc. of the International Conference on Reliable Software (ICRS'75)*.
- [30] Chris Latner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*.
- [31] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering Methodology (TOSEM)* 14, 1 (Jan. 2005), 1–41.
- [32] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehltitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. (Sept. 2013).
- [33] John Plevyak and Andrew A. Chien. 1994. Precise Concrete Type Inference for Object-oriented Languages. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications (OOPSLA '94)*. ACM, New York, NY, USA, 324–340. DOI: <http://dx.doi.org/10.1145/191080.191130>
- [34] Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *Compiler Construction*, Görel Hedin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–137.
- [35] Micha Sharir and Amir Pnueli. 1981. *Two approaches to interprocedural data flow analysis*. Prentice-Hall, Englewood Cliffs, NJ, Chapter 7, 189–234.
- [36] Jiri Slaby, Jan Strejček, and Marek Trtik. 2013. Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution. In *Proc. of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*.
- [37] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. DOI: <http://dx.doi.org/10.1561/2500000014>
- [38] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. *Alias Analysis for Object-Oriented Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 196–232.
- [39] Yulei Sui and Jingling Xue. 2016. On-demand Strong Update Analysis via Value-flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 460–473. DOI: <http://dx.doi.org/10.1145/2950290.2950296>
- [40] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proc. of the 25th International Conference on Compiler Construction (CC'16)*.
- [41] David Trubish, Andrea Mattavelli, Noam Rinetzkzy, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proc. of the 40th International Conference on Software Engineering (ICSE'18)*.
- [42] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'95)*.
- [43] Valentin Wüstholtz and Maria Christakis. 2019. Targeted Greybox Fuzzing with Static Lookahead Analysis. CoRR abs/1905.07147 (2019). arXiv:1905.07147 <http://arxiv.org/abs/1905.07147>
- [44] Hua Yan, Yulei Sui, Shipping Chen, and Jingling Xue. 2018. Spatio-temporal Context Reduction: A Pointer-analysis-based Static Approach for Detecting Use-after-free Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 327–337. DOI: <http://dx.doi.org/10.1145/3180155.3180178>
- [45] Greta Yorsh, Thomas Ball, and Mooly Sagiv. 2006. Testing, Abstraction, Theorem Proving: Better Together!. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. 145–156.