Relocatable Addressing Model for Symbolic Execution

David Trabish
Tel Aviv University
Israel
davivtra@post.tau.ac.il

Noam Rinetzky
Tel Aviv University
Israel
maon@cs.tau.ac.il

ABSTRACT

Symbolic execution (SE) is a widely used program analysis technique. Existing SE engines model the memory space by associating memory objects with *concrete* addresses, where the representation of each allocated object is determined during its allocation. We present a novel addressing model where the underlying representation of an allocated object can be dynamically modified even after its allocation, by using *symbolic* addresses rather than concrete ones. We demonstrate the benefits of our model in two application scenarios: dynamic *inter*- and *intra*-object partitioning. In the former, we show how the recently proposed *segmented memory model* can be improved by dynamically merging several object representations into a single one, rather than doing that a-priori using static pointer analysis. In the latter, we show how the cost of solving array theory constraints can be reduced by splitting the representations of large objects into multiple smaller ones. Our preliminary results show that our approach can significantly improve the overall effectiveness of the symbolic exploration.

CCS CONCEPTS

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

KEYWORDS

Symbolic execution, Addressing model, Memory partitioning

ACM Reference Format:
David Trabish and Noam Rinetzky. 2020. Relocatable Addressing Model for Symbolic Execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397363

1 INTRODUCTION

Symbolic execution (SE) is a widely used program analysis technique, that lies at the core of many applications including automatic test generation [3, 4, 27], bug finding [15], debugging [19], patch testing [23, 29], automatic program repair [24, 25], and reverse engineering [7]. During symbolic execution, the program is run with a *symbolic* input, rather than a concrete one. Whenever the symbolic executor reaches a branch that depends on a symbolic value, an SMT solver [12] is used to determine the feasibility of the branch sides, and the appropriate paths are further explored while updating their paths constraints with a corresponding constraint. A test case for a given explored path can be generated by asking the solver to generate a concrete assignment to the accumulated path constraints.

In modern symbolic executors such as KLEE [3] and ANGR [33], the address space of a symbolic state is modeled using a set of memory objects, where each memory object has a fixed concrete address and its own unique and non-overlapping address range. This model is a reasonable implementation choice, but identifying memory object addresses with concrete values is not essential: As long as the non-overlapping property of the memory objects holds, the values of the assigned addresses should not affect the execution.

We propose a new addressing model where the address expressions observable by the symbolic state are symbolic values rather than concrete ones. We preserve the non-overlapping property by maintaining additional *address constraints*, which constrain each symbolic address value to some constant value. This addressing model gives us the ability to *relocate* a given memory object, that is, modify its underlying address constraint in a way that is *transparent* to the symbolic state. Note that relocating a memory object is not possible under the existing addressing model, since an object is allocated with a fixed constant address that can't be modified.

To illustrate the benefits of such addressing model, consider the program from Figure 1, inspired by code found in our benchmarks. First, the program executes an initialization loop, where at each iteration it creates a hash table (line 43) and initializes it with some values (line 45). Then, at line 49 it performs a lookup on one of the tables with a symbolic key *k*. The lookup function `table_lookup` computes the hash of the input key, and iterates over the nodes of the relevant bucket to find the matching element. When the function `table_lookup` is called at line 49, the value of the pointer `node` at line 18 is symbolic, since it depends on the symbolic hash value which is derived from the symbolic value `k`. Therefore, at line 20, `node->key` dereferences a symbolic pointer.

Symbolic pointers present a challenge for SE [4, 20]. Modern SE systems associate each memory object with a different SMT array. Queries involving memory objects are then translated into SMT constraints involving the corresponding SMT arrays. When a symbolic pointer is dereferenced, the SE engine needs to *resolve* that symbolic pointer, that is, determine the memory objects it can refer to. If the symbolic pointer is resolved to more than one memory object, then we are in the case of *multiple resolution*. Several memory models for handling multiple resolutions have been considered in the past: The *forking model* [3, 27] forks the current symbolic state for each of the resolved objects, and in each forked state constrains the symbolic pointer expression to the range of the

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397363

resolved object. This approach is relatively efficient in terms of constraint solving, but may contribute to path explosion due to forking. The *merging model* [15, 33] creates a disjunction with one disjunct for each of the resolved objects. This way, forks are avoided but the path constraints become more complex due to the introduction of disjunctions.

Similarly to the merging model, the *segmented memory model* [20] proposes an approach which avoids forking, but achieves this by using *array theory* [14] rather than disjunction. In this model, the memory is split into segments using static pointer analysis [18, 30, 34, 35], such that each pointer (concrete or symbolic) refers to objects in a single segment. The segments are computed as follows: First, static pointer analysis is invoked to compute the points-to set of each pointer, that is, a set of abstract memory objects which are identified by static allocation sites. Then, every two intersecting points-to sets are merged into one points-to set, until a fixpoint is reached, that is all the points-to sets are disjoint. A segment is created for each of these disjoint points-to sets, such that all the memory objects associated with that points-to set will be allocated in that segment.

With this approach, forks are avoided when symbolic pointers are encountered, as each pointer is guaranteed to point to exactly one segment. However, this approach is limited by the precision of pointer analysis, and the created segments might contain too many objects. A big segment corresponds to a big SMT array[1], which results in more complex constraints. To illustrate the limitations of pointer analysis, consider again the program at Figure 1. Pointer analysis can't distinguish between the different objects that are allocated at line 12, as they all have the same static allocation site. As a result, the bucket arrays of all 3 hash tables are allocated in one segment. Similarly, all the nodes allocated at line 33 are allocated in one segment as well. The SMT arrays of both segments are involved in the constraints, as both are accessed with a symbolic offset: The segment of buckets at line 18, and the segment of nodes at line 20. The sizes of these SMT arrays are at least three times bigger than those created by the forking model, due to merging of spurious objects. Therefore, despite of the reduction in the number of paths, the forking model still outperforms the segmented memory model in this case.

With our addressing model, we can dynamically relocate a memory object, in a way that is transparent to the symbolic state. Instead of determining the segments ahead of time, we create them on demand: If a symbolic pointer is resolved to multiple memory objects, we create a new segment and relocate the resolved memory objects to that segment. Now, a segment will contain only memory objects that can be pointed by a given symbolic pointer, without any spurious ones. The symbolic pointer at line 18 is resolved to one memory object, the bucket array of the first hash table, which doesn't require creating a segment. The symbolic pointer at line 20 is resolved only to nodes from the first hash table, so the created segment doesn't contain nodes from the other two hash tables. This way, we are able to avoid forking while creating smaller SMT arrays, which allows us to outperform both the segmented and the forking memory models.

[1]SMT arrays are actually unbounded, but the SE engine records the allocated size and never accesses elements beyond it.

Another challenge arising from the Program in Figure 1 relates to solving *array theory* [14] constraints. An array access with a concrete offset can be handled similarly to scalar variables, but accessing an array with a symbolic offset is more challenging, as the symbolic offset can refer to multiple locations in the array. In that case, the accessed value is expressed using an SMT formula over arrays, which creates a variable for each offset of the array. Such formulas are hard to solve, especially with big arrays, thus hindering symbolic execution. The hash tables created at line 43 are initialized with a bucket array of 300 entries, therefore the size of its corresponding SMT array is 2400 bytes. When calling `table_lookup` at line 49, the bucket is accessed with a symbolic offset at line 18, which triggers the usage of array theory. This constraint propagates into the path constraints that are created later during execution, thus slowing down the exploration.

With our addressing model, when a big enough memory object is accessed with a symbolic offset, we can relocate that memory object and split it into several smaller adjacent memory objects. For example, the symbolic pointer at line 18 was resolved to exactly one memory object, the bucket array of the hash table. Now, when this memory object is relocated and split, that symbolic pointer is resolved to multiple memory objects with smaller SMT arrays. In this case, despite of having more explored paths due to additional multiple resolutions, the reduced complexity of the constraints eventually results in faster exploration.

Our main contributions can be summarized as follows:

(1) We propose a new addressing model, that allows *seamless* and *dynamic* relocation of memory objects during symbolic execution.
(2) We provide an implementation based on KLEE [3], a state-of-the-art symbolic executor, which we make available as open source.[2]
(3) We show the benefits of our addressing model in two scenarios: improving the segmented memory model, and reducing the cost of solving array theory constraints with big arrays.

2 PROPOSED ADDRESSING MODEL

In this section, we shortly describe the current addressing model of SE engines and present our relocatable model.

2.1 Existing Addressing Model

Symbolic executors, e.g., KLEE, represent the program's address space using a set of *memory objects*:

$$mo = (addr, size, arr) \in 2^{\mathsf{N}^+ \times \mathsf{N}^+ \times \mathsf{A}}.$$

A memory object $mo = (addr, size, arr)$ has a concrete *base address* $addr \in \mathsf{N}^+$ and spans $size \in \mathsf{N}^+$ bytes. In addition, a memory object is backed by an SMT array $arr \in \mathsf{A}$ with the same size $size$, which tracks the values stored into the memory object. If a value v is written to the e-th byte of the object, the array arr of mo is replaced by new array generated using a *store* expression: $store(arr, e, v)$. If

[2]https://www.tau.ac.il/~davivtra/projects/ram/.

```
1  typedef struct node_s {
2      unsigned long data;
3      char *key;
4      struct node_s *next;
5  } node_t;
6  typedef struct {
7      node_t **buckets;
8      size_t size;
9  } table_t;
10
11 void table_init(table_t *t, size_t n) {
12     t->buckets = calloc(n, sizeof(node_t *));
13     t->size = n;
14 }
15
16 node_t *table_lookup(table_t *t, unsigned k) {
17     unsigned long h = hash(&k, sizeof(k)) % t->size;
18     node_t *node = t->buckets[h];
19     while (node != NULL) {
20         if (memcmp(&node->key, &k, sizeof(unsigned)) == 0) {
21             return node;
22         }
23         node = node->next;
24     }
25     return NULL;
26 }
27
28 void table_insert(table_t *t, unsigned k, int data) {
29     if (table_lookup(t, k)) {
30         return;
31     }
32     unsigned long h = hash(&k, sizeof(k)) % t->size;
33     node_t *node = calloc(1, sizeof(node_t));
34     node->key = k;
35     node->data = data;
36     node->next = t->buckets[h];
37     t->buckets[h] = node;
38 }
39
40 int main(int argc, char *argv[]) {
41     table_t tables[3];
42     for (unsigned i = 0; i < 3; i++) {
43         table_init(&tables[i], 300);
44         for (unsigned j = 0; j < 5; j++) {
45             table_insert(&tables[i], j, 7);
46         }
47     }
48     unsigned k; // symbolic
49     table_lookup(&tables[0], k);
50     return 0;
51 }
```

Figure 1: Motivating example.

the program accesses the e-th byte of the object, the read value is expressed using a *select* expression: $select(arr, e)$. [3]

The allocated objects span distinct addresses, i.e., for every two distinct memory objects $(addr_1, size_1, arr_1)$ and $(addr_2, size_2, arr_2)$ it holds that:

$$[addr_1, addr_1 + size_1) \cap [addr_2, addr_2 + size_2) = \emptyset.$$

[3] SE engines use an optimized representation of the memory object when the object is always accessed using concrete (non-symbolic) offsets. For simplicity, we avoid describing this optimization.

The non-overlapping requirement reflects the fact that different objects are located at different parts of the memory, and enables identifying memory objects by addresses: A concrete address *addr* can belong to at most one object. Thus, when the program accesses memory location *addr*, the SE engine can determine which SMT array represents the content at that address and act accordingly.

2.2 Relocatable Addressing Model

We propose a new addressing model, where the address of an object is a symbolic value, rather than a concrete one. As before, the program's address space is represented using a set of *memory objects*:

$$mo = (\alpha, size, arr) \in 2^{\mathsf{E}\times\mathsf{N}^+\times\mathsf{A}}.$$

However, the base address of an object is now a symbolic value $\alpha \in \mathsf{E}$ and not a concrete one. We enforce the non-overlapping property of different objects using *hidden* concrete base addresses: Whenever the program allocates a memory object, we create an *address pair* which consists of two values: a symbolic one α and a concrete value c. The concrete value c is used to ensure that the allocated objects do not overlap in the same way that is done in the existing model. The symbolic value α is the value that propagates to the symbolic state.

We maintain the correlation between the symbolic addresses and the concrete ones using *address constraints* (AC). These constraints, which are distinct from the path constraints of the symbolic state, record equalities of the form:

$$\alpha = e$$

where e is an expression over the hidden concrete addresses and the symbolic ones.

Note that the address constraints are not part of the path constraints, they are only used when we construct a query for the solver. Under this model, the address expressions stored in the symbolic state are symbolic, and any other expression might depend on these symbolic values, which are not constrained under the path constraints. Therefore, we substitute the address constraints before passing an expression e to the solver, that is: $e[\alpha_i/e_i]$ (for each address constraint $\alpha_i = e_i$).

Remark. Another way to achieve the non-overlapping property is to extend the path constraints of the symbolic state with appropriate constraints regarding the values of the base addresses. If we have memory objects $(\alpha_1, size_1, arr_1), \ldots, (\alpha_n, size_n, arr_n)$, then we could have used the following constraints:

$$\forall i \neq j. \quad [\alpha_i, \alpha_i + size_i) \cap [\alpha_j, \alpha_j + size_j) = \emptyset.$$

However, we found this approach not scalable, as it adds additional constraints and symbolic values (depending on the number of objects in the symbolic state), making constraint solving significantly harder.

To illustrate our new addressing model, consider the program from Figure 2. When the array of pointers is allocated at line 2, we do the following: Assuming that a pointer size is 8 bytes, we first create a new memory object $mo_1 = (\alpha_1, 16, arr_1)$ with an address pair (α_1, c_1), and add mo_1 to the address space. Then, we add a new address constraint $\alpha_1 = c_1$, and the symbolic value α_1 is assigned to be the value of the local variable `array`. Note that c_1 is chosen such that the addresses in the range $[c_1, c_1 + 16)$ are distinct from those

```
1 #define N (2)
2 char **array = calloc(N, sizeof(char *));
3 for (unsigned int i = 0; i < N; i++)
4     array[i] = calloc(256, 1);
5
6 unsigned int i; // symbolic, i < 2
7 unsigned int j; // symbolic, j < 100
8 if (array[i][j] == 1)
9     // do something...
```

Figure 2: A simple program allocating a two dimensional matrix using an array of pointers and multiple buffers.

of any already allocated object. If the memory objects allocated at line 4 are mo_2 and mo_3 with the corresponding address pairs (α_2, c_2) and (α_3, c_3), then before executing line 8 the address space consists of:

$$\{mo_1, mo_2, mo_3\}$$

and the address constraints are:

$$\{\alpha_1 = c_1, \alpha_2 = c_2, \alpha_3 = c_3\}$$

At line 8, where `array` is accessed with symbolic offset i, the value of $a[i]$ is a *select* expression:

$$select(store(store(arr_1, 0, \alpha_2), 1, \alpha_3), i)$$

This symbolic pointer expression has to be resolved using the solver, but instead of passing the above expression we substitute our address constraints and the actual expression passed to the solver is:

$$select(store(store(arr_1, 0, c_2), 1, c_3), i)$$

With this model, objects can be now seamlessly relocated. Suppose that after the loop at line 5 we want to relocate the memory object mo_2 to a new address. This is achieved by the following steps: First, we allocate a new memory object mo_4 with an address pair (α_4, c_4) of the same size as mo_2, and copy the contents of mo_2 to mo_4. Second, we update the address space by removing mo_2 and adding mo_4. Finally, we modify the address constraint $\alpha_2 = c_2$ to be $\alpha_2 = c_4$. After the relocation, the expression obtained by the symbolic read $a[i]$ at line 8 is the same as before:

$$select(store(store(arr_1, 0, \alpha_2), 1, \alpha_3), i)$$

But now, the substituted expression that will be passed to the solver is different:

$$select(store(store(arr_1, 0, c_4), 1, c_3), i)$$

In this addressing model the address values observable by the symbolic state are always symbolic, but those passed to the solver are concrete, which allows efficient constraint solving. Using this addressing model we can perform *dynamically* two operations which are not possible with the existing model: merging multiple objects into one segment (Section 2.3), and splitting an object to multiple objects (Section 2.4).

Limitations. Our symbolic addressing model is applicable for *well-behaved* programs where the actual address of a memory object should not affect the behavior of the program. In particular, the program should only compare pointers of different objects for equality, as required by the C standard, and avoid using fixed addresses besides *null*.

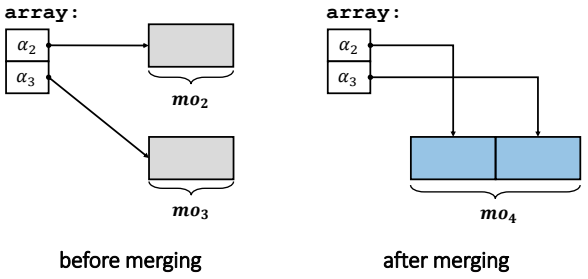

Figure 3: Merging multiple objects into a single segment.

2.3 Application: Inter-object Partitioning

In this section, we show how the relocatable addressing model allows to dynamically merge the representations of several objects allocated by the program into a single memory object, thus reducing the cost of handling symbolic pointers.

2.3.1 Segmented Memory Model. Symbolic pointers present a particular challenge for SE [4, 20]. Since symbolic pointers can potentially refer to multiple memory objects, the SE engine first needs to *resolve* the pointer expression, that is, find all the memory objects to which the pointer could refer to, such that the right SMT arrays can be referenced. The case of multiple resolution, where we have more than one resolved object, is especially challenging, and several approaches have been proposed in the past.

In the *forking model* [3, 27], the current symbolic state is forked for each of the resolved objects, and each forked state constrains the symbolic pointer expression to the range of the resolved object. This approach is relatively efficient in terms of constraint solving, but may contribute to path explosion due to forking. The *merging model* [15, 33] uses disjunction to constrain the symbolic pointer to one of the resolved objects. This way, forks are avoided but the path constraints become more complex due to the introduction of disjunctions. Similarly to the merging model, the *segmented memory model* [20] also proposes an approach which avoids forking, but here this is achieved by using array theory rather than disjunction. In this model, the memory is split into segments, such that each symbolic pointer refers to objects in a single segment. The memory is partitioned into segments using conservative pointer analysis. The *points-to* set of a pointer is a set of abstract memory objects which are identified by their static allocation site (line). Two intersecting points-to sets are merged into one points-to set, until all the points-to sets are disjoint. For each of the (disjoint) points-to sets a new memory segment is created, such that all the memory objects associated with that points-to set will be allocated in that segment.

With this approach, objects pointed by any pointer (symbolic or concrete) are guaranteed to reside in exactly one segment, which makes the process of symbolic pointer resolution much more efficient. However, this approach is limited by the precision of pointer analysis, and the computed segments might be too large for complex programs. A larger segment results in a larger SMT array, thus affecting the performance of the solver.

2.3.2 Dynamically Segmented Memory Model. Instead of conservatively computing the segments ahead of time using pointer analysis, we propose a dynamic memory partitioning strategy that creates the segments on the fly using our relocatable addressing model. When we encounter a symbolic pointer that refers to multiple memory objects $mo_1, ..., mo_n$ where:

$$mo_i = (\alpha_i, size_i, arr_i)$$

we create a new segment (a memory object of an appropriate size), and relocate all these objects to this segment. First, we allocate a new segment $(\alpha_s, size_s, arr_s)$ with an address pair (α_s, c_s), such that $size_s = \sum_i size_i$. Then we copy the contents of the memory objects $mo_1, ..., mo_n$ into that segment, such that after the copy it holds that:

$$\forall 0 \leq j < size_i. \quad select(arr_s, o_i + j) = select(arr_i, j)$$
$$\text{where} \quad o_i = \sum_{k<i} size_k$$

Finally, we remove from the address space the memory objects $mo_1, ..., mo_n$, and the address constraints on α_i are updated to:

$$\alpha_i = \alpha_s + o_i$$

In the example from Figure 2, the symbolic pointer obtained by reading `a[i]` (at line 8) is resolved to two memory objects: $mo_2 = (\alpha_2, 256, arr_2)$ and $mo_3 = (\alpha_3, 256, arr_3)$. We then create a new segment $mo_4 = (\alpha_4, 512, arr_4)$ with the address pair (α_4, c_4), add mo_4 to the address space, and add the address constraint $\alpha_4 = c_4$. Then we remove from the address space mo_2 and mo_3, and update the address constraints of α_2 and α_3 to:

$$\alpha_2 = \alpha_4, \quad \alpha_3 = \alpha_4 + 256$$

After this transformation, our symbolic pointer is resolved to only one object (mo_4), thus reducing the number of forks. Note that with this approach, the segments that we dynamically create don't contain redundant objects, those that are not pointed by the symbolic pointer. We merge only the objects that were resolved using the solver, thus reducing the size of the created segments.

The above transformation is graphically depicted in Figure 3. The state of the memory is shown before and after the merge transformation. Note that the contents of the object `array` are not affected by this transformation.

2.3.3 Optimizations. The segments created using our approach are guaranteed to be smaller, compared to the segments computed by pointer analyses based partition. However, the resolution process with our approach is more expensive, as a symbolic pointer still may point to multiple objects, before those are merged into one segment. The array theory constraints which are added due to the merging of objects are harder to solve, which makes constraint solving and symbolic pointer resolution more expensive. To address these issues, we propose several optimizations.

Context Based Resolution. Resolving symbolic pointers is a challenging task, as a symbolic pointer may refer to multiple memory objects. To determine these memory objects, symbolic executors (such as KLEE) scan the entire memory, and check for each scanned memory object an appropriate range condition using the solver. The dependence on the solver makes this process expensive, especially when the number of memory objects is high.

We observed that the resolution process can be optimized when using a context abstraction of the allocated objects. When a memory object is allocated, its k-context abstraction is obtained by the calling instructions of the last k stack frames, including the current instruction. Once we learn the contexts of the resolved objects at a given location (instruction), we can use that information to speed up the resolution process at the next time we have a resolution at that location. When objects are scanned during the resolution process, an object whose context is not one of the recorded contexts will be skipped, that is, a query will not be sent to the solver. Once the resolved objects are merged into a new segment (as described in Section 2.3.2), we can check the completeness of the resolution process with a single solver query that checks if the symbolic pointer *must* point to the newly created segment. If that's not the case, we fallback to the default resolution mechanism. Note that applying context-based resolution in the forking model is not beneficial, as checking completeness requires scanning the entire memory.

Reusing Segments. Modern symbolic executors use various heuristics for optimizing constraint solving. One of the key heuristics used in KLEE is query caching [3, 4], which associates the query expression to the result of the query. Consider again the program from Figure 2, and suppose that two symbolic states execute the branch instruction at line 8. With the dynamically segmented memory model, when the first state executes the branch, the resolved memory objects pointed by `a[i][j]` are merged to a new segment. Suppose that the memory object allocated at line 2 is $(\alpha_1, size_1, arr_1)$, and the created segment is $(\alpha_2, 512, arr_2)$ with the address pair (α_2, c_2). In that case, the expression corresponding to the branch condition `a[i][j] == 1` after substituting the address constraints will be:

$$select(arr_2, (select(arr_1, i) + j) - c_2) = 1$$

Similarly, if in the second symbolic state the created segment is $(\alpha_3, 512, arr_3)$, then the expression for the same condition will be:

$$select(arr_3, (select(arr_1, i) + j) - c_3) = 1$$

The query caching is perform *syntactically* on the expression level, therefore the second symbolic state will have a cache miss for this query and will invoke the solver.

To handle this issue we attempt to reuse previously allocated segments. If at a program location L, a symbolic pointer was resolved to memory objects $\{mo_i\}$ that were merged to a segment $(\alpha_s, size_s, arr_s)$ whose address pair is (α_s, c_s), then we record the mapping between the tuples $(L, \{mo_i\})$ and (c_s, arr_s). If later another symbolic state resolves a symbolic pointer at program location L to the same set of memory objects $\{mo_i\}$, the created segment will be $(\alpha'_s, size_s, arr_s)$ with the address pair (α'_s, c_s).

This way, when the second symbolic state performs the merge at line 8, its address pair will be (α_3, c_2), and its SMT array will be arr_2. Therefore, the expression of the branch condition will be equal to that of the first symbolic state, which will result in a cache hit.

2.4 Application: Intra-object Partitioning

In this section, we show how the relocatable addressing model can dynamically transform the memory state such that a single object allocated by the program can be represented by several adjacent

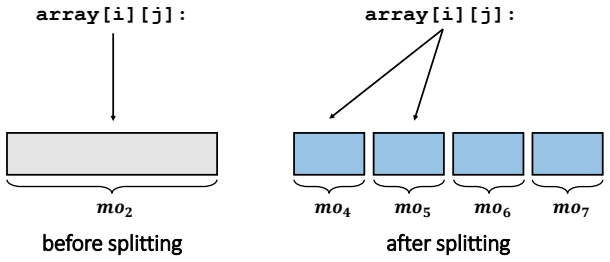

Figure 4: Splitting an object into adjacent smaller objects.

smaller memory objects, thus reducing the size of the SMT arrays and consequently the cost of constraint solving.

When a memory object is accessed with a symbolic offset, the resulting values are translated using array theory to *select* and *store* expressions. Solving array theory constraints is usually much harder than regular bit-vector constraints, especially when the arrays are big. During symbolic execution, we might have many such queries, which then results in a significant slowdown.

To make constraint solving more efficient, we attempt to reduce the size of big SMT arrays by dynamically splitting their corresponding memory objects into smaller ones. We split only memory objects which were accessed with a symbolic offset, as array theory will not be used for memory objects that are accessed only with concrete offsets.

The split operation on a memory object $mo = (\alpha, size, arr)$ with an address pair (α, c) works as follows: We allocate n new memory objects $mo_1, ..., mo_n$ with address pairs (α_i, c_i), such that their addresses are consecutive:

$$\forall 1 \leq i < n.\ c_{i+1} = c_i + size_i$$

and add the address constraints $\{\alpha_i = c_i\}$. We initialize the contents of each memory object $mo_i = (\alpha_i, size_i, arr_i)$ using mo such that:

$$\forall 0 \leq j < size_i. \quad select(arr_i, j) = select(arr, o_i + j)$$

$$\text{where} \quad o_i = \sum_{k<i} size_k$$

Then we remove from the address space the memory object mo, and update the address constraint on α to $\alpha = \alpha_1$, the symbolic address of the first split memory object. The sizes $\{size_i\}$ of the memory objects $\{mo_i\}$ are determined according to a given partitioning strategy.

Consider the symbolic execution of the program from Figure 2 under the forking model. Assuming that the memory object allocated at line 2 is $(\alpha_1, 16, arr_1)$, the expression of the pointer from which the value `a[i][j]` is read is:

$$select(arr_1, i) + j$$

This symbolic pointer is resolved to the two objects allocated at line 4, namely mo_2 and mo_3, and the symbolic state is forked. If we continue the execution with the symbolic state which constrains the symbolic pointer to the memory object $mo_2 = (\alpha_2, 256, arr_2)$, as depicted at the upper part of Figure 4, then the value of `a[i][j]` would be:

$$select(arr_2, select(arr_1, i) + j - \alpha_2)$$

Since this value is read from mo_2 with a symbolic offset, we would like to split mo_2 . Suppose that we choose a partitioning strategy that splits a given memory object into n memory objects of equal size. Then for $n = 4$ we split mo_2 into 4 objects: mo_4, mo_5, mo_6, mo_7, as depicted in the lower part of Figure 4. After the split, we re-execute the load instruction that references the previous symbolic pointer, but now it is resolved to two objects, mo_4 and mo_5, due to the constraint $j < 100$.

Assuming that $mo_4 = (\alpha_4, size_4, arr_4)$, in the first newly forked state the expression of the value `a[i][j]` is now:

$$select(arr_4, select(arr_1, i) + j - \alpha_4)$$

Due to the split we explore an additional path, the one that constrains the symbolic pointer to mo_5, but we get smaller SMT arrays: arr_4 and arr_5. The size of arr_4 is 64, which is four times smaller than the size of arr_2, which makes the new expression much easier to solve.

The effect of the split operation depends on the partitioning strategy: For smaller split objects the SMT arrays are smaller, but the number of resolved objects is higher, and therefore the number of forks is higher as well. For bigger split objects the number of resolved objects and forks is lower, but the resulting SMT arrays are consequently bigger. The number of split objects also affects the resolution process which works by scanning the entire memory. We investigate this trade-off in Section 4.3.

3 IMPLEMENTATION

We implemented our addressing model on top of the KLEE symbolic execution engine [3], configured with LLVM 7.0.0 and STP 2.3.3. We modified KLEE's allocation API to return symbolic addresses instead of concrete ones, and extended the symbolic state with the address constraints. Our addressing model is actually implemented as a mixed *concrete-symbolic* one, that is, we allow allocation of memory objects with concrete addresses as well. Obviously, the applications described in Sections 2.3 and 2.4 can't be applied to such memory objects.

In our implementation, symbolic addresses can be assigned to both stack and heap memory objects. By default, we don't create symbolic addresses for stack variables, as they are rarely involved in multiple resolutions or array theory constraints. From technical reasons we currently don't support global variables with symbolic addresses, but this can be solved by automatically rewriting the program such that global variables would be allocated on the heap upon the program's startup.

When a memory object is split, we need to ensure that reads and writes to primitive fields are performed within a single memory object. We assume that struct fields are aligned to 8 bytes, so the size of each split object must be aligned to 8 bytes as well.

4 EVALUATION

We perform several experiments in our evaluation: In Section 4.1, we show the correctness of our addressing model. In Sections 4.2 and 4.3 respectively, we show the benefits of our addressing model

Table 1: The benchmarks used throughout the evaluation, with their versions and number of source code lines (SLOC).

Benchmark	Version	SLOC
m4	1.4.18	80K
make	4.2	28K
sqlite	3.21	127K
apr	1.6.3	60K
gas	2.31.1	266K
libxml2	2.9.8	197K
coreutils	8.31	188K

when applied in the context of inter-object partitioning (dynamic merging) and intra-object partitioning (dynamic splitting).

Experimental Setup. We performed our experiments on an a machine running Ubuntu 16.04, equipped with an Intel i7-6700 processor (8 cores) and 32GB of RAM.

Benchmarks. The benchmarks used in our evaluation are listed in Table 1. *GNU m4*[4] is a macro processor included in most Unix-based systems. *GNU Make*[5] is a tool which controls the generation of executables and other non-source files, also widely used in Unix-based systems. *SQLite*[6] is one of the most popular SQL database libraries in the world. *Apache Portable Runtime*[7], (APR) is a library used by the Apache HTTP server that provides cross-platform functionality for memory allocation, file operations, containers, and networking. *GNU Assembler*[8], commonly known as *gas*, is the assembler used by the GNU project, and is the default back-end of GCC. The *libxml2*[9] library is a XML parser and toolkit developed for the Gnome project. *GNU Coreutils*[10] is a collection of utilities for file, text, and shell manipulation.

4.1 Correctness

In this experiment, we empirically validate the correctness of our addressing model. To do so, we check that the existing addressing model (vanilla KLEE) and our model are consistent in terms of path exploration. Here we use our addressing model without applying the merging (Section 2.3) and splitting (Section 2.4) operations, therefore the number of explored paths in both models is expected to be identical. For this experiment, we used the programs listed in Table 1, where in *coreutils* we selected 15 programs which behave deterministically across multiple runs.

For each program, we proceed with the following evaluation process: First, we run KLEE for roughly one hour, and record the number of executed instructions. Then, we run KLEE again up to the number of recorded instructions, with both its default addressing model and our addressing model. Finally, we validate that the number of paths explored by both addressing models is identical. To enforce determinism, we ran each program with the DFS search heuristic, using the deterministic memory allocator.

Our experiments confirmed that the number of explored paths with both addressing models is indeed the same. We also measured the runtime overhead induced by our addressing model, which comes from substituting expressions and maintaining additional symbolic values for address expressions, as described in Section 2.2. Figure 5 shows the termination time (in seconds) for each program with the two addressing models. For the programs we tested, the maximum runtime overhead was 16% (in *cat* from *coreutils*), and the average runtime overhead was 4%.

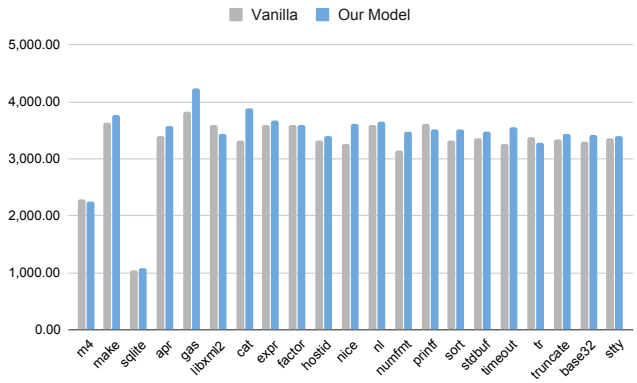

Figure 5: The termination times in *seconds* for each program with the existing addressing model (Vanilla) and our addressing model (Our Model).

4.2 Inter-object Partitioning

4.2.1 Dynamically Segmented Memory Model. In this experiment we compare the performance of our dynamically segmented memory model (DSMM) with the segmented memory model (SMM) proposed in [20], and the forking model (FMM) used in vanilla KLEE. We perform the same experiment as in [20] (Figures 6,7,8,9 from the original paper). The benchmarks of this experiment are: *m4*, *make*, *sqlite*, and *apr*[11]. These programs create symbolic pointers which trigger multiple resolutions, as they all use hash tables with symbolic keys. Programs in which symbolic pointers with multiple resolutions are few (*gas*) or absent (*libxml2*, and *coreutils*) are not used in this experiment. The impact of our addressing model on the runtime overhead for these programs is discussed in Section 4.1.

We run each program with a timeout of 24 hours using three search heuristics (DFS, BFS and KLEE's default search heuristic) with the deterministic memory allocator, and measure the termination time and the memory consumption with all three memory models. Our dynamically segmented memory model is run with several optimizations, whose impact is investigated in Section 4.2.2.

[4] https://www.gnu.org/software/m4/
[5] https://www.gnu.org/software/make/
[6] https://www.sqlite.org/
[7] https://apr.apache.org/
[8] https://sourceware.org/binutils/docs/as/
[9] http://www.xmlsoft.org/
[10] https://www.gnu.org/software/coreutils/

[11] In *sqlite* we disabled the *counter-example caching* query optimization, as it lead to inconsistent termination times. For SMM, the result was timeout with and without this optimization. Note that this optimization is different from the *query caching* discussed in Section 2.3.3.

Table 2: Maximum segment size in bytes.

Program	Max. Size	
	SMM	DSMM
m4	2753	1008
make	7574	1776
sqlite	17604	528
apr	8316	240

Table 3: Termination time in *hh:mm* or *TO* (timeout) and memory usage in *GB* or *OOM* (out-of-memory) with different memory models: FMM**,** SMM **and** DSMM**.**

		Time			Memory		
	Search	FMM	SMM	DSMM	FMM	SMM	DSMM
m4	DFS	21:03	01:04	00:26	0.1	0.2	0.3
	BFS	09:32	01:04	00:30	OOM	0.3	0.4
	Default	09:41	01:10	00:35	OOM	0.5	0.5
make	DFS	06:35	22:49	03:35	1.6	0.8	0.5
	BFS	06:33	23:03	03:34	1.6	0.8	0.6
	Default	07:27	23:04	03:38	1.5	0.8	0.4
sqlite	DFS	00:18	TO	01:36	0.2	0.8	0.4
	BFS	00:18	TO	01:34	0.3	0.7	0.4
	Default	00:18	TO	01:36	0.2	0.7	0.5
apr	DFS	01:01	00:07	00:19	0.1	0.1	0.1
	BFS	00:59	00:07	00:19	0.1	0.1	0.1
	Default	00:57	00:07	00:19	0.1	0.1	0.1

Table 2 shows the maximum segment sizes for both SMM and DSMM. The maximum segment size created using our approach is reduced on average by 83%, where the reduction is 63% in *m4*, 77% in *make*, 97% in *sqlite*, and 97% in *apr*. The sizes of the created segments are crucial for the performance, as the sizes of their corresponding SMT arrays affect the complexity of constraint solving.

Table 3 shows for each benchmark and search heuristic the termination time and the memory consumption obtained with the three memory models. We first discuss the performance comparison between SMM and DSMM, and then discuss the performance of FMM compared to the segmentation-based memory models (SMM and DSMM).

In *m4*, our approach achieves an average speedup of 2.2× compared to the segmented memory model, with a slightly higher memory usage. In *make*, our approach achieves an average speedup of 6.3× compared to the segmented memory model, and the memory usage is roughly the same. In *sqlite*, the segmented memory model doesn't terminate before the 24 hours time limit (for all search heuristics), which results in an average speedup of at least 14.2× for our approach. The memory usage is roughly the same with both approaches in this case. In *apr*, the memory usage is equally low in both approaches, but it terms of termination time, this is the only case where our approach performs worse. As was mentioned above, the maximum segment size with our approach is significantly smaller, but the segmented memory model is still 2.3× faster on overage. The program allocates several small objects using *libc*'s standard allocation API, and some other objects using a custom pool allocator, that internally uses an array of 8192 bytes. During the symbolic execution of the program, the SMT array associated with the big array (of the pool allocator) is involved in the queries, which slows down the exploration. In the case of the segmented memory model, the big array and the other small objects are merged into one segment. With our approach, some of the small objects are dynamically merged into one segment, but the big array remains untouched. In both approaches, we have a big array of roughly the same size which is involved in the constraints, thus slowing down the solver. The advantage of our approach is having smaller segments, but symbolic pointers are still needed to be resolved (possibly to multiple memory objects), which leads to higher resolution time. In the segmented memory model a symbolic pointer is guaranteed to point to one segment at most, so the resolution process is less costly. In this case, the resolution process with our approach is indeed higher, as each resolution query involves the big array that was mentioned before. Actually, the resolution process takes roughly 50% of the total execution time, which explains the extra time required for our approach to terminate.

When comparing the performance of FMM with the segmentation-based memory models (SMM and DSMM), the results are mixed. In *m4* and *apr*, FMM performs significantly slower with all search heuristics than other memory models. The memory usage in *apr* is basically identical across all memory models, but in *m4* the memory usage with FMM reaches the limit (with BFS and Default), which leads to an incomplete exploration and early termination. In *make*, FMM performs faster than SMM but slower than DSMM, and its memory usage is higher compared to other memory models. In *sqlite*, FMM outperforms SMM and DSMM in terms of termination time and memory usage.

4.2.2 Optimizations. Here we further investigate the impact of the optimizations discussed in Section 2.3.3. We use the same benchmarks as in Section 4.2.1, and run each of them with the DFS search heuristic, using the deterministic memory allocator.

Context-Based Resolution. To understand how a given context abstraction affects the process of symbolic pointer resolution, we examine the number of resolution queries (those which are created during a resolution). We evaluate the default resolution mechanism which scans the entire memory, and our context-based resolution with the k-context abstraction which takes into account the last k calling instructions from the stack trace of a given allocation site, including the current instruction.

Table 4 shows the number of resolution queries in different modes: default resolution and context-based resolution with $0 \leq k \leq 4$. The largest reduction occurs when $k = 4$, where the number of queries is decreased by 44% in *m4*, 71% in *make*, 82% in *sqlite*, and 2% in *apr*. We can also see that as k increases, the number of resolution queries (non-strictly) decreases. The impact of k on the reduction rate varies across benchmarks: In *make* and *m4* we have a significant reduction for $k = 0, 1$, but increasing k further does not result in a significant improvement. In *sqlite* a reduction of 25% is obtained already with $k = 0$, and increasing k further

Table 4: The number of resolution queries with different context abstractions.

Program	Default	K-Context				
		0	1	2	3	4
m4	12050	11434	6920	6920	6808	6808
make	8104	2632	2365	2365	2365	2365
sqlite	9134	6816	6816	3320	3320	1668
apr	96	94	94	94	94	94

to $k = 2, 4$ results in a reduction of 64% and 82%, respectively. Compared to other benchmarks, the number of created memory objects in *apr* is relatively small, so the resolution process with the default mechanism is almost optimal. The number of resolution queries is reduced by only two queries for $k = 0$, and using higher values does not give any better results.

Columns None and Opt_1 of Table 5 show the termination time with the default resolution mechanism and with context-based resolution (for $k = 4$). In *m4*, *make*, and *sqlite*, the termination time was reduced by 26%, 13%, and 62% respectively. In *apr*, the number of reduced resolution queries was minor, therefore the termination time was not affected. Note that the reduction in termination time depends not only on the number of reduced queries, but also on the relative proportion of resolution time: If the resolution time is already low, then reducing the number of resolution queries is likely to result in a minor improvement. When the resolution time is high, a significant speedup can be achieved even with a minor reduction in the number of resolution queries.

In general, note that using the highest value for k (or a full-context abstraction with $k = \infty$) is not guaranteed to be beneficial: If the value of k is too high, our context-based resolution might skip too many memory objects, which will result in an *incomplete* resolution.

Reusing Segments. The reusing segments optimization attempts to achieve speedup by improving the solver query caching, that is, reducing the number of solver queries which are actually passed to the SMT solver.

The impact of reusing segments can be seen in columns None and Opt_2 of Table 5. In *m4* and *apr*, the termination time was reduced by 85% and 17% respectively, while in *make* and *sqlite* the reduction was relatively small with 3% and 8% respectively. As was mentioned before, the benchmarks in this experiment use hash tables with buckets, which are typically implemented using an array of pointers. In the case of *m4* and *apr*, the hash tables are initialized at startup, and are not modified after that. Therefore, an SMT array that corresponds to an array of pointers is identical for all the symbolic states, which allows efficient caching when segments are reused. In the case of *make* and *sqlite*, the hash tables are modified after the initialization, so when different states update a hash table by adding a new element, the corresponding SMT arrays are different as well, which makes the reuse mechanism less efficient. To mitigate these issues, one can try to reuse addresses for memory objects in general, not only segments. We leave this direction for future research.

Table 5: Termination time in *hh:mm* in different modes: None: without any optimizations, Opt_1: with context-based resolution (for $k = 4$), Opt_2: with reusing segments, and Both: with both optimizations.

Program	Termination Time			
	None	Opt_1	Opt_2	Both
m4	03:34	02:37	00:33	00:26
make	04:14	03:42	04:06	03:35
sqlite	04:16	01:37	03:58	01:36
apr	00:23	00:23	00:19	00:19

Table 6: Maximum size of split objects.

Program	Size
m4	4072
make	8192
sqlite	328
apr	8192
gas	524411
libxml2	4096

4.3 Intra-object Partitioning

In this experiment, we investigate the impact of splitting arrays on the termination time and the number of explored paths, in programs that create array theory constraints with big arrays. For each program we compare the results obtained by vanilla KLEE and the splitting approach with different partitioning strategies. When the splitting approach is used with a partitioning strategy P_n, a memory object is split into smaller memory objects of size n. We use a *split threshold* of 300 bytes, that is, we split only memory objects whose size is bigger than that given threshold. We use the DFS search heuristic and the deterministic memory allocator.

The benchmarks in this experiment are: *m4*, *make*, *sqlite*, *apr*, *gas*, and *libxml2*. Similarly to the experiments in Section 4.2, we achieve termination by running these programs with a partially symbolic input, except for *libxml2* which is run with a fully symbolic input. In Section 4.2, the programs *m4* and *make* (which were taken from [20]) were run with decreased sizes for some of the arrays: In *m4*, the hash table size was decreased using one of the program's command line flags (`-H`), and in *make* some of the arrays were manually patched to have smaller sizes. In this experiment we restore the default array sizes, in order to test the splitting approach with arrays which are big enough.

Table 7 shows the termination time and the number of paths with both vanilla KLEE and our splitting approach (with different partitioning strategies). In terms of termination time, the speedup of the splitting approach relative to vanilla KLEE varies between 6.0×-13.4× in *m4*, 1.2×-17.9× in *make*, 0.9×-4.0× in *sqlite*, 13.3×-132.1× in *apr*, 33.6×-43.8× in *gas*, and 1.0×-2.5× in *libxml2*. Nevertheless, there were two cases where our approach performed worse: In

sqlite, the size of the split object is 328 bytes, therefore using P_{512} affects neither the memory objects nor the number of explored paths, with the termination time being higher by 4% due to the overhead incurred by our addressing model. Running *libxml2* with P_{32} resulted in a slightly higher termination time, mainly due to the increased number of explored paths.

Table 6 shows the maximum size of a split object for each benchmark. The sizes vary between roughly 500KB in *gas* and only 328 bytes in *sqlite*, which shows that the splitting approach can be successfully applied with both big arrays and relatively small ones.

The partitioning strategy used in the splitting approach directly affects the termination time and the number of explored paths. When an object is split according to some partitioning strategy, a more refined partition will (non-strictly) increase the number of memory objects that a symbolic pointer can point to. Since we are in the forking model, when we decrease n, that is refine the partition, the number of resolved memory objects with P_n increases together with the number of explored paths. In addition, when the partitioning is more refined, the number of memory objects grows, which may result in a slower symbolic pointer resolution. Nevertheless, using a more refined partitioning creates smaller SMT arrays, which makes constraint solving easier. This tradeoff between the complexity of the constraints on one side, and the number of paths and the resolution time on the other, eventually determines the termination time with a given partitioning strategy.

When trying to understand the impact of a given partitioning strategy, we observed two main patterns. When n is decreased in *sqlite* and *apr*, the overhead of forks and resolution remains relatively low and SMT arrays also become smaller, which results in better overall performance. In other benchmarks (*m4*, *make*, *gas*, and *libxml2*), decreasing n toward small values (32) results in a slowdown due to an increased number of explored paths. In *make* and *m4*, increasing n too much toward high values (512) results in a slowdown as well, due to the growing complexity of constraints over bigger SMT arrays. The sweet spot value for n lies somewhere between 64 and 256.

5 RELATED WORK

Coppa et al. [10] model the symbolic memory as a set of tuples, where each tuple associates an address expression to a value expression, along with a timestamp, and a condition. When a write is performed, the memory is updated with a new tuple containing the corresponding address and value expressions. When a read is performed, the memory is scanned to determine the tuples that match the given address expression. The read value is then expressed using an *if-then-else* expression, which is built using the matching tuples. This approach attempts to improve the merging memory model used in ANGR [33], by avoiding concretizations of symbolic pointers when they are encountered in reads or writes. In our work, accessing memory objects with symbolic offsets is handled using array theory. The segmented memory model, in its static [20] and dynamic forms, is similar in spirit to the merging approach, but here instead of using *if-then-else* expressions or disjunctions, we compute some memory partitioning while using array theory. Our splitting approach attempts to improve the constraint solving of array theory constraints which are not used in [10].

Table 7: Termination time in *hh:mm:ss* and number of explored path with vanilla KLEE and different splitting strategies.

Program	Mode	Time	Paths
m4	Vanilla	00:39:46	82
	P_{512}	00:06:40	82
	P_{256}	00:02:58	101
	P_{128}	00:03:16	145
	P_{64}	00:03:50	257
	P_{32}	00:05:08	485
make	Vanilla	09:04:13	3386
	P_{512}	01:13:50	4488
	P_{256}	00:30:26	6088
	P_{128}	00:39:47	10136
	P_{64}	01:53:15	21304
	P_{32}	07:44:19[12]	55928
sqlite	Vanilla	00:18:38	147
	P_{512}	00:19:26	147
	P_{256}	00:15:21	213
	P_{128}	00:09:46	259
	P_{64}	00:07:26	355
	P_{32}	00:04:38	465
apr	Vanilla	01:01:38	961
	P_{512}	00:04:39	1024
	P_{256}	00:02:21	1225
	P_{128}	00:01:16	1444
	P_{64}	00:00:47	1849
	P_{32}	00:00:28	2025
gas	Vanilla	05:18:26	5
	P_{512}	00:07:16	5
	P_{256}	00:07:25	5
	P_{128}	00:07:43	5
	P_{64}	00:08:23	5
	P_{32}	00:09:29	5
libxml2	Vanilla	01:22:27	4413
	P_{512}	00:33:26	5003
	P_{256}	00:33:38	5230
	P_{128}	00:39:13	5821
	P_{64}	00:59:06	6885
	P_{32}	01:23:00	8718

The idea of modeling addresses *not* as constant values was discussed in past work. For example, Hajdu et al. [17] model address values in smart contracts as un-interpreted symbols as in this context addresses can be only queried for equivalence. We allow for arbitrary queries over symbolic addresses.

The idea of using memory partitioning for improving program analysis has been explored before. In the context of bounded model

checking, partitioned models have been used based on various complementary analyses such as points-to analysis [1, 36, 37], data structure analysis (DSA) [21], and type based analysis [2, 9]. CBMC [8] and ESBMC [11] also use points-to analysis to refine their memory models. SeaHorn [16] uses a context-insensitive variant of DSA [21]. The memory partitioning used in prior work is computed ahead of time, while our dynamically segmented memory model doesn't require additional pre-computations, and enables a *path-specific* memory partitioning during runtime, thus resulting in a more accurate partitioning.

Our context-based resolution uses the last k call sites to learn the contexts of resolved objects in order to accelerate the process of symbolic pointer resolution. The usage of call-site abstraction is inspired by context-sensitivity in program analysis [32].

Scaling constraint solving is a well known challenge in symbolic execution [5, 6]. Prior work has used different optimizations such as arithmetic transformations [4, 31], caching query results [3, 4], caching counter examples [3, 38, 39], splitting constraints into independent sets [4], multiple solvers support [26], interval-based solving [13], and even fuzzing-based solving [22]. Perry et al [28] focus on reducing the cost of array theory constraints using several semantics-preserving transformations. These transformations attempt to eliminate array constraints as much as possible by replacing them with constraints over their indices and values. The approaches mentioned above are orthogonal to our splitting approach, with which they could be combined.

6 CONCLUSION AND FUTURE WORK

We presented a novel addressing model where the underlying representations of allocated objects can be dynamically modified, by using *symbolic* addresses rather than concrete ones. We showed how this model can improve the existing segmented memory model, and reduce the cost of solving array theory constraints.

We presented our addressing model, and its two applications, assuming the usage of array theory (as is the case in KLEE). In general, the merging approach discussed in Section 2.3 does not require using array theory, and regardless of the fact the one of the optimizations (2.3.3) is array theory specific, we believe that the core idea can be applied to other symbolic executors as well.

Our work opens the opportunity for different research directions: In our dynamic intra-object partitioning, we used a rather simple partitioning strategy. Automatically determining and customizing the partitioning strategy for a given object in a program might further improve performance. Another challenge is predicting when a splitting or merging transformation is likely to payoff. In addition, these approaches could be applied simultaneously.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the Lev Blavatnik and the Blavatnik Family foundation, Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, Pazy Foundation, and Israel Science Foundation (ISF) grants No. 1996/18.

[12] In this configuration, we use the standard (libc) memory allocator and not the deterministic one used in all other experiments as the latter does not reuse addresses and ran out of space during the experiment.

REFERENCES

[1] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Technical Report.
[2] Rodney M Burstall. 1972. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence* 7, 23-50 (1972), 3.
[3] 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)* (San Diego, CA, USA).
[4] 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)* (Alexandria, VA, USA).
[5] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1066–1071.
[6] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795
[7] Vitaly Chipounov and George Candea. 2010. Reverse engineering of binary device drivers with RevNIC. In *Proc. of the 5th European Conference on Computer Systems (EuroSys'10)* (Paris, France).
[8] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)* (Barcelona, Spain).
[9] Jeremy Condit, Brian Hackett, Shuvendu K Lahiri, and Shaz Qadeer. 2009. Unifying type checking and property checking for low-level code. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 302–314.
[10] Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Rethinking pointer reasoning in symbolic execution. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 613–618.
[11] L. Cordeiro, B. Fischer, and J. Marques-Silva. 2012. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transactions on Software Engineering (TSE)* 38, 4 (July 2012), 957–974.
[12] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.
[13] Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. 2018. PARTI: a multi-interval theory solver for symbolic execution.
[14] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Proc. of the 19th International Conference on Computer-Aided Verification (CAV'07)* (Berlin, Germany).
[15] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)* (San Diego, CA, USA).
[16] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*. Springer, 343–361.
[17] Ákos Hajdu and Dejan Jovanović. 2019. solc-verify: A modular verifier for Solidity smart contracts. *arXiv preprint arXiv:1907.04262* (2019).
[18] 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)* (Snowbird, UT, USA).
[19] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)* (Zurich, Switzerland).
[20] Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, 774–784.
[21] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'07)* (San Diego, CA, USA).
[22] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. 2019. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 521–532.
[23] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)* (Saint Petersburg, Russia).
[24] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. ACM, 691–701.

[25] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)* (San Francisco, CA, USA).

[26] Hristina Palikareva and Cristian Cadar. 2013. Multi-solver Support in Symbolic Execution. In *Proc. of the 25th International Conference on Computer-Aided Verification (CAV'13)* (Saint Petersburg, Russia).

[27] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. (Sept. 2013).

[28] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating Array Constraints in Symbolic Execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'17)* (Santa Barbara, CA, USA).

[29] David A. Ramos and Dawson Engler. 2015. Under-constrained Symbolic Execution: Correctness Checking for Real Code. In *Proc. of the 24th USENIX Security Symposium (USENIX Security'15)* (Washington, D.C., USA).

[30] 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.

[31] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)* (Lisbon, Portugal).

[32] M. Sharir and A. Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D. Jones (Eds.). Prentice-Hall, Englewood Cliffs, NJ, Chapter 7.

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'16)* (San Jose, CA, USA).

[34] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. https://doi.org/10.1561/2500000014

[35] 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.

[36] Bjarne Steensgaard. 1996. Points-to analysis by type inference of programs with structures and unions. In *International Conference on Compiler Construction*. Springer, 136–150.

[37] B. Steensgaard. 1996. Points-to analysis in almost-linear time. In *Principles of Programming Languages (POPL)*.

[38] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12)* (Cary, NC, USA).

[39] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'12)* (Minneapolis, MN, USA).