Symbolic Memory with Pointers*

Marek Trtík[1,**] and Jan Strejček[2]

[1] VERIMAG, Grenoble, France
Marek.Trtik@imag.fr
[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic
strejcek@fi.muni.cz

Abstract. We introduce a segment-offset-plane memory model for symbolic execution that supports symbolic pointers, allocations of memory blocks of symbolic sizes, and multi-writes. We further describe our efficient implementation of the model in a free open-source project Bugst. Experimental results provide empirical evidence that the implemented memory model effectively tackles the variable storage-referencing problem of symbolic execution.

1 Introduction

Symbolic execution [9, 2, 7] is a classic automated program analysis technique based on a simple idea to execute a program on symbols representing arbitrary input data. It is nowadays used in many automatic test-generation and bug-finding tools including industrial ones. Some of the best known tools are Exe [4], Klee [3], Cute [10], Sage [6], and Pex [15].

As symbolic execution runs a program on symbols instead of concrete input data, it has to manipulate expressions over these symbols instead of standard datatype values like integers or floats. However, reading and writing symbolic expressions is not the main problem associated with memory in symbolic execution. It is the *variable storage-referencing problem* originally presented by King [9]. The problem appears when one needs to read a value from (or write a value to) a memory location dependent on input symbols. For example, if we want to execute an assignment `A[i]:=0`, the memory location that should be set to 0 depends on the symbolic value stored in `i`. The issue becomes even more serious when we introduce pointers because a symbolic pointer may point literally to any memory location, not only to elements of one array.

King [9] proposed two possible solutions of the problem for symbolic execution (and he immediately mentioned that *'neither is very satisfactory'*):

1. Symbolic execution is forked for each memory location which is a potential concrete value of the symbolic pointer. This solution leads to an exhaustive case analysis. This approach is further improved in [8, 5].

* The authors have been supported by The Czech Science Foundation, grant GBP202/12/G061.

** The paper has been written during M. Trtík's doctoral study at Faculty of Informatics, Masaryk University, Brno, Czech Republic.

2. The second solution prevents intensive forking of symbolic execution by storing *conditional values* in the symbolic memory. For example, if array element `A[3]` has a value e and `i` has a value $\underline{i}$, then an assignment `A[i]:=0` changes the value of `A[3]` to $\mathbf{ite}(\underline{i} = 3, 0, e)$ meaning that the value is 0 if $\underline{i} = 3$, and e otherwise. Note that each write can theoretically prolong all memory records by one application of **ite**. Hence, this symbolic memory grows very quickly unless we use some reduction methods. Unfortunately, the reduction methods are typically expensive.

The presented symbolic memory elaborates on the second approach. Besides symbolic pointers, our approach also supports allocations of memory blocks of symbolic sizes and multi-writes, i.e. operations that write to symbolic number of memory locations at once. This is useful for example when one sets a block of allocated memory to 0, where the number of allocated bytes in the block is given by a symbolic expression.

Full description of our symbolic memory is divided into two parts: Section 2 explains our *segment-offset-plane memory model* and Section 3 then describes its implementation. Both the memory model and its implementation are designed to manipulate as simple expressions as possible, to make operations in symbolic memory efficient. The suggested symbolic memory provides just basic memory operations (i.e. allocation, read, write, deallocation, and test for memory initialisation). Handling of some advanced memory-related operations (e.g. manipulation with composed objects or unions) using our symbolic memory is discussed in Section 4. The high efficiency of our symbolic memory implementation is confirmed by measurements presented in Section 5.

2 Segment-Offset-Plane Memory Model

Our symbolic memory is not bound to any particular programming language or data types. For sake of accessibility, all examples use C statements and programs. Further, we assume that integers and pointers are 4 bytes long.

First we describe the structure of the memory model. The crucial memory operations (namely allocation, read, and write) are then illustrated on a simple example. Finally, we introduce extended versions of allocation and write operations called *multi-allocation* and *multi-write*. The remaining two operations provided by our symbolic memory interface (namely deallocation and test for memory initialisation) are described in the following section.

2.1 Structure of the Model

Structure of the model reflects needs of symbolic execution. A specific aspect of memory allocations in symbolic execution is that sizes of requested allocations can be given by symbolic expressions instead of concrete numbers. For example, if an integer variable `n` has a value represented by a symbol $\underline{n}$, then symbolic execution of `malloc(n * sizeof(int))` allocates $4\underline{n}$ bytes, which can represent

4 bytes as well as 4 megabytes. If we use a standard memory model where memory cells are ordered into a linear sequence, it is very complicated to track which cells are allocated and which are free. We rather represent every allocated block as an isolated part of the memory called *segment*. Each segment is identified by a unique integer number. Memory cells within the block are identified by a nonnegative integer called *offset*. Hence, an address in our memory model is a pair *segment*:*offset*.

Further, write and read operations know what type of data they manipulate. Our memory model takes advantage of this fact and stores data of each basic type into a separated part of the memory called *plane*. The separation increases performance of our symbolic memory. For example, if we read an integer, we do not have to deal with chars, floats, or any data of other types stored in the memory. Pointers are composite datatypes and thus they are stored in two planes: segments in the plane *Segments* and offsets in the plane *Offsets*.

The memory model is called *segment-offset-plane* as every read or write operation needs to know the address (i.e. its segment and its offset) and the plane it should read from or write to.

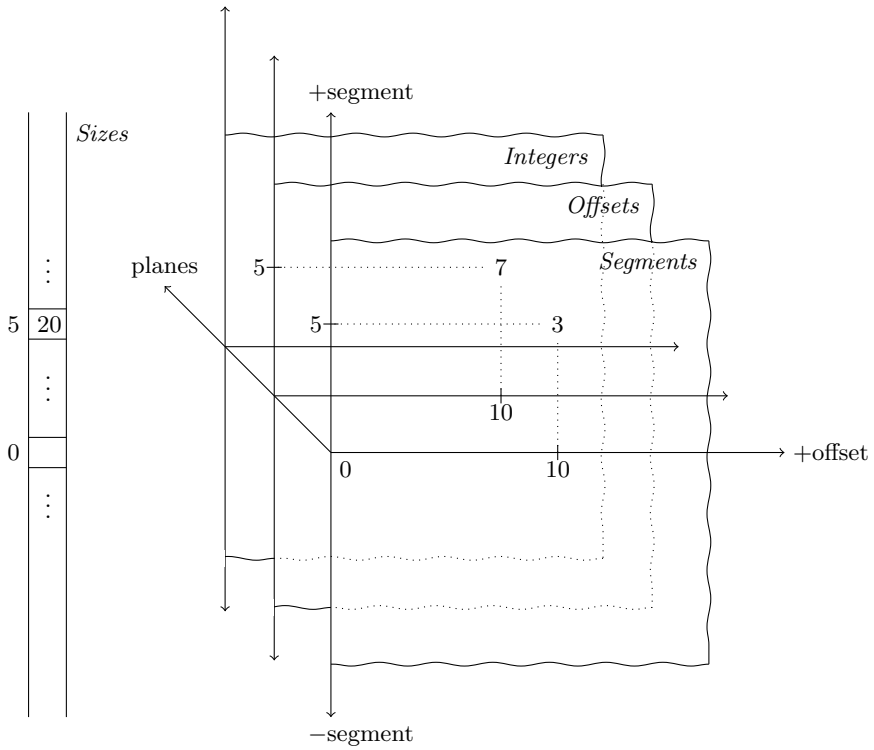

Fig. 1. A simple instance of the segment-offset-plane memory model.

Figure 1 depicts a simple instance of the segment-offset-plane memory model. The segment axis (vertical) has two directions to positive and negative values. The offset axis (horizontal) has only one direction to positive values. On the planes axis there are depicted three parallel memory planes. They share the same address space. In the picture there is further depicted a pointer $3:7$ which is stored at the address $5:10$. We see that the segment 3 of the pointer is stored in the *Segments* memory plane, while the offset 7 is stored in the plane *Offsets*. Although both segment and offset are stored at the same address, they are stored into different planes. The figure also shows an array *Sizes* that stores the size of each allocated segment. In the figure, the size of segment 5 is 20 bytes. As we mentioned above, the size of a segment can be symbolic. Segments with size 0 are not allocated. The segment 0 is never used to store any data: addresses with segment 0 are interpreted as NULL pointers (assigning NULL sets a pointer to $0:0$).

In our approach, memory operations do not automatically check whether addresses they work with are allocated or not. Instead, we provide an additional function that checks whether a given address points to an allocated memory or not. This function simply looks into the given address in the array *Sizes* and checks whether the value is greater then zero. We similarly do not implicitly test for memory initialisation in memory operations. We discuss details about the function providing memory initialisation test in the next section.

To unify the structures used in the model, we represent the array *Sizes* as another plane where we use only memory offsets 0, i.e. the size of a segment x is stored in plane *Sizes* at the address $x:0$.

The content of a plane is represented by a list of write records, where each record has the form $(segment : offset, value)$. In fact, the list reflects history of the plane content: a new record is always added at the end of the list. We use this representation just to explain principles of the model. An effective representation of a plane's content is described later in Section 3.

2.2 Basic Functionality of the Model

We explain the basic functionality of the memory model using a simple example. Let us consider the program depicted in Figure 2. The program contains a defi-

```
1   int* A = NULL;
2   int foo(int n, int i) {
3       A = (int*)malloc(n * sizeof(int));
4       A[3] = 777;
5       A[4] = 888;
6       A[3*i+1] = 999;
7       return A[3]+A[4];
8   }
```

Fig. 2. Running example with a global pointer variable `A` and a function `foo`.

nition of a global pointer variable `A` and a function `foo` accepting two parameters `n` and `i`. We symbolically execute the program with input values of variables `n` and `i` represented by symbols $\underline{n}$ and $\underline{i}$ respectively. As we are interested in an effect of statements forming the body of the function, we start by a description of the symbolic memory content just before execution of line 3. We especially need to know where program variables are stored in the memory and what are their values. Let the global variable `A` be stored at address $1:0$, and the stack variables `n` and `i` be stored at addresses $2:0$ and $2:4$ respectively. Note that we can consider the segment 1 as a memory block for the common 'data segment' of the program and we can consider the segment 2 as a memory block for the common 'stack segment' of the program. All other segments (except 0) then represent the program heap. As the program uses only pointers and integers, we will work with four planes: *Segments*, *Offsets*, *Integers*, *Sizes*. Before executing line 3, the planes have the following content:

$$\begin{aligned} \mathit{Segments} &\equiv [(1:0,0)] & \mathit{Integers} &\equiv [(2:0,\underline{n}),(2:4,\underline{i})] \\ \mathit{Offsets} &\equiv [(1:0,0)] & \mathit{Sizes} &\equiv [(1:0,4),(2:0,1024)] \end{aligned}$$

Note that the record $(1:0,4)$ in *Sizes* says that the data segment of the program consists of four bytes only. It is enough for storing the global pointer `A` (initialised to NULL, i.e. $0:0$) as we assume that a pointer is 4 bytes long. Further, the record $(2:0,1024)$ says that we reserved 1024 bytes for the program stack. Currently, only variables `n` and `i` occupy their 8 bytes. Note that instead of a fixed size of the stack we can introduce a fresh symbol for its symbolic size.

Execution of line `3` of the program results in two modifications in the memory. First, we allocate $4\underline{n}$ bytes of memory in the first free segment which is the segment 3. We do so by a single write into the plane *Sizes* and we get

$$\mathit{Sizes} \equiv [(1:0,4),(2:0,1024),(3:0,4\underline{n})].$$

Second, we assign the address $3:0$ of the first byte of the allocated memory to the pointer `A`. More precisely, the segment 3 and the offset 0 of the address are stored to the planes *Segments* and *Offsets* respectively, both to the address of `A` which is $1:0$. We have

$$\mathit{Segments} \equiv [(1:0,0),(1:0,3)] \quad \text{and} \quad \mathit{Offsets} \equiv [(1:0,0),(1:0,0)].$$

The statement at line `4` writes the value 777 to the address $3:(0+4\cdot 3)$ computed from the address $3:0$ (stored in the pointer `A`) by its increment by $4\cdot 3$ bytes (which is the size of 3 four–byte integers). We obtain

$$\mathit{Integers} \equiv [(2:0,\underline{n}),(2:4,\underline{i}),(3:4\cdot 3,777)].$$

The statements at lines `5` and `6` are resolved similarly. The resulting content of the plane *Integers* is

$$\mathit{Integers} \equiv [(2:0,\underline{n}),(2:4,\underline{i}),(3:4\cdot 3,777),(3:4\cdot 4,888),(3:4\cdot(3\underline{i}+1),999)].$$

Note that the last record refers to a symbolic offset. In fact, any part of a record is a symbolic expression (concrete number is a special kind of such expressions).

Now we execute line `7` with two read operations. The first operation reading `A[3]` is resolved in the plane *Integers* such that we compare the address where we read, i.e. the address $3 : 4 \cdot 3$, with addresses in all records in the list in the reverse order, and we build the composed **ite** expression

$$\begin{array}{l}\textbf{ite}(3 = 3 \ \wedge \ 4 \cdot 3 = 4 \cdot (3 \cdot \underline{i} + 1), 999,\\ \quad \textbf{ite}(3 = 3 \ \wedge \ 4 \cdot 3 = 4 \cdot 4, 888,\\ \quad\quad \textbf{ite}(3 = 3 \ \wedge \ 4 \cdot 3 = 4 \cdot 3, 777,\\ \quad\quad\quad \textbf{ite}(3 = 2 \ \wedge \ 4 \cdot 3 = 4, \underline{i},\\ \quad\quad\quad\quad \textbf{ite}(3 = 2 \ \wedge \ 4 \cdot 3 = 0, \underline{n},\\ \quad\quad\quad\quad\quad \delta(3, 4 \cdot 3))))),\end{array}$$

where $\delta(3, 4 \cdot 3)$ denotes a symbolic default value stored initially at the address $3 : 4 \cdot 3$ in the memory plane *Integers*. This default value can be used for detection of read operations from uninitialised memory. After few trivial simplifications we reduce the **ite** expression to $\textbf{ite}(2 = 3 \cdot \underline{i}, 999, 777)$. We can further see that the equation $2 = 3 \cdot \underline{i}$ does not have any solution, since $\underline{i}$ represents only integer values. With this knowledge we can simplify the expression even further to the final value 777.

Constraints like $2 = 3 \cdot \underline{i}$ can be resolved automatically by an SMT solver. Simplifications based on satisfiability checking of constraints have an important impact on size of expressions returned from the memory. As these expressions are often modified by the program and then stored back to the memory, the simplifications also reduce memory size and improve its performance. In Section 3 we present an actual implementation of the memory model, which substantially reduces the construction of compound **ite** expressions. In particular, the read of `A[3]` in our running example returns 777 without construction of any composed **ite** expressions.

The last memory operation of our running example reads `A[4]`. It proceeds in the same way as the previous one and results into the value $\textbf{ite}(1 = \underline{i}, 999, 888)$.

2.3 Multi-Writes and Multi-Allocations

Our memory supports *multi-write* operations that can change content of more memory locations at once. This ability has some natural applications in symbolic execution. We only sketch the concept of multi-writes using the example code

```
char A[n];
memset(A,0,n);
```

that allocates an array `A` of `n` bytes and sets all its elements to 0. Let $\underline{n}$ represent the value of `n`. We need to write to $\underline{n}$ addresses. Use of one multi-write is definitely more efficient here than iterating over the array and writing to one address each time, especially when we do not know the concrete length of the array.

Let us assume that the array A is stored at an address $\sigma_{\mathtt{A}} : \omega_{\mathtt{A}}$. Then we need to write 0 to every address with segment $\sigma_{\mathtt{A}}$ and an offset ω satisfying the formula

$$\phi(\omega) = (\omega_{\mathtt{A}} \leq \omega < \omega_{\mathtt{A}} + \underline{n}).$$

We can describe the addresses using λ-notation as $\sigma_{\mathtt{A}} : \lambda\omega.\, \phi(\omega)$.

Formally, we always work with λ-expressions of the form $\lambda\bar{\sigma}.\, \lambda\bar{\omega}.\, f(\bar{\sigma}, \bar{\omega})$, i.e. functions of both, a segment $\bar{\sigma}$ and an offset $\bar{\omega}$. Thus, the arguments of the considered multi-write are

$$(\sigma_{\mathtt{A}} : \lambda\bar{\sigma}.\, \lambda\bar{\omega}.\, \phi(\bar{\omega}), 0),$$

which is precisely the record that is added to the corresponding plane. In general, the values set by a multi-write operation do not have to be constant. They can also be given by a function of a segment and an offset. For example, the multi-write

$$(\sigma_{\mathtt{A}} : \lambda\bar{\sigma}.\, \lambda\bar{\omega}.\, \phi(\bar{\omega}),\ \lambda\bar{\sigma}.\, \lambda\bar{\omega}.\, (\bar{\omega} - \omega_{\mathtt{A}}) \bmod 2)$$

sets all even elements of the array to 0 and all odd elements to 1.

Besides multi-writes, our model also supports a multi-allocation that allocates a number of segments given by a symbolic expression at once. A multi-allocation is basically a multi-write into the *Sizes* memory plane. Segments allocated in this way have negative numbers. We provide more information about multi-allocations in the next section.

3 Implementation of the Memory Model

This section describes data structures used for effective representation of planes' contents. Further, it describes the algorithms for basic memory operations.

The implementation distinguishes two types of addresses: *constant* and *symbolic*. An address is constant if its segment and offset are both concrete integer numbers. Non-constant addresses are called symbolic. Note that a segment or an offset of a symbolic address is either a symbolic expression not equivalent to a concrete integer number, or a boolean λ-function determining a set of integers.

In the previous section, plane contents are represented as lists of records. As most memory operations work with concrete addresses, we use a specific structure to quickly resolve operations on these addresses. More precisely, the content of a plane is held in two structures: *boostMap* and *iteList*. The *boostMap* contains only data stored at concrete addresses and not colliding with any newer record. For example, symbolic address $3 : 4{\cdot}(3{\cdot}\underline{i}{+}1)$ collides with concrete address $3 : 52$ as the two addresses are identical when $\underline{i} = 4$. On the other hand, the symbolic address does not collide with $3 : 53$. The *boostMap* is implemented as a map assigning stored values to the corresponding constant addresses. All other records are stored in a doubly linked list called *iteList*, where the oldest record is at the beginning and the youngest at the end. An example of the two structures is depicted in Figure 3.

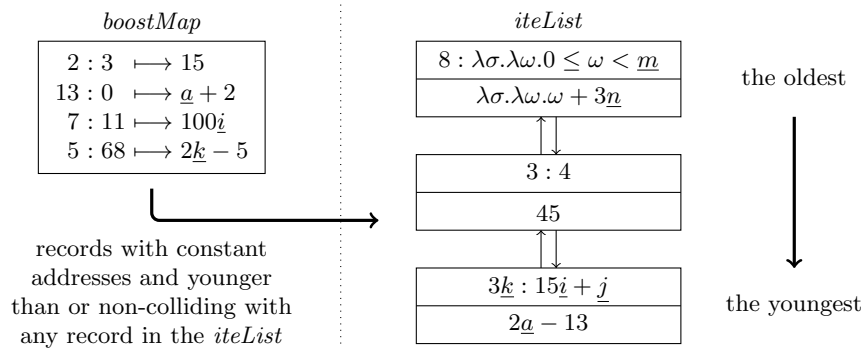

Fig. 3. An example of data structures *boostMap* and *iteList*. Each record in *iteList* is represented by an address (upper line) and the corresponding value (lower line).

Write Operation The write operation of a memory plane stores a passed symbolic expression ν at a given address $\sigma : \omega$. If the address is constant, the procedure is very simple: we remove the old value stored at the address from the *boostMap* (if any) and then we insert the pair $(\sigma : \omega, \nu)$ into the *boostMap*.

If the passed address is symbolic, then there is a possibility that it collides with some constant addresses stored in the *boostMap*. To keep the *boostMap* correct, we must first detect all such collisions and move the colliding records from the *boostMap* to the *iteList*. Let $(\sigma' : \omega', \nu')$ be a record with a concrete address stored in the *boostMap*. There is a collision between the addresses $\sigma' : \omega'$ and $\sigma : \omega$ iff the *collision formula* $\Gamma(\sigma', \omega', \sigma, \omega)$ defined as $\sigma' = \sigma \ \wedge \ \omega' = \omega$ is satisfiable.[3] We ask an SMT solver to decide the satisfiability. To be on the safe side, we assume that the formula is satisfiable even if the SMT solver returns UNKNOWN (recall that SMT queries can refer to some undecidable theories). If the formula is satisfiable, we remove the record $(\sigma' : \omega', \nu')$ from the *boostMap* and we insert it at the end of the *iteList*. Otherwise, the record remains in the *boostMap*. When all records in the *boostMap* are examined, we finish the write operation by inserting a new record $(\sigma : \omega, \nu)$ at the end of the *iteList*.

We illustrate the write operation using the example of Figure 2. Let us assume that the first 5 lines of the program are already symbolically executed. Since these lines call writes to constant addresses only, all data are stored in *boostMap*s. We focus on the plane *Integers*, which has the following content:

$$\begin{aligned} \mathit{Integers.boostMap} &= \{(2:0, \underline{n}), (2:4, \underline{i}), (3:4 \cdot 3, 777), (3:4 \cdot 4, 888)\} \\ \mathit{Integers.iteList} &= [] \end{aligned}$$

Execution of line 6 of the program produces a write of the record $(3 : 4(3\underline{i} + 1), 999)$ to the plane *Integers*. As the address $3 : 4(3\underline{i} + 1)$ is symbolic, the record

[3] The structure of the collision formula is slightly different if some of its arguments are λ-expressions. For example, $\sigma' : \omega'$ collides with $\sigma : \lambda\bar{\sigma}.\, \lambda\bar{\omega}.\, \phi(\bar{\sigma}, \bar{\omega})$ iff $\Gamma(\sigma', \omega', \sigma, \lambda\bar{\sigma}.\, \lambda\bar{\omega}.\, \phi(\bar{\sigma}, \bar{\omega}))$ defined as $\sigma' = \sigma \ \wedge \ \phi(\sigma', \omega')$ is satisfiable.

will be added to the *iteList*. Before we do so, we have to detect collisions of the records in the *boostMap* with the new record:

collision test	collision formula	result
$2:0 \quad \text{vs.} \quad 3:4(3\underline{i}+1)$	$2=3 \ \wedge \ 0=4(3\underline{i}+1)$	UNSAT
$2:4 \quad \text{vs.} \quad 3:4(3\underline{i}+1)$	$2=3 \ \wedge \ 4=4(3\underline{i}+1)$	UNSAT
$3:4\cdot 3 \quad \text{vs.} \quad 3:4(3\underline{i}+1)$	$3=3 \ \wedge \ 4\cdot 3=4(3\underline{i}+1)$	UNSAT
$3:4\cdot 4 \quad \text{vs.} \quad 3:4(3\underline{i}+1)$	$3=3 \ \wedge \ 4\cdot 4=4(3\underline{i}+1)$	SAT

Since only the collision with the last record in the *boostMap* is possible, only this one is moved into *iteList*. We finish the write operation by extending the *iteList* by the new record $(3, 4(3\underline{i}+1), 999)$. The updated plane is represented by:

$$\mathit{Integers.boostMap} = \{(2:0, \underline{n}), (2:4, \underline{i}), (3:4\cdot 3, 777)\}$$
$$\mathit{Integers.iteList} = [(3:4\cdot 4, 888), (3, 4(3\underline{i}+1), 999)]$$

Read Operation Given an address $\sigma:\omega$ and a memory plane, the read operation computes a single symbolic expression which determines the value stored in the memory plane at the passed address. If the address is constant, we check whether it lies in the domain of the *boostMap*. If so, we return the symbolic expression stored in the map for the address.

In all other cases, we construct a nested **ite** expression ψ holding the value. We initialise ψ to a symbol $\delta(\sigma,\omega)$ representing the default value stored in the plane at the passed address $\sigma:\omega$. This initialisation of ψ covers the case when no value has been written to the passed address so far. Now we enumerate records in the *iteList* from the oldest one to the youngest one. For each enumerated record $(\sigma':\omega', \nu')$ we check whether its address collides with $\sigma:\omega$. That is, we build the collision formula $\Gamma(\sigma',\omega',\sigma,\omega)$ as described in the write operation. If the formula is satisfiable, we update ψ to

$$\textbf{ite}(\Gamma(\sigma',\omega',\sigma,\omega), \rho, \psi), \text{ where } \rho = \begin{cases} \nu' & \text{iff } \nu' \text{ is not a } \lambda\text{-function,} \\ f(\sigma,\omega) & \text{iff } \nu' \equiv \lambda\bar{\sigma}.\,\lambda\bar{\omega}.\,f(\bar{\sigma},\bar{\omega}). \end{cases}$$

After processing all records in the *iteList*, we do the same with the records stored in the *boostMap* (processed in an arbitrary order) unless $\sigma:\omega$ is a constant address. If it is a constant address, we already know that it does not collide with any record in the *boostMap* as this was already checked at the beginning.

We illustrate the read operation using the example of Figure 2. We describe two read operations, from the array of integers allocated at line 3, performed during symbolic execution of the line 7. The content of the plane *Integers* after symbolic execution of the first 6 lines is already shown right before the description of the read operation. Execution of `A[3]` invokes the read operation in the plane *Integers* at the address $3:4\cdot 3$. Since there is a record in the *boostMap* for the address, we directly return its value 777. Execution of `A[4]` invokes the read operation in the plane *Integers* at the address $3:4\cdot 4$. Since there is no record in the *boostMap* for this address, we construct the resulting expression ψ from all records in the *iteList* as depicted in the following table:

collision test	collision formula	result	ψ
–	–	–	$\delta(3, 4 \cdot 4)$
$3 : 4 \cdot 4$ vs. $3 : 4 \cdot 4$	$3 = 3 \wedge 4 \cdot 4 = 4 \cdot 4$	SAT	888
$3 : 4(3\underline{i} + 1)$ vs. $3 : 4 \cdot 4$	$3 = 3 \wedge 4(3\underline{i} + 1) = 4 \cdot 4$	SAT	$\mathbf{ite}(\underline{i} = 1, 999, 888)$

In the first row there we initialise ψ to the default value $\delta(3, 4{\cdot}4)$. In the following lines we perform collision checks before we update ψ. Note that we automatically applied trivial simplification of ψ. In particular, in the second row we simplified ψ from $\mathbf{ite}(3 = 3 \wedge 4 \cdot 4 = 4 \cdot 4, 888, \delta(3, 4 \cdot 3))$ to 888 and in the third row we simplified the condition $3 = 3 \wedge 4(3\underline{i} + 1) = 4 \cdot 4$ to $\underline{i} = 1$.

Allocation and Deallocation We distinguish allocations of a single segment and multi-allocations. We maintain an *allocation counter* initialised to 1 and a *multi-allocation counter* initialised to -1.

An allocation of a single segment of a symbolic length ψ proceeds in the following three steps. Let γ be a value of the allocation counter before the allocation. In the first step we write the passed size ψ into *Sizes* memory plane at the address $\gamma : 0$. Next, the counter is updated to the value $\gamma + 1$. Finally, we return the address $\gamma : 0$ as the result of the allocation.

Let φ and ψ be symbolic expressions. A *multi-allocation* of φ segments of the common size ψ proceeds in three steps as well. Let γ be a value of the multi-allocation counter before the allocation. In the first step we write the size ψ into the plane *Sizes* at addresses $(\lambda\bar{\sigma}.\, \lambda\bar{\omega}.\, \gamma \geq \bar{\sigma} > \gamma - \varphi) : 0$. In the next step the multi-allocation counter is updated to the value $\gamma - \varphi$. Finally, we return the address $(\gamma - \varphi + 1) : 0$ as the result of the allocation. Note that the returned address points to the memory block with the lowest segment identifier.

Segment deallocation works exactly the same way for all memory blocks, regardless of types of their allocation. We simply write the number 0 into the memory plane *Sizes* at the passed address. Note that the offset of the passed address must always be the number 0, since all memory blocks are allocated at that offset. We can also perform a multi-deallocation by the corresponding multi-write into the *Sizes* memory plane. Note that deallocations do not change allocation and multi-allocation counters.

Test for Memory Initialisation Given an address $\sigma : \omega$ and a memory plane, test for memory initialisation returns a formula ψ over input symbols, which is valid for the concrete inputs for which the memory location $\sigma : \omega$ in the given plane is initialised. Computation of ψ proceeds as follows. If the address $\sigma : \omega$ is constant and it belongs to the domain of *boostMap* of the plane, then $\psi \equiv \mathit{true}$.

In all other cases, we construct ψ in form of disjunction. We first initialise ψ to *false*. Then we enumerate records in the *iteList* in any order. For each enumerated record $(\sigma' : \omega', \nu')$ we build the collision formula $\Gamma(\sigma', \omega', \sigma, \omega)$ as described in the write operation and then we update ψ to $\psi \vee \Gamma(\sigma', \omega', \sigma, \omega)$. After processing all records in the *iteList*, we do the same with the records stored in the *boostMap* (also processed in an arbitrary order) unless $\sigma : \omega$ is a constant address. If it is a constant address, we already know that it does not collide with any record in the *boostMap* as this was already checked at the beginning.

The passed address $\sigma : \omega$ can contain λ-expressions and thus it can represent a set of addresses. In this case, the returned formula ψ describes the concrete inputs for which *all* the represented locations are initialised. Hence, ψ is constructed in a slightly different way for addresses with λ-expressions. If $\sigma : \omega$ has the form $\lambda\bar{\sigma}.\, \lambda\bar{\omega}.\, \phi(\bar{\sigma}, \bar{\omega}) : \omega$, then ψ is defined as $\forall\bar{\sigma}.\, \phi(\bar{\sigma}, \omega) \rightarrow \psi'$, where ψ' is constructed by the algorithm described above for the address $\bar{\sigma} : \omega$ (instead for the original address $\sigma : \omega$). The construction of ψ for addresses with a λ-expression in the offset is similar.

Caching Satisfiability Queries During read or write operations not resolved by *boostMap*s we intensively construct collision formulae. We use an SMT solver to decide their satisfiability. Unfortunately, resolving SMT queries is usually very time consuming. Fortunately, the constructed collision formulae are often repeated. We thus implemented a cache in front of an SMT solver to improve amortised complexity of symbolic memory operations.

Remark Our implementation of symbolic memory can be further improved. For example, it currently never removes any record from an *iteList* even if one can easily construct an example where such a record becomes useless. We left the removal of useless records for future work for two reasons: it does not seem to be a bottleneck in our evaluation, and it can be expensive to decide whether a record in *iteList* is useless or not.

4 Use of the Memory in Symbolic Execution Tool

Our symbolic memory defines language- and platform-independent low-level memory layout with basic operations only. However, symbolic executors often need to handle some higher-level features of supported language. For example, symbolic executors of C programs have to handle composed data types, unions, `void*`, implementation of type casting expressions, etc. In this section, we suggest possible implementation of the mentioned high-level features using our low-level symbolic memory.

Composed data types A symbolic execution tool may create a plane for each basic data type of its instruction language and composed data (even nested) are treated simply as (nested) tuples of basic data types. So, individual attributes of an instance of a composed type are spread into the corresponding planes of basic types. Moreover, the tool can also introduce special separate planes for selected composed types of an analysed program.

Unions Union is a special composed data type, which can easily be represented such that its attributes reside in different planes but all at the same address.

void* Like other pointers, void pointers can be stored in the predefined planes *Segments* and *Offsets*. We do not define types for addresses, i.e. we treat all pointers the same way. It is responsibility of the tool to know which (pointer) variable has which type.

Type casting This feature of a programming language allows a programer to reinterpret meaning of referenced data. If the language also supports pointers and pointer arithmetic, then any sequence of bytes (starting basically at any address) can be reinterpreted according to programmer's will. This flexibility complicates designing of a symbolic executor. Here we show that our memory model provides a ground for efficient implementation of symbolic memory even for such flexible languages.

Let us consider the following C statement: `float f = *(float*)p;`, where `p` is of `int*` type. Obviously, the correct execution of this statement requires that the memory plane *Floats* contains at the address `p` such floating point number whose memory representation is equal to the memory representation of the integer stored at the same address in the plane *Integers*. This means that the last write into the plane *Integers* at the address `p` must have been extended by the corresponding write to the memory plane *Floats*.

With our memory model, a symbolic execution tool can optimise performance of the memory by implementing a data-flow analysis which detects all those write statements in the program whose extension is indeed necessary. Without any such analysis, the tool would have to extend each write such that it is performed to memory planes of all basic data types.

Note that in case of type-safe programming languages execution of type casting instructions is optimal in our memory model since no writes have to be extended. Therefore, performance of our memory scales according to properties of programming languages used in symbolic executors.

5 Experimental Results

We have implemented the symbolic memory as a library Segy of an open-source project called Bugst [14]. The library is used by a symbolic execution tool Rudla, which is another part of the project Bugst. The tool performs both classic [9] and compact [11] symbolic execution. We run Rudla on a collection of benchmarks from the category 'Loops' of SV–COMP 2013 [1], revision 229. We have chosen this category for two reasons. First, the benchmarks manipulate with arrays. Reading from and writing to array elements with input-dependent indexes lead to memory operations on non-constant addresses. Second, compact symbolic execution of program loops is the source of multi-write operations. The category contains 79 benchmarks, but only 70 of them can be translated into Rudla's internal program representation by the current version of Bugst. We symbolically executed each of the 70 benchmarks, both by classic and compact symbolic executions. Classic symbolic execution does not use multi-writes and thus the performed memory operations are relatively simple, while compact symbolic execution uses multi-writes which insert λ-expressions to the memory and make subsequent memory operations harder.

All experiments were performed on a laptop Acer Aspire 5920G (Intel® Core™ 2 Duo 2GHz, 2GB RAM) running Windows 7 Professional 64-bit. We used Z3 SMT Solver 4.3.0 [16] for deciding satisfiability queries. We apply a

settings		visited nodes
classic SE	naive implementation	417627
	efficient implementation	8083024
compact SE	naive implementation	219285
	efficient implementation	3547706

Table 1. Comparison of efficient and naive implementation of the memory model.

operation		*boostMap*				*iteList*			
		count		time		count		time	
		#	%	[s]	%	#	%	[s]	%
classic SE	write	7014327	99.98	15.50	69.01	1288(0)	0.02	6.959	30.99
	read	6765528	99.74	75.35	57.03	17748	0.26	56.77	42.97
compact SE	write	6365255	99.96	15.48	52.44	2698(236)	0.04	14.04	47.56
	read	3793442	98.38	65.81	26.48	62606	1.62	182.7	73.52
summary		23938552	99.65	172.1	39.78	84340	0.35	260.5	60.22

Table 2. Usage of *BoostMap* structures and *iteList* structures. Numbers in brackets are counts of multi-writes and they are included in the numbers of write operations.

five minutes timeout for execution of each benchmark. In all experiments, we present cumulative data for classic symbolic execution of the 70 benchmarks and for compact symbolic execution of the 70 benchmarks.

The first experiment compares overall efficiency of our implementation with a naive implementation of the segment-offset-plane model. The naive implementation (also available in Bugst library Segy) represents the content of a plane by a simple list of records. The naive read operation produces a nested **ite** expression containing all records of the list and asks an SMT solver to simplify it. Table 1 presents cumulative numbers of symbolic execution tree nodes visited during classic and compact symbolic executions of the 70 benchmarks (each with the five minutes timeout) using either the naive or the efficient symbolic memory implementation. The results show that classic symbolic execution runs more than 19 times faster when using our efficient implementation of the symbolic memory compared to the naive implementation. The compact symbolic execution with the efficient symbolic memory runs more than 16 times faster. The numbers of tree nodes visited by classic and compact symbolic executions should not be compared as nodes in compact trees have a slightly different semantics than nodes in classic symbolic execution trees.

The following experimental data provide more information about performance of our efficient implementation. We focus on read and write operations since they are essential for the memory. Note that memory allocations and deallocations are also considered as they are writes into the plane *Sizes* actually.

Table 2 shows total counts of read and write operations resolved purely by *boostMap* structures and the operations accessing *iteList* structures. The table

operation		*cache hits*				*cache misses*			
		count		time		count		time	
		#	%	[s]	%	#	%	[s]	%
classic	write	3791	83.28	0.360	5.55	761	16.72	6.131	94.45
SE	read	27031	85.04	3.385	7.69	4756	14.96	40.65	92.31
compact	write	2471	67.74	1.700	12.69	1177	32.26	11.70	87.31
SE	read	69231	83.80	25.59	16.96	13382	16.20	125.3	83.04
summary		102524	83.62	31.04	14.45	20076	16.38	183.78	85.55

Table 3. Efficiency of our cache in front of Z3 SMT solver.

also provides the total time (in seconds) of these operations. We always present absolute as well as relative numbers. One can see that overwhelming majority of the memory operations are resolved in *boostMap*s. The table also shows that accesses to *boostMap*s are much faster than those to *iteList*s. So we have an empirical evidence that implementation of memory planes by the two structures *boostMap* and *iteList* is indeed very important.

Although memory operations accessing *iteList*s are relatively slow, we actually achieved an impressive speed up by introducing a cache in front of an SMT solver called by memory operations (note that operations resolved by *boostMap*s do not produce SMT queries). Table 3 shows the counts of cache hits and cache misses. Again, we show also total time needed to solve the cached and non-cached SMT queries. We can see that more than 80% of all SMT queries led to cache hits and thus to very fast responses.

Finally, the results also show that the performance of our symbolic memory scales according to complexity of expressions passed to the memory: the ratio of operations resolved by *boostMap*s is higher for classic symbolic execution than for compact symbolic execution, and the same holds for cache hits of SMT queries.

6 Other Approaches to Symbolic Memory

As far as we know, our symbolic memory is the only one that supports fully symbolic addresses. Other recognized tools based on symbolic execution including EXE [4], KLEE [3], CUTE [10], SAGE [6], PEX [12], and SIMC [13] solve the variable-storage referencing problem in different ways. For example, KLEE and SAGE support symbolic offsets, but only concrete segments. If a pointer can point to n memory segments, KLEE clones symbolic execution n times and fix the segment part of the pointer to one of the segments in each clone. Concolic executors like SAGE often take advantage of the fact that they perform both concrete and symbolic execution along the same path. Hence, if a symbolic pointer can point to more segments, SAGE fix the segment part of the pointer to the value of this pointer in the corresponding concrete execution. Another approach is used in CUTE: it supports only pointers that are either `NULL`, or they point to a concrete address, or they directly correspond to some input symbol.

None of the mentioned tools support allocation of memory blocks of symbolic size or multi-writes.

Different approaches and abilities of our symbolic memory and symbolic memories of the above tools prevent their reasonable performance comparison. Indeed, other tools solve dereference of fully symbolic pointers outside symbolic memory. This allows them to use simpler structures and faster algorithms implementing symbolic memory, but for the price of more symbolic executions (due to cloning in KLEE) or loss of information (like in SAGE where a symbolic value is replaced by the corresponding value in a concrete execution).

7 Conclusion

We presented a symbolic memory supporting symbolic pointers, allocations of memory blocks of symbolic sizes, and multi-writes. The memory is based on storing conditional values. It uses the introduced segment-offset-plane memory model where addresses are *segment* : *offset* pairs. Data stored in the memory are distributed into memory planes according to their semantic information, e.g. data type. The model leads to a natural fragmentation of the memory, which makes memory operations faster. We also describe our implementation of the memory model that uses specific data structures and a cache for SMT queries to improve efficiency of the symbolic memory. Experimental results give us an empirical evidence that the implemented symbolic memory successfully tackles the variable storage-referencing problem.

References

1. D. Beyer. Second competition on software verification. In *TACAS*, volume 7795 of *LNCS*, pages 594–609. Springer, 2013.
2. R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – A formal system for testing and debugging programs by symbolic execution. In *ICRS*, pages 234–245. ACM, 1975.
3. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.
4. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335. ACM, 2006.
5. X. Deng, J. Lee, and Robby. Efficient and formal generalized symbolic execution. *Autom. Softw. Eng.*, 19(3):233–301, 2012.
6. B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA*, pages 129–140. ACM, 2009.
7. W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. Software Eng.*, 3:266–278, 1977.
8. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, volume 2619 of *LNCS*, pages 553–568. Springer, 2003.
9. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

10. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, pages 263–272. ACM, 2005.
11. J. Slaby, J. Strejček, and M. Trtík. Compact symbolic execution. In *ATVA*, volume 8172 of *LNCS*, pages 193–207. Springer, 2013.
12. D. Vanoverberghe, N. Tillmann, and F. Piessens. Test input generation for programs with pointers. In *TACAS/ETAPS*, pages 277–291. Springer-Verlag, 2009.
13. Z. Xu and J. Zhang. A test data generation tool for unit testing of C programs. In *QSIC*, pages 107–116. IEEE, 2006.
14. Bugst. http://sourceforge.net/projects/bugst.
15. Pex. http://research.microsoft.com/Pex.
16. Z3. http://z3.codeplex.com.