# A Segmented Memory Model for Symbolic Execution

Timotej Kapus
Imperial College London
United Kingdom

Cristian Cadar
Imperial College London
United Kingdom

## ABSTRACT

Symbolic execution is an effective technique for exploring paths in a program and reasoning about all possible values on those paths. However, the technique still struggles with code that uses complex heap data structures, in which a pointer is allowed to refer to more than one memory object. In such cases, symbolic execution typically forks execution into multiple states, one for each object to which the pointer could refer.

In this paper, we propose a technique that avoids this expensive forking by using a *segmented memory model*. In this model, memory is split into segments, so that each symbolic pointer refers to objects in a single segment. The size of the segments are bound by a threshold, in order to avoid expensive constraints. This results in a memory model where forking due to symbolic pointer dereferences is significantly reduced, often completely.

We evaluate our segmented memory model on a mix of whole program benchmarks (such as *m4* and *make*) and library benchmarks (such as *SQLite*), and observe significant decreases in execution time and memory usage.

## CCS CONCEPTS

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

## KEYWORDS

Symbolic execution, memory models, pointer alias analysis, KLEE

**ACM Reference Format:**
Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338936

## 1 INTRODUCTION

Symbolic execution has seen significant uptake recently in both research and industry, with applications across many different areas, such as test generation [17], bug finding [7], debugging [20], program repair [27] and vulnerability analysis [9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia*
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338936

In symbolic execution, the program under analysis is run on a *symbolic* input, i.e. an input which is initially allowed to have any value. As the program runs, the operations on symbolic values are evaluated as symbolic expressions. When the program branches on one of these symbolic expressions, symbolic execution explores both branches, if they are feasible, adding corresponding symbolic constraints to the respective path conditions. Upon path termination, symbolic execution can ask the solver for a solution to the current path conditions, thus obtaining a concrete test input that will follow the same path when executed concretely.

One of the main strengths of symbolic execution is that it can reason precisely about *all* possible values of each (control-flow) execution path it explores through the code under testing. This amplifies its ability of finding bugs and other corner cases in the code being analysed. However, symbolic execution still struggles to reason in this way about complex data structures that involve pointers that can refer to multiple memory objects at once.

To illustrate this problem, consider the C code in Figure 1, where we set $N$ to 40. When `SINGLE_OBJ` is defined, the program allocates on line 3 a 2D matrix as a single object, for which all elements are initially zero. It then writes value 120 into `matrix[0][0]` on line 10. On lines 12 and 13, it declares two symbolic indexes `i,j` which are constrained to be in the range `[0,N)`. Finally, on line 15, it checks whether the element `matrix[i][j]` is positive.

The program has two paths: one on which `matrix[i][j]` is positive (which happens only when `i` and `j` are both zero), and the other in which it is not (which happens when either `i` or `j` are not zero). Unsurprisingly, a symbolic execution tool like KLEE [5] explores the two paths in a fraction of a second.

Now consider the program in the same figure, but without defining `SINGLE_OBJ`. Now the matrix is allocated as multiple objects, one for each row in the matrix. In this case, KLEE will explore $N + 1$ paths, one for each row in the matrix, and an additional one for the first row in which both sides of the `if` statement are feasible. It will take KLEE significantly longer—12s instead of 0.3s in our experiments—to explore all 41 paths.

The reason is that KLEE, as well as other symbolic executors, use the concrete addresses of memory objects to determine to which objects a symbolic pointer could refer to. In our example, `matrix[i]` is a symbolic pointer formed by adding a concrete base address (the address of `matrix`) with a symbolic offset `i`. When this pointer gets dereferenced on line 15, KLEE needs to know to which memory object it points. Therefore, KLEE scans all the memory objects and asks the solver if the symbolic pointer can be within its bounds. If that is true for more than one object, we are in what is called a *multiple resolution* case. This is the case in our example, where the pointer `matrix[i]` can point to $N$ different objects, namely those corresponding to the $N$ rows of the matrix. (Note that when

```
1  int main() {
2  #ifdef SINGLE_OBJ
3    int matrix[N][N] = { 0 };
4  #else
5    int **matrix = malloc(N * sizeof(int*));
6    for (int i = 0; i < N; i++)
7      matrix[i] = calloc(N, sizeof(int));
8  #endif
9
10   matrix[0][0] = 120;
11
12   int i = klee_range(0, N, "i");
13   int j = klee_range(0, N, "j");
14
15   if (matrix[i][j] > 0)
16     printf("Found␣positive␣element\n");
17 }
```

**Figure 1: 2D matrix allocated as a single object when SINGLE_OBJ is defined, and as multiple objects when it is not.**

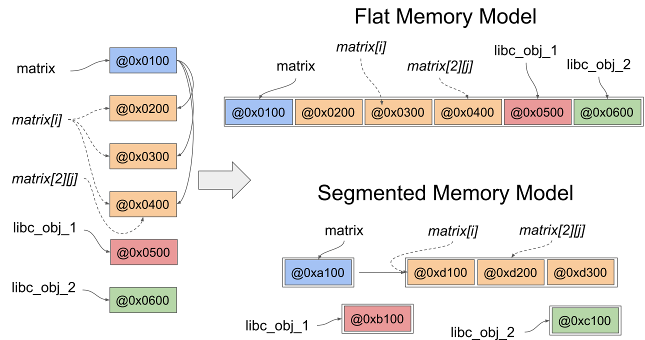



**Figure 2: A concrete memory layout for the program in Figure 1 when SINGLE_OBJ is not defined, illustrating the flat and segmented memory models.**

SINGLE_OBJ is defined this is not the case, as the compiler allocates a 2D array as a 1D array indexed by N*i+j.) So KLEE forks one path for each of the $N$ objects, constraining the pointer to refer to a single memory object in each case. For the case where matrix[i] refers to the first row of the matrix, KLEE explores two paths, one in which matrix[0][j] is greater than zero, and one where it is not. For the cases in which matrix[i] refers to one of the other rows, only the case where matrix[i][j] is not greater than zero is feasible. This gives us a total of N+1 paths.

This *forking memory model* for symbolic pointers has several important drawbacks. First, it leads to path explosion, as each time the code dereferences a symbolic pointer, the execution is forked into as many paths as there are objects to which the symbolic pointer could refer. Second, and related to the first point, it leads to memory exhaustion, as each additional path requires a certain amount of space in memory. Third, it disables the ability of symbolic execution to reason about all possible values on a certain control-flow program path.

In this paper, we propose a novel approach that uses pointer alias analysis to group memory objects that may alias each other into a *memory segment*. This avoids forking, because conservative alias analysis guarantees that a symbolic pointer can only point to objects within a single memory segment.

We describe our technique in detail in Section 2, present its implementation in Section 3, and evaluate it in Section 4. We then discuss its trade-offs in Section 5, present related work in Section 6 and conclude in Section 7.

## 2 PROPOSED MEMORY MODEL

Consider the program in Figure 1 (with SINGLE_OBJ not defined) for $N = 3$, with a memory layout shown on the left-hand side of Figure 2. matrix is an array of pointers allocated at $0x0100$ that points to the rows of the matrix allocated at addresses $0x0200$, $0x0300$ and $0x0400$. Suppose that libc also allocates two memory objects in the background, at addresses $0x0500$ and $0x0600$. There are also two symbolic pointers: matrix[i], which is constrained to point to one of the three matrix rows, and matrix[2][j], which is constrained to point to one of the entries of the last row.

Most symbolic executors reflect this memory layout in their internal memory model. They record the start and end addresses of each memory block and associate a corresponding array in the constraint solver. Therefore, they can handle symbolic pointers such as matrix[2][j] well, because their value resolves to a single memory object. The symbolic address then gets translated into a symbolic index into the solver array that is backing the memory object. Solvers can then solve the associated queries with the popular theory of arrays [15].

The problem arises when a symbolic pointer can point to multiple memory objects, as is the case for pointer matrix[i]. This pointer resolves to three different memory objects (the matrix rows), backed by three different solver arrays. The theory of arrays cannot be used here, because it cannot express such constraints.

### 2.1 Existing Memory Models

There are several memory models for handling multiple resolution that have been considered in the context of symbolic execution: single-object, forking, flat memory and merging. Each memory model aims to resolve a dereference of a symbolic pointer to an access into a solver array.

**Single-object model**. In this approach, the pointer is resolved to one possible object and the other possibilities are discarded. This is the model used by most concolic executors, such as CREST [11], where the single object considered is the one given by the concrete input used in the current round of concolic execution. This is also the model used by the dynamic symbolic executor EXE [7], where the pointer is concretised to one possible value. FuzzBall [26] employs a more general version of this model, in which the pointer is concretized to multiple values that are selected randomly.

This approach scales well, but is incomplete and may miss important execution paths. For instance, both CREST and FuzzBall usually finish running the multi-object version of the code in Figure 1 (when slightly adapted to use the CREST/FuzzBall APIs) without exploring the feasible path where "Found positive element" is printed, incorrectly giving the impression to the user that the statement is not reachable.

**Algorithm 1** Forking model

```
1: function ForkingDereference(Pointer p)
2:     foreach MemoryObject o do
3:         ForkPath( )
4:         AddConstraint(o.start ≤ p < o.end)
5:         solverArray ← GetSolverArray(o)
6:         return solverArray[p − o.start]
7:     end foreach
8: end function
```

**Forking model.** In this approach, shown in Algorithm 1, execution is forked for each memory object to which the pointer can refer (line 3), and on each forked path the pointer is constrained to refer to that object only (line 4). AddConstraint terminates the path if the added constraint is provably false (in our case, if $p$ cannot refer to that object on that path). Finally, the corresponding access into the solver array associated with that object is returned (line 6). As a detail, ForkPath uses the initial path for the last object (instead of forking). Also, for ease of exposition, in this algorithm and the following ones, we ignore out-of-bounds checking and assume dereferences of `char*` pointers.

Consider running Algorithm 1 on our running example from Figure 1 and the layout from Figure 2, with $p$ being `matrix[i]`. The loop will iterate six times, forking for each of the six memory objects in the program. In the first iteration, the path will be immediately killed by AddConstraint, since $p$ cannot point to object at $0x0100$. In the second iteration, $p$ will be constrained to point to the first object it can point to ($0x0200$), and then the corresponding access into the solver array associated with that object is returned, i.e. $solverArray_{0x0200}[p-0x0200]$. The remaining iterations are similar, resulting in three paths, one for each of the three objects at $0x0200$, $0x0300$ and $0x0400$ that $p$ can point to.

Figure 3 shows the behaviour of the forking model compared to other solutions. In particular, it shows the behaviour of KLEE and Symbolic PathFinder (SPF) [28], both of which implement the forking model. Figure 3a shows results for the code in Figure 1 (with `SINGLE_OBJ` not defined), while Figure 3b shows results for a variant of the code that performs two symbolic lookups into the matrix. These figures show that the forking model scales poorly. This is particularly clear when two symbolic accesses are performed, where the execution time increases exponentially with the dimension of the matrix. Note that SPF appears to have an efficient implementation of forking for a small number of paths, which explains its good performance on the single lookup example.

**Merging model.** This approach keeps a single path, and creates an *or* expression, with one disjunct for each possible object to which the pointer could refer.

For space reasons, we omit the formal algorithm and illustrate how it works on our running example. Suppose that the symbolic executor encounters the expression $*p == 1$, with $p$ being again $matrix[i]$. This will be translated into a disjunction, with one disjunct for each of the three memory objects to which $p$ can refer. Each disjunct will express the fact that $p$ points to that object and will replace the pointer with the associated solver array for that object. That is, the expression $*p == 1$ will be translated to the following disjunction, where $sa$ stands for $solverArray$:

**Algorithm 2** Flat memory model

```
1: function NaiveFlatMemoryDereference(Pointer p)
2:     return memorySolverArray[p]
3: end function
```

$$
\begin{aligned}
&(0x0200 \le p < 0x020C \wedge sa_{0x0200}[p - 0x0200] = 1) \\
\vee\ &(0x0300 \le p < 0x030C \wedge sa_{0x0300}[p - 0x0300] = 1) \\
\vee\ &(0x0400 \le p < 0x040C \wedge sa_{0x0400}[p - 0x0400] = 1)
\end{aligned}
$$

Approaches following this idea at a high level were proposed in the context of SAGE [13] and Angr [14].

As illustrated in Figures 3a and 3b, the scalability of this approach on the running example is similar or slightly better than that of forking. In practice (§4), its performance is benchmark dependent.

**Flat memory model.** This approach tackles the multiple resolution problem by treating the whole memory as a single object, backed by a single solver array. Therefore, a solver can reason about symbolic dereferences using the theory of arrays, because all queries are using a single flat array.

The algorithm for translating a dereference is straightforward and shown in Algorithm 2. Any pointer in the program is an offset in the large solver array *memorySolverArray*. This approach is also illustrated in the top right of Figure 2.

We are not aware of any symbolic execution engine that implements this approach. However, we explored this approach in the past and found it not to scale in the context of the symbolic executor EXE [7], the constraint solver STP [15] and a 32-bit address space: "a straightforward remedy to this problem [of a symbolic pointer referring to multiple objects] would be to model memory as a single STP array indexed by 32-bit bitvectors, but this approach is currently too slow to be practical" [8].

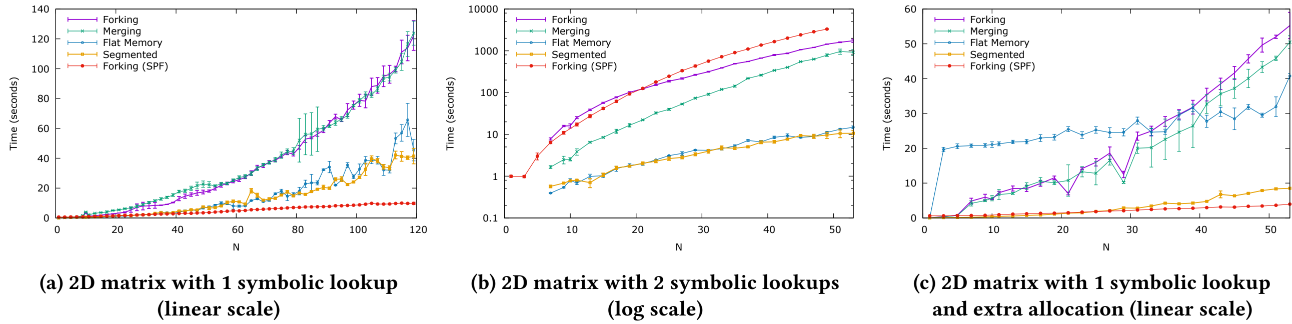
As shown in Figures 3a and 3b, this approach achieves good scalability on our running example and the variant with two symbolic lookups. However, this is because these examples allocate little memory. In an additional experiment, we modify the example to perform an additional irrelevant allocation of 30KB, which is meant to simulate a real application in which most of the memory is not involved in any single dereference. The results in Figure 3c show that the flat memory model also scales poorly.

## 2.2 Segmented Memory Model

The key idea behind our approach is the realisation that memory can be divided into multiple segments, such that any symbolic pointer can only refer to memory objects in a single memory segment. We can then associate one solver array per memory segment, and translate the symbolic pointer to an offset into its associated memory segment. As long as the individual memory segments are small enough, the approach can scale while still handling the problem of symbolic pointers referring to multiple objects.

On our running example, Figure 3 shows this approach has the advantages of the flat memory model in the first two cases, while maintaining its good performance when irrelevant memory allocations are performed.

**Computing memory segments.** To divide memory objects into segments, we use a conservative points-to analysis [1, 18]. The




**(a) 2D matrix with 1 symbolic lookup (linear scale)**

**(b) 2D matrix with 2 symbolic lookups (log scale)**

**(c) 2D matrix with 1 symbolic lookup and extra allocation (linear scale)**

**Figure 3: Runtime of different memory models on a family of 2D matrix benchmarks based on Figure 1 with `SINGLE_OBJ` undefined. All memory models are implemented in KLEE, except for the one explicitely mentioning Symbolic PathFinder.**

**Algorithm 3** Computing and using memory segments

1: **function** ComputeMemorySegments
2: PerformPointsToAnalysis( )
3: $ptSets \leftarrow \emptyset$
4: **foreach** pointer $p$ **do**
5: $ptSets \leftarrow ptSets \cup$ PointsToSet($p$)
6: **end foreach**
7: **while** $ptSets$ changes **do**
8: **foreach** $s \in ptSets$ **do**
9: **if** $\exists s' \in ptSets \,.\, s' \neq s \wedge s' \cap s \neq \emptyset$ **then**
10: $ptSets \leftarrow (ptSets \setminus \{s, s'\}) \cup \{s \cup s'\}$
11: **end if**
12: **end foreach**
13: **end while**
14: **foreach** $s \in ptSets$ **do**
15: $seg \leftarrow$ new MemorySegment( )
16: AssocPTSet($seg$) $\leftarrow s$
17: **end foreach**
18: **end function**
19:
20: **function** GetMemorySegment(Pointer $p$)
21: $pts \leftarrow$ PointsToSet($p$)
22: **foreach** MemorySegment $seg$ **do**
23: **if** $pts \subseteq$ AssocPTSet($seg$) **then**
24: **return** $seg$
25: **end if**
26: **end foreach**
27: **end function**
28:
29: **function** HandleAlloc(Pointer $p$, Size $sz$)
30: $seg \leftarrow$ GetMemorySegment($p$)
31: **return** AllocateIn($seg$, $sz$)
32: **end function**

result of a points-to analysis is a mapping between pointers and points-to sets. The *points-to set* of a pointer $p$ is a set of pointers and abstract memory objects to which $p$ may refer to during execution. An *abstract memory object* is identified by the static allocation point in the program. For instance, in Figure 1, there are two abstract memory objects, $AO_1$, corresponding to the allocation at line 5, and $AO_2$, corresponding to the allocation at line 7.

The function ComputeMemorySegments in Algorithm 3 shows the algorithm for computing memory segments. After running the points-to analysis (line 2), the set *ptSets* is initialised to contain all the points-to sets computed by the analysis (lines 3–6). Then, any two points-to sets $s$ and $s'$ that overlap are merged until no such sets exist anymore (lines 7–13). Finally, for each of the resulting points-to sets a new memory segment is created (lines 14–15) in which all the objects associated with that points-to set will be allocated. The map AssocPTSets remembers the points-to set associated with each memory segment (line 16).

In our example from Figures 1 and 2, there are two pointers we consider: `matrix` and `matrix[i]`. Pointer analysis tells us the points-to set of `matrix` is a singleton set containing the abstract object $AO_1$. Similarly, the points-to set of `matrix[i]` is a singleton set containing $AO_2$. Continuing with Algorithm 3, we therefore initialise $ptSets = \{\{AO_1\}, \{AO_2\}\}$ (lines 3-6). In this case there are no common elements in those sets, so lines 7-13 have no effect. Finally, we create two memory segments corresponding to the two elements in *ptSets*. $AO_1$ corresponds to the memory segment starting at $0xa100$ in Figure 2, where the array of pointers to the matrix rows will be allocated. $AO_2$ corresponds to the memory segment starting at $0xd100$, where the actual rows of the matrix will be allocated.

The function GetMemorySegment in Algorithm 3 returns the memory segment associated with a pointer $p$ in the program. It does so by finding the memory segment whose associated points-to set contains the points-to set of $p$. (We note that an implementation could be optimised by keeping a reverse mapping from points-to sets to memory segments.)

GetMemorySegment is then used by HandleAlloc in Algorithm 3 to handle a heap allocation of the form `p = malloc(sz)` in C. The function allocates memory for the target pointer $p$ in its associated memory segment. In our example, it returns the memory segment starting at $0xa100$ when encountering the allocation for `matrix` on line 5. For the allocation on line 7 where row allocations are made, it returns the memory segment starting at $0xd100$.

Finally, Algorithm 4 shows how a dereference of a pointer $p$ is handled. We first obtain the memory segment associated with $p$ (line 2), then we get the solver array associated with that segment (line 3), and return the corresponding solver array access (line 4).

**Restricting segment size.** The approach of Algorithm 4 scales as long as the size of each memory segment remains small. In practice, some segments may become very large, imposing significant

**Algorithm 4** Pure segmented memory model

```
1: function PureSegmentedDereference(Pointer p)
2:     seg ← GetMemorySegment(p)
3:     solverArray ← GetSolverArray(seg)
4:     return solverArray[p − seg.start]
5: end function
```

**Algorithm 5** Segmented memory model

```
1: function SegmentedMemoryDereference(Pointer p)
2:     foreach MemorySegment seg do
3:         ForkPath( )
4:         AddConstraint(seg.start ≤ p < seg.end)
5:         solverArray ← GetSolverArray(seg)
6:         return solverArray[p − seg.start]
7:     end foreach
8: end function
9:
10: function AllocateIn(MemorySegment seg, Size sz)
11:     if seg.size ≤ Threshold then
12:         return Extend(seg, sz)
13:     else
14:         seg′ ← ChooseSegment( )
15:         return AllocateIn(seg′, sz)
16:     end if
17: end function
```

overheads on the solver. To ensure segment sizes remain small, we refine the technique by imposing a threshold on the maximum segment size. If the program tries to allocate memory in a certain segment and the current size of that segment already exceeds the threshold, a new segment is used. When this happens, some pointers may now refer to several segments. In that case, dereferencing a pointer forks execution for each memory segment. Essentially, the model becomes a hybrid between the pure segmented memory model of Algorithm 4 and the forking model of Algorithm 1.

Our finalised approach is illustrated in Algorithm 5, (together with the previously discussed Algorithm 3). The dereference function, SegmentedMemoryDereference, is similar to that of the forking model, only that now we iterate over all segments, trying to find those to which $p$ could refer. Remember that the AddConstraint function kills a path if the added constraint is *false*, that is when the pointer cannot refer to that segment.

The other change is required in the allocation function. Instead of simply allocating the required memory in the given segment, the modified *AllocateIn* function works as follows. If the current size of the segment is under the threshold, the allocation is performed in that segment (lines 11–12). Otherwise, we choose another segment (which could either be a new segment or an existing one) and we try to allocate in there (14–15).

Another consequence of this hybrid model is that it is resilient to bugs in pointer alias analysis. Should it incorrectly report two pointers not aliasing each other, putting them in different memory segments, the hybrid approach would simply fork for the two cases without losing any precision. Conversely, this kind of bug would cause the pure segmented memory model to miss paths.

In practice, the performance of the algorithm depends on the precision of the points-to analysis. If the analysis is extremely imprecise, it may put all pointers into a single points-to set, and our model would degenerate to the flat memory model. At the other extreme, if the threshold is too small (or the points-to analysis extremely buggy), our model would degenerate to the forking model.

## 3 IMPLEMENTATION

We implemented our approach on top of KLEE commit 0283175f, configured to use LLVM 7 and the STP constraint solver 2.1.2. Below we provide additional details about our implementation.

**Points-to analysis.** We use SVF [34], a scalable LLVM-based framework for value-flow construction and pointer analysis. In particular, we use whole-program wave propagation-based Andersen analysis [29], a fast flow-insensitive, context-insensitive inclusion-based points-to analysis. We found it to be significantly faster than the basic Andersen analysis [1] while maintaining precision. It terminated in under two minutes for all our benchmarks. Compared to the high runtime of symbolic execution, we found the analysis time to be insignificant.

**Memory segment implementation.** For each memory segment that needs to be created in Algorithm 3 (line 15) we reserve 200KB of address space using `mmap`. At this point the segment has size 0, but can hold an object of maximum size 200KB. We use `mmap` instead of `malloc` to save memory, as in practice many memory segments will be empty and will therefore not be mapped to physical memory.

The threshold size for the hybrid approach was set to 10KB, which was small enough for STP to be reasonably performant, while still being large enough to significantly limit forking.

We write the size of each allocation just before the returned pointer to enable `realloc` and `free`. The `realloc` function is handled with an LLVM pass that runs before the execution and replaces all calls to `realloc` with `malloc`, `memcpy` and `free`. We use the previously stored allocation size information to determine the size of the region to copy with `memcpy`.

The memory freed by calls to `free` is added to a list of free space in each memory segment. That list is then scanned before a memory segment is extended for allocation. We try to find the smallest gap that fits the requested size, and if such a gap is found, we perform the allocation there. We found this to be an important optimisation to keep the sizes of memory segments small.

**Constants and local variables.** A common performance optimisation in pointer alias analysis is to model all constant objects (such as constant strings) as single memory objects [19]. We similarly adopt a scheme in which constant objects are disregarded by the alias analysis and allocated in their own memory segments. We make the same choice for local memory objects, namely objects allocated on the stack. We found only one program where constant objects are involved in multiple resolution, but this was a case where the forking model would work well too, as the pointer could only refer to two different objects. We found no programs where local objects were involved in multiple resolution. Of course, the approach presented here is easily adapted to support these kind of objects, should a need for it arise.

**Flat memory model.** We found it easy to implement the flat memory model in KLEE, by making all allocations into a single memory segment backed by a single solver array.

**Merging model.** The merging model is implemented by leveraging KLEE's ability to merge states. Upon hitting a multiple resolution, we let KLEE fork as usual and then immediately merge all the states thus creating the *or* expression of their path conditions. We note that this approach might not be optimal—however, KLEE spent most of the time in constraint solving activities in the merging runs in our evaluation. Therefore, we believe the threat to validity against this implementation choice is minimal.

# 4 EVALUATION

Symbolic pointer dereferences that trigger multiple resolutions often make existing symbolic execution tools unusable. However, only some types of programs trigger such dereferences, and we discovered that benchmark suites used in the past to evaluate symbolic execution, such as *GNU Coreutils* [16], do not trigger many multiple resolutions.

In general, benchmarks which are prone to trigger multiple resolutions are those where a symbolic input (e.g. coming from files, command-line arguments or environment variables) flows into a data structure indexed by that input. Hash tables are the prime example of such benchmarks, widely used in a variety of programs.

Therefore, we selected several large programs that use hash tables—*m4*, *make*, *APR* and *SQLite*—and used test harnesses that drive execution toward the parts of the code employing these hash tables. Essentially, our evaluation is meant to show that our model can make a big difference in such cases, while acknowledging that it is not relevant all the time. We further discuss this aspect in §5.

Ideally, we would like to compare the fraction of the search space explored in these programs under each memory model. Unfortunately, this is challenging to do, especially since the number of explored paths changes across models due to forking and merging. Therefore, we decided to build our test harnesses in such a way that programs terminate. Then, we can simply compare the time needed to finish the exploration under different memory models. For one program where we didn't manage to write such a driver, *make*, we measure how quickly each memory model reaches coverage saturation. As a sanity check, we made sure that terminating benchmarks reach the same coverage at the end under all memory models, and that the flat memory model and the two variants of the segmented memory model explore the same number of paths (recall that we didn't see any forking in these benchmarks with the hybrid segmented memory model).

We also measure the memory consumption of KLEE under each memory model as a proxy for how many states KLEE has to keep in memory at each point and to illustrate a further tangible benefit of choosing the right memory model.

We observed that execution time and memory consumption can vary significantly with different search strategies. Therefore, we conducted our experiments using three different strategies: DFS and BFS representing extreme behaviors in terms of memory consumption, and KLEE's default search heuristic representing a good general strategy. KLEE's default heuristic interleaves random path selection with a heuristic that aims to maximize coverage [5].

**Table 1: Impact of points-to analysis on our benchmarks and its runtime.**

| Benchmark | Total mem. (bytes) | Max. segment size (bytes) | | Analysis runtime (s) |
|---|---|---|---|---|
| | | ideal | SVF | |
| *APR* | 24,904 | 120 | 16,588 | 1.9 |
| *GNU m4* | 3,392 | 1,051 | 2,753 | 4.3 |
| *GNU make* | 17,663 | 664 | 7,936 | 4.7 |
| *SQLite* | 45,461 | 506 | 17,604 | 70.3 |

```
1 define(`A', `l')
2 define(`P', 2)
3 ?
4 ?
5 ifelse(?, `l', ifelse(?, P, eval(1 + n)) ,`failed')
```

**Figure 4: Symbolic input to *m4*, where ? denotes a symbolic character.**

The experiments were run on an Intel(R) Core(TM) i7-6700 at 3.40GHz with 16GB of memory. We use a timeout of two hours in our experiments and a memory limit of 4GB.

## 4.1 Impact of Points-to Analysis

The performance of the segmented memory model is highly dependent on the size of the segments, which is directly related to the precision of the points-to analysis. In order to understand by how much our results would be improved by a more precise points-to analysis, we decided to approximate a best-case points-to analysis.

To do so, we run the forking approach on our benchmarks until it hits the first multiple resolution, where we record the allocation sites with allocation context of all the objects involved in the multiple resolution and terminate the execution. We then restart the execution, placing all objects allocated at those allocation sites in the appropriate contexts into a segment. Should a new multiple resolution be encountered, we again record the allocation site and its context, create a new segment and merge any segments if needed, and repeat this process until no more multiple resolutions are encountered (within a certain timeout). In practice, we only needed fewer than five iterations for our benchmarks. Essentially, we group into segments only the objects actually involved in multiple resolutions in a given execution. This gives us results which are at least as good as the most precise points-to analysis possible.

Table 1 shows the impact of the pointer alias analysis on the benchmarks we use in our evaluation (and which will be described in detail in §4.2 to §4.5). The first column shows the total memory used by the dynamically-allocated memory objects in each program. This is the size of the flat memory segment.

The next two columns show the maximum segment sizes for both the SVF pointer analysis and our idealized analysis. For all benchmarks, the points-to analysis significantly reduces the largest segment size. The maximum segment size of the ideal analysis is significantly smaller than for the SVF analysis, suggesting there are important opportunities for improving the precision of SVF.

Using the SVF analysis, the *APR* and *SQLite* benchmarks reach the maximum threshold of 10KB for our segments and get split. Despite this split, we did not observe any forking in the segmented memory model for *APR*. For *SQLite*, we observed 2 forks into 2 paths each.

The last column of Table 1 shows the runtime of the SVF points-to analysis. For all benchmarks but *SQLite*, the analysis took less than 5 seconds, which is insignificant compared to the cost of symbolic execution. Even for complex programs like *SQLite*, the analysis took 70s, which is noticeable, but still little in a symbolic execution context.

## 4.2 *GNU m4*

GNU *m4* [24] is a popular implementation of the *m4* macro processor included in most UNIX-based operating systems. It is a relatively small program consisting of about 8000 lines of code, which makes extensive use of hash tables to look up strings. In a nutshell, *m4* operates by reading in the input text as a sequence of tokens. It then looks up each token in a hash table. If the token is found, it replaces it with the value in the hash table. Otherwise, it outputs the token and continues with the next one. That makes it a good candidate to benefit from the proposed memory model, as a significant part of *m4*'s computation consists of hash table lookups which trigger forking due to multiple resolution.

To run *m4* using symbolic execution, we need to make its input symbolic. To reach deeper parts of the code, where *m4* operates as described in the paragraph above, we run *m4* on a partially symbolic input. The outline of the input we used is shown in Figure 4. We define several concrete one-character macros, using the `define` keyword. Then we expand two macros, with the name of the macro being symbolic, which results in a symbolic lookup in the hash table. Finally, we expand two more macros inside `ifelse` constructs. The example is set in a way that illustrates the multiple dereference problem, but we believe it is quite representative of how *m4* is used.

We ran KLEE on *m4* for two hours under DFS, BFS and KLEE's default strategy, with forking, merging, flat and segmented memory models. The segmented memory model is further broken down into two runs. The first one is using points-to information computed from SVF. The second one uses ideal points-to information, which was obtained with a pre-run as described in §4.1.

Figure 6 shows, for each search strategy, KLEE's termination time and memory usage for each of the five memory models. The forking and flat memory models do not terminate before the 2-hour timeout. The segmented memory model using SVF terminates in around one hour, with the ideal version a few minutes earlier. In contrast to the segmented memory model, which is only mildly influenced by the search strategy, the performance of the merging model heavily depends on the search strategy: for DFS, it terminates in only 25 minutes, for BFS in 88 minutes, while with KLEE's default strategy it times out after 2 hours.

Under DFS, the memory consumption is unsurprisingly low, as only a small number of states are kept in memory at one time (note that the y-axis has a different scale for DFS compared to BFS and the default KLEE strategy). The SVF segmented memory has slightly higher overall memory usage due to the fixed memory cost of keeping all points-to information. For BFS and the default search strategy, all memory models have low memory usage, except for the forking model, which quickly reaches the 4GB memory limit.

```
1 a := word1
2 b := word2
3 d := $?
4 e := $?
```

**Figure 5: Symbolic input to *GNU make*, where ? denotes a symbolic character.**

## 4.3 *GNU make*

*GNU make* [25] is a popular build system, used extensively in the open source community. It is a larger program, consisting of about 35,000 lines of code. It uses significantly more memory during runtime than *m4*. To make execution easier for KLEE, we reduced the sizes of several caches in *make*.

Our evaluation focused on the variable expansion capabilities of *make*. Similarly to *m4*, we made the makefiles only partially symbolic. Figure 5 illustrates the makefile we used as symbolic input in our experiments.

Figure 7 shows the memory consumption of the different memory models. This benchmark was the only one where none of the runs terminated before reaching the timeout. Unsurprisingly, the forking model's memory consumption grows the quickest. The two segmented memory models behave similarly, with the SVF version consuming a constant amount of additional memory due to holding the whole points-to set.

The merging model appears to have the best memory consumption, but this is only because it executes two times fewer instructions than the next slowest approach (SVF segmented memory). When compared to *m4*, the *make* runs are slower (execute fewer instructions per second), therefore the differences between the memory models are also smaller.

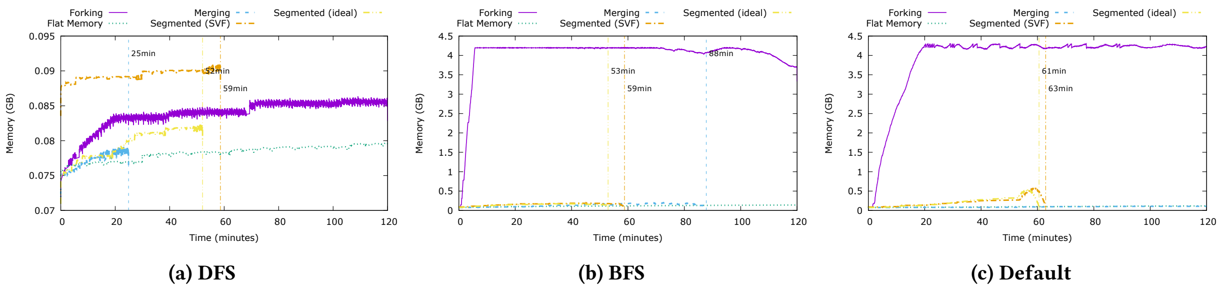
Since this benchmark is not terminating, we report the coverage. For DFS and BFS, the coverage was the same across all memory memory models at 21%. For the default heuristic, the forking and ideal segmented memory models had the highest coverage at 21.49%, followed by SVF segmented memory model at 18.75% and the flat memory model at 18.27%. Merging only achieved 15.44% coverage.

## 4.4 SQLite

*SQLite* [33] is a SQL database engine library written in C, which claims to be the most used database in the world. It has a big codebase with over 200,000 lines of code. We focused our evaluation on a part of *SQLite* that triggers multiple resolutions in symbolic execution.

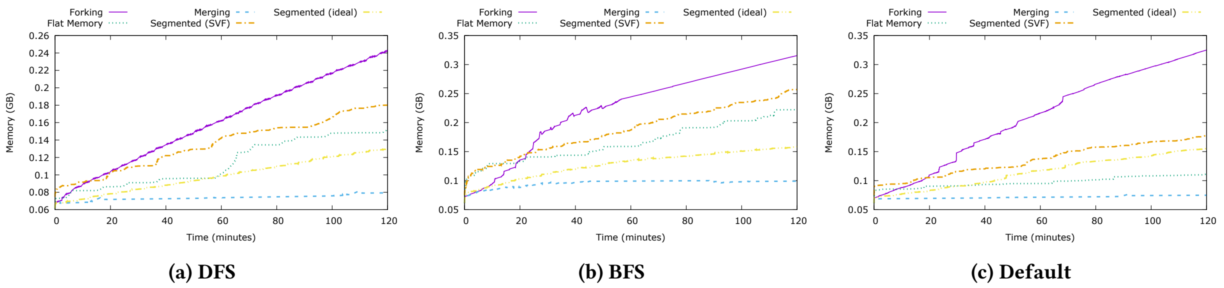
In particular, *SQLite* uses a hash table to keep track of all triggers associated with a table. We create several triggers with concrete names and then try to create a trigger with symbolic name, which should involve a lookup of a symbolic key in the hash table of triggers.

We use an in-memory database, create a table with two `int` columns and add 15 triggers to this table. We need to create more than 10 triggers as the *SQLite* hash table is optimized to be just a linked-list when there are fewer than 10 elements. Choosing a



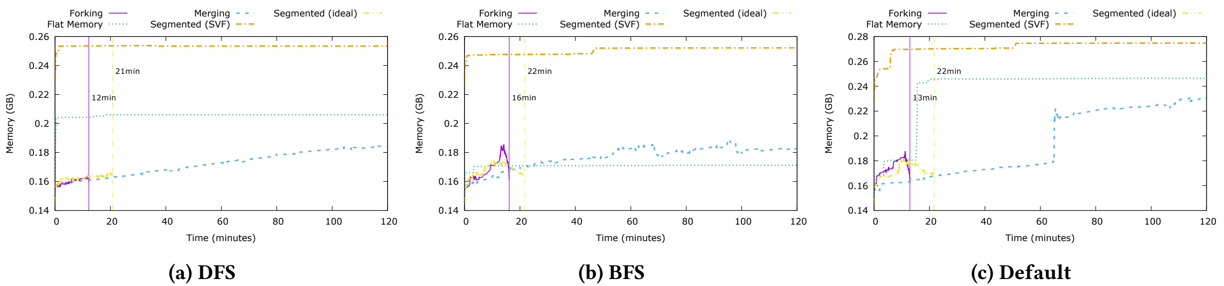

(a) DFS (b) BFS (c) Default

**Figure 6: Runtime and memory consumption of the different memory models for *GNU m4* across different search strategies.**



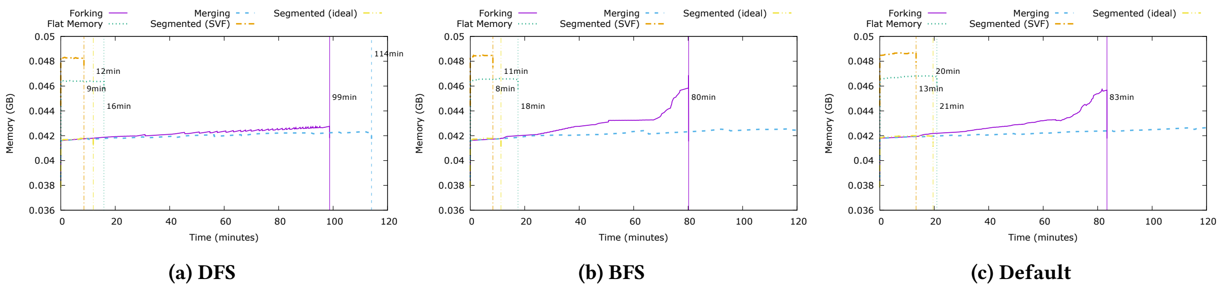

(a) DFS (b) BFS (c) Default

**Figure 7: Memory consumption of the different memory models for *GNU make* across different search strategies.**




(a) DFS (b) BFS (c) Default

**Figure 8: Runtime and memory consumption of the different memory models for *SQLite* across different search strategies.**[1]




(a) DFS (b) BFS (c) Default

**Figure 9: Runtime and memory consumption of the different memory models for *APR* across different search strategies.**

higher amount of triggers would tilt the results against the forking model, but we believe 15 triggers is a good trade-off between showcasing the multiple dereference issue and a realistic use of the library. Finally, we create two more triggers whose names are symbolic.

Figure 8 shows the results. The forking approach terminates in 12-16 minutes, the ideal segmented memory model terminates in

[1] These *SQLite* graphs have been amended to correct a mistake found after publication. We thank David Trabish for his feedback while reproducing our experiments.

```
1 apr_hash_t *ht = apr_hash_make();
2 for (int i = 0; i < 15; i++) {
3     int *key = malloc(sizeof int);
4     *key = i;
5     apr_hash_set(ht, key, sizeof(int), "A_value");
6 }
7
8 int i, j = symbolic();
9 apr_hash_get(ht, &i, sizeof(int));
10 apr_hash_get(ht, &j, sizeof(int));
```

**Figure 10: Apache Portable Runtime baseline benchmark.**

21-22 minutes, while the SVF segmented memory, flat memory and merging models don't terminate within the 2-hour time budget.

There are no significant differences in memory consumption across search strategies. SVF segmented memory uses a constant additional amount of memory since it needs to be holding the points-to sets in memory. The flat memory model uses more memory due to holding expressions over large arrays in memory.

This case study raises two interesting points. First, the precision of the points-to analysis significantly impacts the performance of the segmented memory model. As can be seen in Table 1, the difference in the size of the computed segments between ideal points-to analysis and SVF is huge. This is has an immediate impact on the performance of the segmented memory model, as illustrated in Figure 8. Therefore, improving the precision of the points-to analysis is a promising future avenue for improving the segmented memory model.

Second, the segmented memory model performs worse than forking, when there is little or no forking in the program. This is because grouping memory objects together increases the size of solver arrays, which makes constraints harder. This is a price the segmented memory approach has to pay regardless of whether the program causes multiple resolutions. In this case, the constraints in the forking model can be solved relatively fast, so forking can brute force its way through multiple resolutions. We note that increasing the number of triggers in the benchmarks would likely tilt results against the forking model, but our choice of 15 triggers was so that at least some runs finish within the 2-hour time budget.

## 4.5 Apache Portable Runtime

Apache Portable Runtime (*APR*) [2] is a C library that provides cross-platform functionality for memory allocation, file operations, containers, networking and threads, among others. It was initially used by Apache HTTP Server, but its use now extends to projects such as LibreOffice and Apache Subversion.

We focused our evaluation on *APR*'s hash table API. At a high level, we add 15 elements to a hash table and then do two symbolic lookups in this table. The number of keys and lookups was chosen to be the same as for *SQLite*. The skeleton code is shown in Figure 10. We used the default hashing algorithm for the *int* keys, which is the popular "times 33" algorithm, used by Perl for example.

Figure 9 shows the results. All runs terminate, except for merging under BFS and the default search strategy. Merging is also the slowest under DFS, taking 114 minutes to terminate. Forking terminates in 80-99 minutes, depending on the strategy. Flat memory performs well on this benchmark, terminating in 16-21 minutes. The segmented memory models (SVF) perform the best, terminating in 8-13 minutes. Memory consumption is small overall.

The most interesting observation about this benchmark is that the segmented memory model with ideal points-to analysis performs slightly worse than the one with SVF analysis. This is due to an interesting interaction with the solver. To understand why, we need to describe the relevant memory objects involved. These are a large 8.2KB memory object and several small objects, totalling about 120 bytes in size. The ideal points-to analysis groups the small objects into a single segment and puts the large 8.2KB object into a separate segment. The SVF analysis bundles both the 8.2KB object and the small objects into a single segment. During execution, there are queries involving arrays associated with both the 8.2KB object and the small objects. The ideal segmented memory model generates constraints with two arrays, one with 120 bytes and one with 8.2KB. The SVF segmented memory model generates constraints with only a single array. This gives better solver query caching for SVF segmented memory model, which results in a quicker runtime.

However, we note that this was the only benchmark where such a difference is observed, and a more precise points-to analysis is otherwise associated with better runtime performance. However, this case study suggest that there is future work in improving caching in symbolic execution, which would have implications beyond this memory model.

# 5 DISCUSSION

Our experience shows that our segmented memory model can effectively deal with multiple resolutions that occur in the context of complex data structures such as hash tables. However, the model is only useful to the extent that the code triggers such symbolic dereferences.

To get an idea of how our model performs when there are no multiple resolutions, we performed an experiment on *GNU Coreutils* version 8.29 [16]. We first ran all 105 utilities for 60 minutes with KLEE under the (default) forking model using depth-first search strategy and recorded the number of instructions KLEE executed for each utility. Then we ran each utility again up to the number of recorded instructions, with both the forking model (as a sanity check) and our segmented memory model, using a larger timeout of 80 minutes. A total of 18 utilities timed out after 80 minutes with the segmented memory model. For the remaining utilities, the segmented memory model was between 70% slower and 20% faster, but on average 4% slower. *GNU Coreutils* are an unfavorable case for our memory model, because the lack of multiple resolutions means that the model incurs a cost without any benefits.

While for most utilities, the impact is not significant, for 18 utilities it is. The cost is due to the grouping of arrays into memory segments, which can significantly increase the difficulty of the constraints sent to the solver. Therefore, we believe that our approach should be enabled on demand: one would first run the forking approach, and only if that run reports several multiple resolution cases with large branching factors, switch to the segmented memory approach. In our experience, anything with more than 5 occurrences of multiple resolution with a branching factor of more

than 10 should be sufficient for our proposed model to become beneficial.

Furthermore, the approach is particularly useful when one is interested in reasoning about properties requiring all-value analysis for the paths explored. When forking is used, such reasoning is not possible, as individual control-flow paths are split. If only high coverage is of interest, the single object or forking models might sometimes be more appropriate. Our approach also provides a convenient way to trade off forking and all-value reasoning, by adjusting the threshold for the size of memory segments. We also note that the flat memory and merging approaches can also perform all-value analysis.

We found the merging approach to be a competitive alternative to segmented memory. However, in our experiments the merging approach performed worse overall (e.g. unlike the segmented memory approach, merging timed out for all three *SQLite* experiments and on two out of the three *APR* experiments) and was more sensitive to the search heuristic used. However, in some cases it performed better than the segmented approach (e.g. *m4* with DFS). As a threat to validity, as mentioned before, we note that our implementation of merging might not be optimal, but we believe this threat to be small (see §3).

Our comparison between the two versions of the segmented memory model—one using SVF and the other an approximation of the ideal points-to sets—shows that the precision of the points-to analysis directly influences the performance of our approach. However, we found an interesting case where better precision slightly decreased performance, suggesting that in some cases merging arrays could be beneficial.

# 6 RELATED WORK

Symbolic execution has attracted a lot of attention recently, with different tools implemented for several different languages [5, 6, 11, 23, 26, 28]. These tools use different memory models, which influence both their power and scalability.

CUTE [31] introduced a simple memory model, which only handles equalities and inequalities for symbolic pointers. As discussed in §2.1, EXE [7] and CREST [11] implement the single-memory model, FuzzBALL [26] a generalisation of it, KLEE [5] the forking model, and an extension of SAGE [13] and Angr [32] the merging model.

A recent idea paper [14] proposes a model that associates symbolic addresses with values, along with a time stamp and a condition. The symbolic memory is then represented as a list of these associations. When a read occurs, the most recent value matching the address and the condition is returned. ESBMC [10] uses a similar technique of chaining if-then-else expressions to model pointers that can point to multiple objects, in the context of model checking.

MAYHEM [9] creates a new memory object on each read operation, which is a subset of the whole memory that the read operation can alias. This approach is at a high-level similar to ours, but there are important differences. In particular, the approach does not handle writes via symbolic pointers that may refer to multiple objects—instead, these pointers are concretised as in the single-object model. Furthermore, the approach still involves solver queries, as in the forking model, to determine the objects to which the pointer may refer. Unfortunately, a direct comparison with Mayhem is not possible, as the code is closed-source, and the memory model is complex enough to make a reimplementation difficult.

David et al. [12] summarise the concretisation approaches, such as the ones employed by MAYHEM and EXE, by proposing a framework for specifying concretization-symbolisation policies. While our approach strives to avoid the need for concretisation, it still uses concrete addresses to identify memory objects.

Trtík and Strejček [35] present a fully symbolic *segment-offset-plane* memory model. They split memory operations involving different types (such as *int*s or *float*s) into different planes, which resemble memory segments, but their solution may group together memory objects which would have been separated in our model.

Research on lazy initialisation for symbolic execution of Java code explores different ways of initialising symbolic memory object references and thus exploring different memory layouts [4, 21, 30]. By contrast, our work improves symbolic execution given a particular memory layout, notably it does not create new objects. We believe our work is complementary, and necessary to efficiently implement this work in symbolic executors for lower-level languages such as KLEE.

The idea of partitioning memory into segments is not new. Lattner and Adve [22] used points-to analysis to partition the heap as in our work. However, they only considered uses in compiler optimisation, whereas the novelty of our work stems from employing it to improve symbolic execution.

Bouillaguet et al. [3] apply a similar idea of partitioning memory based on points-to analysis to deductive verification. They require the points-to analysis to be sound and context-sensitive, whereas our approach is more tolerant to errors and imprecision in the analysis. Their evaluation does not consider real programs and only focuses on verifying a small sort function.

# 7 CONCLUSION

Programs that use complex heap data structures, in which a pointer is allowed to refer to more than one memory object, often cause path explosion and memory exhaustion in symbolic execution. We present a novel segmented memory model for symbolic execution, which uses pointer alias analysis to group objects into memory segments. Our segmented memory model can significantly reduce forking due to symbolic dereferences, sometimes even completely eliminating the need for forking.

We evaluated our approach on *GNU m4*, *GNU make*, *SQLite* and Apache Portable Runtime, highlighting its advantages in terms of performance and memory consumption, as well as its inherent limitations.

# ACKNOWLEDGEMENTS

We thank Frank Busse, Martin Nowack and the anonymous reviewers for feedback on the paper. This research was generously sponsored by the EPSRC through grants EP/N007166/1, EP/L002795/1 and a PhD studentship.

# REFERENCES


[1] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Technical Report.

[2] APR. 2019. Apache Portable Runtime. https://apr.apache.org/.

[3] Quentin Bouillaguet, François Bobot, Mihaela Sighireanu, and Boris Yakobowski. 2019. Exploiting Pointer Analysis in Memory Models for Deductive Verification. In *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'19)*.

[4] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining Symbolic Execution and Search-based Testing for Programs with Complex Heap Inputs. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'17)*.

[5] 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)*.

[6] 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)*.

[7] 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)*.

[8] 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. https://doi.org/10.1145/1455518.1455522

[9] 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)*.

[10] 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.

[11] CREST: Automatic Test Generation Tool for C [n.d.]. CREST: Automatic Test Generation Tool for C. https://github.com/jburnim/crest.

[12] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. 2016. Specification of Concretization and Symbolization Policies in Symbolic Execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'16)*.

[13] 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)*.

[14] Camil Demetrescu Emilio Coppa, Daniele Cono D'Elia. 2017. Rethinking Pointer Reasoning in Symbolic Execution. In *Proc. of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17)*.

[15] 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)*.

[16] GNU. 2019. GNU Coreutils. https://www.gnu.org/software/coreutils/.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*.

[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)*.

[19] Michael Hind and Anthony Pioli. 2001. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming* 39, 1 (2001), 31 – 55. Static Program Analysis (SAS'98).

[20] 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)*.

[21] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*.

[22] Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*.

[23] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. KLOVER: a symbolic execution and automatic test generation tool for C++ programs. In *Proc. of the 23rd International Conference on Computer-Aided Verification (CAV'11)*.

[24] M4. 2019. GNU M4. https://www.gnu.org/software/m4/.

[25] Make. 2019. GNU Make. https://www.gnu.org/software/make/.

[26] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*.

[27] 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)*.

[28] 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. *Automated Software Engineering* 20, 3 (01 Sept. 2013), 391–425.

[29] Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave Propagation and Deep Propagation for Pointer Analysis. In *Proc. of the 7th International Symposium on Code Generation and Optimization (CGO'09)*.

[30] N. Rosner, J. Geldenhuys, N. M. Aguirre, W. Visser, and M. F. Frias. 2015. BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support. *IEEE Transactions on Software Engineering (TSE)* 41, 7 (July 2015), 639–660.

[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)*.

[32] 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)*.

[33] SQLite. 2019. SQLite Database Engine. https://www.sqlite.org/.

[34] 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)*.

[35] Marek Trtík and Jan Strejček. 2014. Symbolic Memory with Pointers. In *Automated Technology for Verification and Analysis (ATVA)*.