Address-Aware Query Caching for Symbolic Execution

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

Shachar Itzhaky
Technion
Haifa, Israel
shachari@cs.technion.ac.il

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

***Abstract*—Symbolic execution (SE) is a popular program analysis technique. SE heavily relies on satisfiability queries during path exploration, often resulting in the majority of the time being spent on solving these queries. Hence, it is not surprising that one of the most vital optimizations SE engines use is query caching. To increase the cache hit rate, queries are transformed into a normal form, which is used as a key for updating the cache. An obstacle to caching queries involving pointers is the presence of numerical address values, which are assigned by the engine according to its memory allocation scheme and are hard to canonicalize across different paths.**

In this paper, we propose a novel query caching technique that allows efficient handling of queries containing expressions that depend on address values. The key insight is that the result of such queries is in fact agnostic to the concrete address values occurring in them, subject to some basic memory safety constraints. This observation can be used to coalesce more queries during cache lookup, thus further increasing cache utilization.

Our extensive evaluation shows that our technique achieves significant performance gains when the analysis encounters queries containing symbolic pointers, while incurring only a modest performance overhead in other cases.

***Index Terms*—Symbolic execution, Query caching**

I. Introduction

Symbolic execution (SE) is a popular program analysis technique, that lies at the core of many applications including automatic test generation [16], [17], [39], bug finding [16], [26], debugging [29], patch testing [42], [33], automatic program repair [36], [34], and quantitative program analysis [22], [15]. During symbolic execution, the program is run with a *symbolic* input that is not constrained to any particular value, rather than with a concrete one. Whenever the symbolic executor reaches a branching point that depends on a symbolic value, an SMT solver [19] is used to determine the feasibility of each branch, and the appropriate paths are further explored while updating their paths constraints with corresponding constraints. Once the execution of a given path is completed, the SMT solver can be used to generate a solution for the corresponding path constraints, which represents a test case that can be used to replay that path.

Often, symbolic execution spends most of its time on solving queries [37], therefore the effectiveness of the symbolic execution engine depends on the effectiveness of its underlying SMT solver. To reduce the cost of constraint solving, symbolic execution engines perform their own optimizations before invoking the SMT solver. Among these optimizations are, for example: caching [16], [44], [47], slicing [16], [44], [47], expression rewriting [16], [40], and logical implications [28].

Symbolic executors perform a large number of queries, therefore one of the most vital optimizations is *query caching*. In this optimization, queries are transformed to a normal form, which is used as a key for maintaining the cache. Different tools adopt different normalization strategies [16], [44], [47], including: canonization, renaming variables, rewriting equalities, and arithmetic simplifications.

Unfortunately, there are still some types of queries that are inefficiently handled by existing query caching solutions. One type of such queries are the so-called *address-dependent* queries, i.e., queries that involve pointer expressions, which are generated in various symbolic execution tools such as KLEE [16], ANGR [44], Manticore [35], and SAGE [23]. To illustrate when such queries are encountered, consider the program in Figure 1. When the value of `array[i][j]` is read at line 17, the corresponding expression depends on the content of `array`, as it is accessed with the symbolic offset `i`. The contents of `array`'s cells are the address values assigned at line 11, therefore the query generated for the branch at line 17 is address-dependent. Due to the branch at line 5, the same flow described above is executed again while exploring a different execution path. Depending on the utilized allocation scheme, the addresses assigned at line 11 by the other symbolic state may be different from those assigned by the previously discussed symbolic state. Therefore, the queries generated at line 17 by these two symbolic states would be syntactically different, although they are clearly *equisatisfiable*. Since existing query caching techniques rely on some form of syntactic normalization, an opportunity to reuse the result of the first query for the second one would be missed. Notice that even if each symbolic state had its own local memory allocator, synchronizing the assigned addresses would be difficult: For example, if additional objects are allocated at line 6, then the two states forked at line 5 are likely to produce different allocation sequences.

In this paper, we introduce a novel query caching technique for symbolic execution, that can efficiently handle address-dependent queries. Such queries are prevalent in programs that dereference *symbolic pointers*, i.e., pointers whose values depend on the symbolic input. A prolific source for such

```
1 #define N (2)
2 #define MAGIC (7)
3
4 int z; // symbolic
5 if (z > 0) {
6    /* allocate objects... */
7 }
8
9 char **array = calloc(N, sizeof(char *));
10 for (unsigned int i = 0; i < N; i++) {
11    array[i] = calloc(N, 1);
12 }
13 array[0][1] = MAGIC;
14
15 unsigned int i; // symbolic, i < N
16 unsigned int j; // symbolic, j < N
17 if (array[i][j] == MAGIC) {
18    /* do something... */
19 }
```

Fig. 1: Motivating example.

programs are programs where the symbolic input propagates into a data structure indexed by that input, as happens for example with hash tables.

At a high level, we utilize the *symbolic addressing model* introduced in [45] to modify the representation of the expressions generated by the symbolic execution engine, such that the concrete address values returned by the allocator are replaced with symbolic values. This allows to track the propagation of these address values to the symbolic states and the subsequent queries, and distinguish pointer expressions from non-address integer expressions. Using this model, we are able to detect address-dependent queries which are not syntactically equivalent but are nonetheless equisatisfiable, thus improving the cache utilization.

Our main contributions can be summarized as follows:

1) We propose a novel query caching technique, that allows efficient handling of address-dependent queries.
2) We give a formal proof for the correctness guarantees of our technique.
3) We provide a KLEE-based implementation, which we make available as open-source.[1]
4) We provide in our evaluation an additional empirical validation, and show that our technique can achieve significant performance gains.

II. Preliminaries

To understand why efficiently caching address-dependent constraints is challenging, we first give some background about the existing addressing model and the existing approach for query caching in modern analysis tools [17], [16], [44], [47], [15].

A. Addressing Model

In modern symbolic executors, e.g., KLEE and Manticore, the address space is represented using a set of *memory objects*:

$$(b, s, a) \in N^+ \times N^+ \times A$$

[1]https://github.com/davidtr1037/klee-aaqc

When a logical object in the program is allocated, it is associated with a memory object (b, s, a) that has a concrete *base address* $b \in N^+$ and spans $s \in N^+$ bytes. In addition, a memory object is backed by an SMT array $a \in A$ with the same size s, which tracks the values stored into the memory object.[2] If a value v is written to the eth byte of the object, the array a is replaced by a new array expressed using a *write* (also known as *store*) expression: $wr(a, e, v)$. If the eth byte of the object is accessed, the read value is expressed using a *read* (also known as *select*) expression: $rd(a, e)$.[3]

The allocated objects span distinct address ranges, i.e., for every two distinct memory objects (b_1, s_1, a_1) and (b_2, s_2, a_2) it holds that:

$$[b_1, b_1 + s_1) \cap [b_2, b_2 + s_2) = \emptyset.$$

This *non-overlapping* requirement reflects the fact that different objects are located at different parts of the memory, and enables identifying memory objects by addresses: A concrete address can belong to at most one object. Thus, when the program accesses that memory location, the SE engine can determine which SMT array represents the content at that address and act accordingly. To detect buffer-overflow errors, the engine allocates after each memory object in the address space a so-called *red zone*: An unmapped memory region residing between each two consecutive memory objects.

B. Query Caching

Symbolic executors [17], [16], [44] and other analysis frameworks [47], [15] use some form of *syntactic* query caching, to improve the performance of constraint solving. Each query is transformed to an equivalent normal form according to some syntactic rules, and this normal form is used as a key for maintaining the cache: In case of a miss, the query is solved using the SMT solver and its result is memoized. Otherwise, the result of a previously solved query is reused without invoking the SMT solver. The normalization is typically achieved via variable renaming, canonization, arithmetic simplification, and equality rewriting.

For instance, consider the following two queries:

$$x < 1 \ \wedge \ x + y + 4 < 7 \qquad x + z + 1 < 4 \ \wedge \ x < 1$$

These queries are syntactically different, but they can be reduced to the same normal form:

$$v_0 < 1 \wedge v_0 + v_1 < 3$$

Therefore, if one of these queries was already solved, the result of the other one could be reused later if needed. This query caching mechanism is effective in practice, but does not provide a *complete method* for determining if two formulas are equisatisfiable: There are queries that are equisatisfiable, but which cannot be reduced to the same normal form.

TABLE I: The queries generated at line 17 in different states with the two addressing models.

Model	State	Symbolic Pointer	Query	AC
Existing	s_3	$p_1 := rd(wr(wr(a_1, 0, 200), 1, 300), i) + j$	$q_1 : i < 2 \wedge j < 2 \wedge 200 \leq p_1 < 202 \wedge rd(a_2, p_1 - 200) = 7$	-
	s_4		$q_2 : i < 2 \wedge j < 2 \wedge 300 \leq p_1 < 302 \wedge rd(a_3, p_1 - 300) = 7$	-
	s_5	$p_2 := rd(wr(wr(a_1, 0, 500), 1, 600), i) + j$	$q_3 : i < 2 \wedge j < 2 \wedge 500 \leq p_2 < 502 \wedge rd(a_5, p_2 - 500) = 7$	-
	s_6		$q_4 : i < 2 \wedge j < 2 \wedge 600 \leq p_2 < 602 \wedge rd(a_6, p_2 - 600) = 7$	-
Symbolic	s_3	$p_3 := rd(wr(wr(a_4, 0, \beta_2), 1, \beta_3), i) + j$	$q_5 : i < 2 \wedge j < 2 \wedge \beta_2 \leq p_3 < \beta_2 + 2 \wedge rd(a_2, p_3 - \beta_2) = 7$	$\beta_2 = 200$
	s_4		$q_6 : i < 2 \wedge j < 2 \wedge \beta_3 \leq p_3 < \beta_3 + 2 \wedge rd(a_3, p_3 - \beta_3) = 7$	$\beta_3 = 300$
	s_5	$p_4 := rd(wr(wr(a_4, 0, \beta_5), 1, \beta_6), i) + j$	$q_7 : i < 2 \wedge j < 2 \wedge \beta_5 \leq p_4 < \beta_5 + 2 \wedge rd(a_5, p_4 - \beta_5) = 7$	$\beta_5 = 500$
	s_6		$q_8 : i < 2 \wedge j < 2 \wedge \beta_6 \leq p_4 < \beta_6 + 2 \wedge rd(a_6, p_4 - \beta_6) = 7$	$\beta_6 = 600$

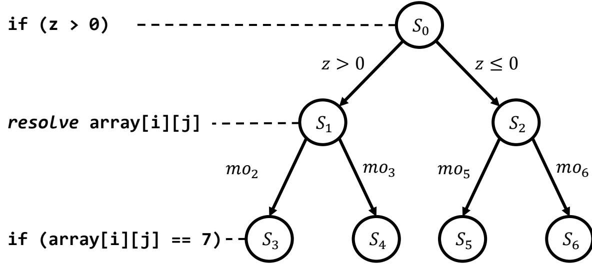

Fig. 2: The execution tree of the program from Figure 1.

III. Address-Aware Query Caching

Our technique enables a more efficient caching of *address dependent* queries, i.e., queries that contain pointer expressions. We achieve that by using a *symbolic addressing model* that modifies the representation of expressions in the symbolic state, in a way that enables distinguishing pointer expressions from integer expressions. We describe the symbolic addressing model in Section III-A, and then show how this model helps efficiently handle address dependent queries.

A. Symbolic Addressing Model

We adopt the addressing model proposed by [45] that was used in the original work in order to dynamically merge and split the underlying representations of memory objects. Here we exploit its ability to modify the representation of the expressions (and queries) generated by the symbolic execution engine, without merging or splitting memory object representations.

In this model, as before, the program's address space is represented using a set of *memory objects*:

$$(\beta, s, a) \in E \times N^{+} \times A$$

However, here the base address of an object is a symbolic value $\beta \in E$ instead of a concrete one, while different objects use distinct symbolic values (i.e., names) as their base addresses. There is still an underlying assumption that memory objects are non-overlapping, only that now, with symbolic base addresses, it is not as straightforward to uphold this property.

We preserve this property using *address-constraints* (AC), that maintain the correlation between symbolic addresses and concrete ones. When a memory object is allocated, we add an address constraint, i.e., an equality of the form $\beta = b$, where β is the symbolic address and b is a concrete address that is allocated in a non-overlapping manner (as in Section II-A). Note that the concrete address b remains *hidden* while β is the one that propagates to the symbolic state. To keep constraint solving efficient, we substitute the address constraints before sending a query to the solver, i.e., replace symbolic addresses with their corresponding concrete values.

To illustrate how this addressing model works, consider the program from Figure 1. Given 8-byte pointers, assume that the allocated object at line 9 is $(\beta_1, 16, a_1)$, and that the allocated objects at line 11 are $(\beta_2, 2, a_2)$ and $(\beta_3, 2, a_3)$. If the underlying concrete addresses are 100, 200, and 300, then the address constraints at line 13 are:

$$\{\beta_1 = 100, \beta_2 = 200, \beta_3 = 300\}$$

Note that the value of the variable `array` is β_1, and the values of its cells are β_2 and β_3. At line 17, where `array` is accessed with symbolic offset i, the value of `array[i]` is a *read* expression:

$$rd(wr(wr(a_1, 0, \beta_2), 1, \beta_3), i)$$

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

$$rd(wr(wr(a_1, 0, 200), 1, 300), i)$$

B. Caching Address Dependent Queries

To illustrate the need for the symbolic addressing model described in Section III-A, consider again the program in Figure 1 whose execution tree is shown in Figure 2. The analysis starts with the initial state s_0, which executes the symbolic branch at line 5 and forks the execution. Then each of the forked states, s_1 and s_2, allocates a two-dimensional matrix using an array of pointers (line 9), and initializes one of the cells to some constant value `MAGIC` (line 13). Say the memory objects allocated in state s_1 at lines 9 and 11 are:

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

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

$$mo_1 = (100, 16, a_1), mo_2 = (200, 2, a_2), mo_3 = (300, 2, a_3)$$

At line 17, where the value of `array[i][j]` is read in s_1, the value of the accessed pointer is:

$$p_1 := rd(wr(wr(a_1, 0, 200), 1, 300), i) + j$$

which is also shown under the column *Symbolic Pointer* of Table I. This is a symbolic pointer that can refer to two objects: mo_2 and mo_3. As a result, the execution is forked again, resulting in two new states: s_3 and s_4. When we follow s_3, the state that resolves p_1 to mo_2, the query generated for the branch condition at line 17 is q_1, which is shown in column *Query* of Table I. Note that this query does not contain the constraint $z > 0$, since the expression of the branch condition at line 17 does not depend on the symbolic value z. This optimization is known as *slicing*, and it's widely used in symbolic execution tools [16], [44].

As for s_2, the other state forked at line 5, say here that the memory objects allocated at lines 9 and 11 are:

$$mo_4 = (400, 16, a_4), mo_5 = (500, 2, a_5), mo_6 = (600, 2, a_6)$$

Similarly, s_5 executes the same flow as s_2, and the query generated for the branch condition at line 17 is q_3. The concrete address values assigned in the states s_3 and s_5 are different, so the mentioned queries (q_1 and q_3) cannot be reduced to the same normal form. Therefore, standard query caching cannot reuse the result of the first query for the second one.

Note that these two queries (q_1 and q_3) are equisatisfiable, and not by chance: The query generated at line 17 is *address-agnostic*, that is, for any memory layout that respects the *non-overlapping* property and the original sizes of the involved memory objects, the generated query will be equisatisfiable to both of the queries above.

In order to detect equisatisfiable address-dependent queries, we need to know which expressions in the constraints are pointer expressions. Since the existing addressing model encodes pointer values as integers, detecting these pointer expressions is hard without additional annotations. However, that can be easily achieved with the symbolic addressing model: In this model the assigned base addresses are symbolic values rather than integers, so if the normal form of one query can be obtained from the normal form of another query by renaming the symbolic base address values, and the sizes of the memory objects corresponding to the matched address values are equal, then these queries are equisatisfiable.

Using the symbolic addressing model in our example, instead of queries q_1 and q_3, we generate q_5 and q_7 in conjunction with the corresponding address constraints shown in column *AC* of Table I. Since q_7 can be obtained from q_5 by renaming β_2 to β_5 and β_3 to β_6, and the sizes of mo_2 and mo_3 match those of mo_5 and mo_6, then these two queries can be determined as equisatisfiable.

Table I shows the query generated at line 17 by each of the four states, with both the existing addressing model and the symbolic one. A similar equisatisfiability observation can be made regarding the queries q_6 and q_8, generated by s_4 and s_6.

Algorithm 1 Equisatisfiability Algorithm.

1: **function** EQUI-SAT(e_1, e_2, m)
2:     **if** *e1 and e2 are unary expressions*
3:         $op(e'_1) \leftarrow e_1,\ op(e'_2) \leftarrow e_2$
4:         **return** EQUI-SAT(e'_1, e'_2, m)
5:     ...
6:     **if** *e1 and e2 are atomic symbolic base addresses*
7:         $s_1 \leftarrow$ GET-SIZE(e_1), $s_2 \leftarrow$ GET-SIZE(e_2)
8:         **return** ADD-PAIR(m, e_1, e_2) and $s_1 = s_2$
9:     **if** *e1 and e2 are atomic arrays*
10:         **return** INIT-VAL(e_1) = INIT-VAL(e_2)
11:     **if** *e1 and e2 are write expressions*
12:         $wr(a_1, i_1, v_1) \leftarrow e_1,\ wr(a_2, i_2, v_2) \leftarrow e_2$
13:         **return** EQUI-SAT(a_1, a_2, m) and
14:             EQUI-SAT(i_1, i_2, m) and EQUI-SAT(v_1, v_2, m)

Our algorithm for determining the equisatisfiability of two queries is given in Algorithm 1. We assume that the expressions e_1 and e_2 passed to the function EQUI-SAT are represented as described in Section III-A, using the symbolic addressing model, i.e., base addresses are symbolic values. We also assume these expressions are already in a canonical form, and that they are represented using an abstract syntax tree (AST), with support for: integers, symbolic values, unary and binary operations, *read* and *write* operations, etc.

The algorithm is almost identical to a standard recursive equality checking routine, except for two main cases:

Symbolic Base Addresses. We use the bidirectional map `m`, to compute a *bijection* between the symbolic base addresses in e_1 and e_2, if such bijection exists. First, we update at line 8 the map `m` with the new pair (e_1, e_2) using the function ADD-PAIR, which returns `true` if the bijection property is preserved, and `false` otherwise. Then, if ADD-PAIR succeeds, we take the sizes of the memory objects corresponding to e_1 and e_2 (fetched at lines 7), and check their equality.

Arrays. We assume that atomic arrays (without *write*'s) are initialized using a vector of constants, which can be accessed using the INIT-VAL function. If e_1 and e_2 are atomic arrays (line 9), then we check that they are equally initialized using the INIT-VAL function. In the case of *write* expressions (line 11), we perform a recursive check on the corresponding arrays, indices, and values.

C. Limitations

The approach described in Section III-B cannot be applied to queries which are not *address-agnostic*, that is, queries where the ordering of the memory objects in the address space affects the satisfiability. Such queries may be generated explicitly by the symbolic execution engine, or when analyzing programs that incur undefined behavior.

Undefined Behavior. Indeed, there are programs whose execution depends on the relationships between the numerical address values. For example, the result of the branch statement at line 2 from Figure 3 clearly depends on the allocation scheme implemented by the C standard library. For this reason,

```
1 char *p = malloc(10), *q = malloc(50);
2 if (p > q)
3     ...
4 if (*(p + 100) == *q)
5     ...
```

Fig. 3: An ill-behaved program due to unsafe pointer arithmetic.

the behavior of such statements is commonly considered to be undefined. Note that not all pointer comparisons are necessarily address-dependent. Comparisons such as `p + i < p + j` have well-defined semantics, and comparisons between pointers within the same object are commonly used and introduced as part of standard compiler optimizations. However, it is known that checking the presence or absence of undefined behavior in a given program is hard [27].

Similarly, pointer arithmetic can also expose address dependency, as demonstrated in the program from Figure 3: The branch condition at line 4 holds, for example, when $p = 100$ and $q = 200$, but may be false under other address assignments. Again, verifying the absence of out-of-bounds pointer arithmetic is too hard in general [46], [25]. Symbolic executors can detect such bugs under some address assignments, but not all, and definitely cannot prove their absence.

In the presence of such undefined behavior, our query caching approach presented in Section III-B may exhibit unsoundness or incompleteness. However, we would point out that in these cases symbolic executors can not provide such guarantees anyway, although our approach can lead to more incorrect results.

Engine-Internal Queries. The symbolic execution engine itself may internally generate queries which are not address-agnostic. For example, when KLEE resolves a symbolic pointer p, it generates the query $\beta \leq p < \beta + s$ in order to check if p may refer to the memory object (β, s, a). To optimize the search procedure it generates an additional validity query of the form $p < \beta + s$ to determine if the search can be completed without scanning any additional memory objects. Clearly, the last query is not address-agnostic: Let p be the symbolic pointer encountered at line 17 from Figure 1:

$$rd(wr(wr(a_1, 0, \beta_2), 1, \beta_3), i) + j$$

If the address constraints are:

$$\beta_2 = 200, \beta_3 = 300, \beta = 700$$

then the optimization query from above is valid, which is not the case if $\beta = 100$.

In the case of *engine-internal* queries, the engine itself is generating the query, so it is easy to tag these queries and locally disable the query caching optimization for them.

In Section IV we formulate the sufficient conditions under which the address dependent queries generated by the symbolic execution engine are guaranteed to be address-agnostic, and prove the correctness of our query caching approach with respect to such queries.

IV. Correctness

In this section we justify the query caching approach described in Section III, by arguing that from the satisfiability or unsatifiability of a given query follows the same result under any isomorphic address spaces.

The objects of interest are logical formulas φ and logical structures M, where a structure satisfying a formula, i.e., a *model*, is denoted by $M \models \varphi$. For the interpretations of expressions (terms) t occurring in a formula in the context of a given structure M, we will use the notation t^M.

We consider formulas in *array theory* [14], [24], with one-dimensional arrays whose index sort is *Int*, as these are the ones that occur in satisfiability queries during symbolic execution of low-level program representations (e.g., LLVM IR). The theory includes the interpreted function symbols: rd, wr, and $K(c)$ (an array initialized with the constant c).

Definition IV.1. Let L_1 be the language of unquantified formulas in the array theory with two sorts for scalars: *Int* and *Ptr*. The *Int* sort admits all the linear integer arithmetic operations; the *Ptr* sort admits equality, a special constant symbol *null*, and a pointer arithmetic operator:

$$+ : \textit{Ptr} \times \textit{Int} \to \textit{Ptr}$$

The intended interpretation for *Ptr* will therefore be numeric, and we assume it to be isomorphic to $\mathbb{N}$ in the sequel, with *null* always interpreted as 0 and $+$ as integer addition. Despite that, the sorts *Ptr* and *Int cannot* be mixed freely in L_1 formulas: for example, comparing pointers with integers is prohibited.

Definition IV.2. An *address space* is a set of disjoint intervals, canonically written as $S = \{[b_i, e_i]\}_{1 \leq i \leq r}$ where $0 < b_i \leq e_i$.

Definition IV.3. Let $t := t_p + t_n$ be an L_1 term, where t_p : *Ptr* and t_n : *Int*. We say that t *respects* an address space S in a structure M if there exists an interval $[b_i, e_i] \in S$ such that $t_p^M \in [b_i, e_i]$ and $t^M \in [b_i, e_i]$. A formula $\varphi \in L_1$ *respects* an address space S in a structure M, when all its sub-terms of the form $t_p + t_n$ respect it.

This requirement is crucial to our treatment of formulas that contain pointer arithmetic operations: It means that whenever such operation occurs in φ, it may not take an address that resides inside one interval and create an address that resides in a different interval. In particular, it cannot *cross the boundaries* and reach another interval.

Definition IV.4. Two address spaces S_1, S_2 will be considered *isomorphic* if there is a bijection $f : S_1 \to S_2$ such that:

$$\forall [b_i, e_i] \in S_1. \quad \big|[b_i, e_i]\big| = \big|f([b_i, e_i])\big|$$

where $\big|[b, e]\big| = e - b$ denotes the size of the interval.

Such isomorphism induces an *address translation* function, denoted by $xt : \bigcup S_1 \to \bigcup S_2$, and defined as:

$$xt(pv) = pv + (b_i' - b_i) \quad \text{where } pv \in [b_i, e_i] \text{ and } f([b_i, e_i]) = [b_i', e_i']$$

The translation xt admits a standard extension to sequences (arrays) of addresses via pointwise application.

Definition IV.5. A formula $\varphi \in L \supseteq L_1$ is *address-agnostic* if for every two isomorphic address spaces S and S', and a structure M where φ respects S, there exists a structure M' such that:

- φ respects S' in M'
- $M \models \varphi \iff M' \models \varphi$

Lemma IV.6. *If $\varphi_1, \varphi_2 \in L \supseteq L_1$ are address-agnostic, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2$ are address-agnostic as well.*

Lemma IV.7. *A formula $\varphi \in L_1$ is address-agnostic.*

Proof. M' is obtained from M by applying the translation xt to every interpretation of a *Ptr*- or *Array[Ptr]*-sorted element in M. It can then be shown that for every subterm t of φ:

- If its sort is *Int* or *Int*[], then $t^M = t^{M'}$.
- If its sort is *Ptr* or *Ptr*[], then $xt(t^M) = t^{M'}$.

And consequently, that for every subformula ψ of φ:

$$M \models \psi \iff M' \models \psi$$

The proof is done by induction on the structure of t and ψ. This requires considering all the possible ways to construct terms and formulas in L_1, which are numerous. The only interesting case is when $t = t_1 + t_2$ (with t_1 : *Ptr*, t_2 : *Int*), where we use the xt translation together with the assumption of Definition IV.3 to show that the address computation is consistent between M and M'. □

Lemma IV.7 states that in the context of L_1, the concrete values of interval boundaries $[b_i, e_i]$ in the address space has no effect on the truth value of the formula, and they can be freely rearranged into any other locations. However, L_1 is not expressive enough for our purposes, as it allows construction of pointer values from integers but not vice versa, thus preventing the use of expressions such as $rd(a, p_1 - p_2)$, which are routinely generated by the symbolic execution engine for the representation of pointer dereference operations.

Definition IV.8. Let L_2 be the extension of L_1 with a second pointer arithmetic operation:

$$- : \mathit{Ptr} \times \mathit{Ptr} \rightarrow \mathit{Int}$$

Ptr still corresponds to a numeric domain so that the subtraction operation is meaningful. In particular, $(p+n)-p = n$ should be a valid formula. We will use $p_1 < p_2$, $p_1 \leq p_2$ as abbreviations for $p_1 - p_2 < 0$, $p_1 - p_2 \leq 0$, respectively.

In order for our formulas to still be address-agnostic, we would have to avoid expressions such as $p_2 - p_1$ (where p_1, p_2 are *Ptr* terms), since these will take different values depending on the relative positioning of the intervals containing p_1 and p_2, if they happen to reside in different address intervals.

Definition IV.9. Let a : *Ptr* be a constant symbol, n : *Int*, and p : *Ptr* be an expression from L_1. The *guard* constraint of p over a and n, denoted by $\gamma(p, a, n)$, is given by the formula:

$$a \leq p < a + n$$

Note that Definition IV.5 talks about structures and address spaces where the formula *respects* the given address space, so $a + n$ cannot cross the boundaries between intervals (objects).

Lemma IV.10. *A guard constraint is address-agnostic.*

Lemma IV.11. *Let $\gamma \wedge \psi$ be a formula in L_2, where γ is a guard constraint, i.e., $\gamma(p, a, n)$, and $\psi \in L_2$. If for every term of the form $p - p'$ in ψ there exists k : Int such that:*

- $\gamma \Rightarrow p - p' = k$
- $\psi[k/(p - p')]$ *is address-agnostic*

then $\gamma \wedge \psi$ is address-agnostic.

Proof. If there is k : *Int* such that $\gamma \Rightarrow p - p' = k$, then in particular $\gamma \wedge \psi \Leftrightarrow \gamma \wedge \psi[k/(p - p')]$. From Lemma IV.10 we know that γ is *address-agnostic*, and we assumed that $\psi[k/(p - p')]$ is *address-agnostic*, so using Lemma IV.6 we conclude that their conjunction is *address-agnostic* as well. □

The only sources of pointer subtraction terms are subtraction statements originating from the program and representations of pointer dereferences. We assume that the program does not subtract pointers that correspond to different objects (according to the C standard), so under this assumption we can formulate the following theorem:

Theorem IV.12. *The constraints generated by the symbolic execution engine are address-agnostic.*

Proof. The proof is done by induction on the size of the path constraints. The base case is trivial, i.e., $PC = true$. For the induction step, we assume that PC is *address-agnostic*, and need to prove that $PC \wedge \varphi$ is *address-agnostic* as well.

When a pointer p is dereferenced, the engine checks if p may point to a memory object $mo = (\beta, s, a)$ using the following resolution query:

$$PC \ \wedge \ 0 \leq p - \beta < s$$

If this formula is satisfiable, i.e., p is resolved to mo, then the accessed value is expressed using $rd(a, p - \beta)$.

The main case to handle is when φ originates from a branch condition: If $\varphi \in L_1$, then $PC \wedge \varphi$ is *address-agnostic* according to Lemma IV.6. Otherwise, φ contains a subtraction term $p - \beta$ that was generated for representing a pointer dereference (the case where it originates from a program statement is easy to handle). According to the previous paragraph, PC must contain the *guard* constraint:

$$\gamma := 0 \leq p - \beta < s$$

and since the number of clauses in PC is finite, we can assume that $p \in L_1$. There clearly exists k : *Int* such that $\gamma \Rightarrow p - \beta = k$, so in order to apply Lemma IV.11 we need to make sure that $PC' \wedge \varphi[k/(p - \beta)]$ is *address-agnostic* (where $PC := \gamma \wedge PC'$). Let $\varphi' := \varphi[k/(p - \beta)]$, if $\varphi' \in L_1$ then we can apply Lemma IV.11 to conclude that $\gamma \wedge \varphi$ is *address-agnostic*, and then $PC \wedge \varphi = (\gamma \wedge PC') \wedge (\gamma \wedge \varphi)$ is *address-agnostic* according to Lemma IV.6. Otherwise, $\varphi' \in L_2$ and we can apply again the same substitution steps as before, until all the pointer subtraction terms in φ are substituted. □

As an example for the application of Lemma IV.11 consider again the program from Figure 1. The access of `array[i][j]` triggers a dereference of the symbolic pointer $p = rd(a_1, i) + j$,

where $a_1 = wr(wr(K(0), 0, \beta_2), 1, \beta_3)$. This pointer is resolved to $mo_2 = (\beta_2, 2, a_2)$ using the following query:

$$PC := 0 \leq i < 2 \ \wedge \ 0 \leq j < 2 \ \wedge \ \beta_2 \leq p < \beta_2 + 2$$

Here, γ (the *guard* constraint) is given by $\beta_2 \leq p < \beta_2 + 2$, and ψ is the rest of the formula. Since ψ is in L_1, it easily follows from Lemma IV.11 that PC is *address-agnostic*.

After the resolution, the query generated for the branch at line 17 is given by:

$$PC' := PC \ \wedge \ rd(a_2, p - \beta_2) = 7$$

Here again, our γ will be $\beta_2 \leq p < \beta_2 + 2$, and ψ is the rest of the formula. It clearly holds for $k = j$ that $\gamma \Rightarrow p - \beta_2 = k$, and the substitution $\psi[k/(p - \beta_2)]$ results in:

$$0 \leq i < 2 \ \wedge \ 0 \leq j < 2 \ \wedge \ rd(a_2, k) = 7$$

which is a formula in L_1 (and therefore *address-agnostic*), so Lemma IV.11 can be applied again.

Remark. Our correctness arguments hold also for formulas where a pointer is allowed to refer to multiple objects, as happens in tools that apply state merging [44], [26], since the *address-agnostic* property is preserved under disjunction. The notion of the *red zone*, discussed in Section II, can be easily incorporated into our correctness arguments and requires no modifications to the proof: It is simply sufficient to consider each interval $[b_i, e_i]$ to consist of the padding block and the address block allocated for the respective memory object. Since the size of the padding is constant for all allocations, the resulting address spaces will still be isomorphic.

V. Implementation

We implemented our query caching approach on top of the state of the art symbolic executor KLEE [16], configured with the STP [24] solver. Similarly to [45], we modified the engine's allocator to return the allocated base addresses as symbolic values, rather than concrete values. Consequently, the expressions (and queries) constructed by the engine do not contain any concrete address values, i.e., integers. In addition, we extended the symbolic state with the *address constraints*, which are used to substitute the address values in the constraints before they are passed to the SMT solver (Section III-A).

In KLEE, the existing query cache is implemented using a hash table of queries. To make the lookup and insert operations efficient, a hash value is maintained for each query (and each expression), which is then used as a key for that hash table. Then, the bucket retrieved with that key is scanned to find the matching query based on *syntactic equality*. To enable efficient caching for our query caching approach as well, we maintain additional hash value for each expression which captures its structure regardless of the symbolic base address expressions. More technically, when we compute this hash value, we set the hash value of each symbolic base address to a pre-defined constant. For example, the following queries will have the same hash value:

$$rd(wr(a, 1, \beta_2)) = 0 \qquad rd(wr(a, 1, \beta_3)) = 0$$

As discussed in Section III, our approach does not apply to queries whose satisfiability depends on the ordering of the memory objects in the address space. When such queries are not *engine-internal*, i.e., generated as a result of a branch in the program, our approach may lead to incorrect results. Automatically detecting such queries is not straightforward, and we leave it for future research.

VI. Evaluation

In our experiments, we evaluate our address-aware query caching approach (AA) against the standard approach used in vanilla KLEE (Base). The challenge of caching address-dependent queries was partially and heuristically addressed in the work that introduced the *dynamically segmented* memory model, using a *segment reuse* heuristic (see paragraph *Reusing Segments*, Section 2.3.3 from [45]): There, when a new segment is dynamically created, it attempts to reuse previously allocated base addresses, in order to reduce the chance for cache misses. Therefore, we evaluate our approach under two memory models: The *forking* memory model (FMM), i.e., vanilla KLEE, and the *dynamically segmented* memory model (DSMM), introduced by [45]. When running under DSMM, we evaluate our approach (while disabling the *segment reuse* heuristic) against DSMM when that heuristic is enabled. An evaluation with different memory models also demonstrates the robustness and applicability of our approach.

Our evaluation is structured as follows: In section VI-A we present our benchmarks. In section VI-B we provide an empirical validation for our approach. In Section VI-C we show the effectiveness of our approach on benchmarks which generate address-dependent queries. In Section VI-D we measure the overhead of our approach on benchmarks that do *not* generate address-dependent queries, where our approach is not expected to produce speedups. Our replication package is available at https://doi.org/10.6084/m9.figshare.13042277.

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

A. Benchmarks

Our experiments used the following code bases as benchmarks: *GNU m4* [5] (80K SLOC) is a macro processor included in many Unix-based systems. *GNU make* [6] (28K SLOC) is a tool which controls the generation of executables and other non-source files, also widely used in Unix-based systems. *SQLite* [11] (127K SLOC) is one of the most popular SQL database libraries. *Apache Portable Runtime* [1] (60K SLOC) is a library used by the Apache HTTP server that provides cross-platform functionality for memory allocation, file operations, containers, and networking. The *libxml2* [9] (197K SLOC) library is a XML parser and toolkit developed for the Gnome project. The *expat* [2] (23K SLOC) library is a stream-oriented XML parser, used in many open-source projects including Mozilla, Perl, Python and PHP. *GNU bash* [3] (106K SLOC) is the well-known Unix shell written for the GNU project. The *json-c* [8] (7K SLOC) library is used for encoding and

decoding JSON objects. *GNU Coreutils* [4] (188K SLOC) is a collection of utilities for file, text, and shell manipulation. The *libosip* [7] (11K SLOC) library is used for parsing SIP messages. The *libyaml* [10] (9K SLOC) library is used for parsing and emitting data in the YAML format.

In Section VI-C we evaluate our approach on a set of terminating programs that generate address-dependent queries. Such queries are typically generated in the presence of symbolic pointers, which are created, for example, when data structures such as hash tables are indexed using a symbolic value as key. Our benchmarks consist of both whole-program utilities (*m4*, *make*, *bash*) and libraries (*sqlite*, *apr*, *libxml2*, *expat* and *json-c*). Four of our benchmarks (*m4*, *make*, *sqlite*, and *apr*) were used in previous work related to symbolic pointers [30], [45]: In *m4* and *make*, which are language-processing utilities, hash tables are used to store the values of variables, functions, and strings. To avoid the analysis of these programs from getting stuck in the early stages and to achieve its termination, these programs are run with a partially symbolic input. The driver in *apr* focuses on the runtime's hash table API, and the driver in *sqlite* creates database triggers using concrete and symbolic SQL queries. Our four additional benchmarks are *libxml2*, *expat*, *bash*, and *json-c*: As was done in the cases of *m4* and *make*, we ran *bash* with a partially symbolic input in order to reach the deeper parts of the code that operate on the various tables that store variables, strings, and functions. In *libxml2* and *expat* we built drivers that parse symbolic HTML and XML inputs, respectively. In *json-c* we built a driver that constructs a JSON object which internally uses hash tables.

In Section VI-D we evaluate our approach on a set of programs that don't generate address-dependent queries: We chose 10 utilities from *coreutils* that behave deterministically across multiple runs, and built drivers for the main parsing API's in both *libosip* and *libyaml*.

B. Empirical Validation

In this experiment, we provide an empirical validation for the correctness of our approach using the following methodology: We ran KLEE on each of the benchmarks with the two approaches (Base and AA), and as our approach must not affect the exploration, we validated that both the number of explored paths and the achieved coverage are indeed identical in both runs. When running with our approach, we additionally checked for each cached query that its cached result is correct by simply comparing it with the result reported by the SMT solver. We performed this experiment using the two memory models (FMM and DSMM).

C. Performance

In this experiment, we compare the performance of the two approaches (Base and AA). KLEE in its default configuration uses two constraint solving heuristics: the existing query caching and the counter-example (CEX) caching. Therefore, in order to have a complete comparison against vanilla KLEE, we enable the CEX caching in our experiments as well. For each approach, we run KLEE with the DFS search heuristic and the deterministic memory allocator, until all the paths are explored. In each run we record the following parameters: the number of queries reaching the SMT solver, the termination time (i.e., analysis time), the size of the query cache, and the memory usage.

TABLE II: Classification of queries and their amounts.

Program	Total	C_1	C_2	C_3
m4	14,022	9,589	9,127	6,394
make	2,565,399	2,477,145	90,027	69,535
sqlite	26,990	18,407	15,589	13,783
apr	15,013	8,960	8,960	8,448
libxml2	708,101	410,789	347,420	347,420
expat	1,797,033	945,192	102,903	102,903
bash	54,078	19,051	10,840	7,860
json-c	28,476	17,484	17,263	14,311

TABLE III: Number of queries with both approaches.

Program	FMM		DSMM	
	Base	AA	Base	AA
m4	10,792	4,265	1,600	1,289
make	347,324	45,471	50,558	9,753
sqlite	5,622	4,681	14,563	12,993
apr	445	300	126	86
libxml2	124,782	6,118	124,782	6,118
expat	89,740	31,747	89,736	31,761
bash	8,538	4,479	7,542	4,098
json-c	15,364	5,246	2,757	1,523

To get an insight into the type of queries encountered in our benchmarks, we analyze in Table II the queries generated by vanilla KLEE: In KLEE, the query caching heuristic handles only *satisfiability* and *validity* queries, and does not handle *model* (assignment) queries. As discussed in Section III, not all the queries passed to the query caching heuristic can be handled with our approach. Column *Total* shows the total number of queries generated during the analysis. Column C_1 shows the number of queries passed to the query caching heuristic. Column C_2 shows the number of queries passed to the query caching heuristic which can be handled by our approach. Column C_3 shows the number of queries passed to the query caching heuristic which can be handled by our approach and are address-dependent. Note that the number of queries that pass through the cache but cannot be handled by our approach, i.e., $C_2 - C_3$, is relatively low.

Table III shows the number of queries for each benchmark with the two approaches and the different memory models. Here, we report the number of queries that reached the SMT solver itself, that is, those that were not handled by any of the constraint solving heuristics (query caching or CEX caching). In FMM, the reduction in the number of queries with our approach varies between 1.20× (in *sqlite*) and 20.40× (in *libxml2*), and its average is 5.11×. In DSMM, the reduction varies between 1.12× (in *sqlite*) and 20.40× (in *libxml2*), and its average is 4.48×.

TABLE IV: Termination time in *hh:mm:ss*.

Program	FMM		DSMM	
	Base	**AA**	**Base**	**AA**
m4	00:13:16	00:04:59	00:19:17	00:14:55
make	06:46:44	02:30:51	03:56:42	01:47:23
sqlite	00:17:20	00:14:24	04:00:17	03:12:22
apr	00:57:33	00:39:05	00:20:20	00:13:39
libxml2	02:33:33	00:17:09	02:27:35	00:17:12
expat	00:26:02	00:23:19	00:25:13	00:23:06
bash	02:37:48	01:23:30	02:39:04	01:14:18
json-c	00:31:36	00:13:20	00:08:05	00:04:19

TABLE V: The size of the query cache with both approaches.

Program	FMM		DSMM	
	Base	AA	Base	AA
m4	11,780	3,493	1,631	1,341
make	348,210	41,927	51,064	12,404
sqlite	6,898	5,354	10,680	9,071
apr	496	279	130	81
libxml2	136,165	5,517	136,165	5,517
expat	92,383	34,226	92,382	34,226
bash	8,774	4,453	7,712	4,308
json-c	15,998	3,634	2,906	1,336

Table IV shows the termination time for each benchmark with the two approaches and the different memory models. In FMM, the speedup relatively to the existing query caching varies between 1.11× (in *expat*) and 8.96× (in *libxml2*), and its average is 2.80×. Note that the speedup depends not only on the reduction in the number of queries, but also on the complexity of the queries. For example, the reduction in the number of queries in *make* is roughly 4 times higher than in *bash* (7.63× *vs.* 1.90×), but the speedup in *make* is only 1.43× higher than in *bash* (2.70× vs 1.90×) as the queries in *bash* are more complex due to larger SMT arrays, that is, array constraints with more *write* expressions. In DSMM, the speedup varies between 1.09× (in *expat*) and 8.59× (in *libxml2*), and its average is 2.73×. The queries in *expat* are relatively simple compared to other benchmarks, therefore the performance gains are less significant in that case.

Table V shows the size of the query cache for each benchmark with the two approaches and the different memory models. In FMM, the reduction in the cache size varies between 1.29× (in *sqlite*) and 24.68× (in *libxml2*), and its average is 6.06×. In DSMM, the reduction varies between 1.18× (in *sqlite*) and 24.68× (in *libxml2*), and its average is 4.93×.

In general, the number of explored paths in DSMM is guaranteed to be at most as high as in FMM, and lower in programs whose analysis trigger symbolic pointers with multiple resolutions. Therefore, in such programs there is also a reduction in the number of queries and the cache size. However, in DSMM the queries are potentially more complex, so the termination time is not necessarily lower compared to FMM, as was shown in [45] and as can be seen in Table IV. The number of explored paths in *libxml2* and *expat* is identical with FMM and DSMM, but in *expat* there is a slight difference in the number of queries (and cache size) due to non-determinism introduced by the SMT solver in model (assignment) queries.

Figure 4 shows the memory usage for each of the benchmarks with the two approaches under the different memory models. Clearly, the size of the query cache affects the memory usage, that is, a smaller cache should result in lower memory usage. In benchmarks where the cache size is relatively low (*m4*, *sqlite*, *apr*, *bash* and *json-c*), the reduction in the cache size has little effect on the memory usage, which is roughly the same with both approaches. However, in other benchmarks where the cache is larger, the difference in the cache size results in larger

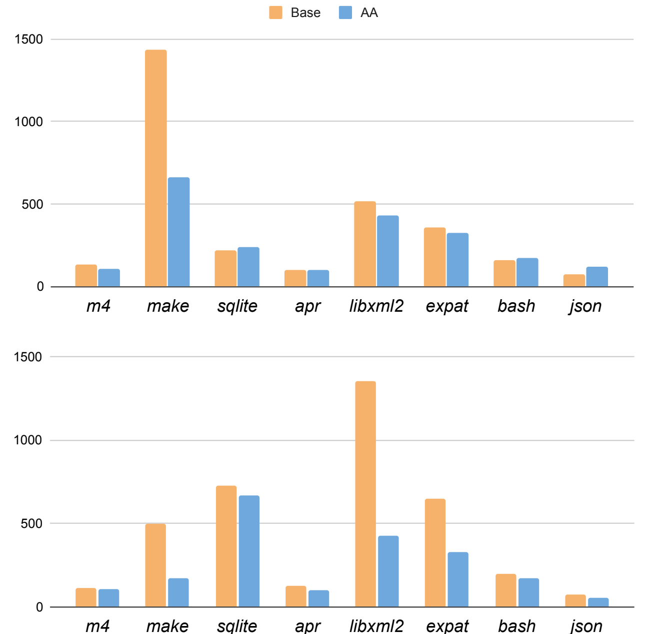

Fig. 4: Memory usage (in *MB*) with both approaches under FMM (top) and (DSMM) (bottom).

difference in memory usage: For example, when FMM is used, the memory usage in *make* and *libxml2* is reduced by roughly 800MB and 100MB, respectively.

The overall performance improvement with our approach suggests that there are queries that are handled by our approach and aren't handled by the CEX caching [4]. Therefore the CEX caching should be seen as complementary to our approach.

D. Overhead

In Section VI-C we showed how our approach can improve the performance for programs which generate address-dependent queries. However, our approach imposes additional computational overhead due to two main reasons: maintaining additional symbolic values for address expressions and substituting expressions (Section III-A).

[4] We internally experimented also with an optimized version of the CEX caching heuristic (using the *cex-cache-try-all* option), which resulted in even better improvement for our approach.

In this experiment, we show the runtime overhead of our approach for programs that do not generate address-dependent queries: *coreutils*, *libosip* and *libyaml*. For each program, we proceed with the following methodology under each of the memory models: First, we ran KLEE for roughly one hour, and recorded the number of executed instructions for each program. Then we ran each program up to the recorded number of instructions with both of the approaches (Base and AA).

In FMM, with regards to termination time, our approach had a maximum overhead of 17% (in *libosip*) and an average overhead of 6%. There was no significant difference in memory usage between the two approaches, with our approach having an overhead of 3%. Similarly to our query caching approach, DSMM relies on the symbolic addressing model as well, which is the main source of overhead compared to vanilla KLEE (FMM). Therefore in DSMM, where the symbolic addressing model is used in both of the approaches, the performance is almost identical in terms of time and space.

Similarly to Section VI-C, we validated that the number of explored paths is identical with both approaches, and performed an additional run for each program to validate the correctness of our query caching approach with respect to the SMT solver.

VII. Discussion and Related Work

We presented our approach assuming that symbolic reads and writes are encoded using array theory. However, our approach does not specifically relies on such encoding and will work with other encodings, such as *if-then-else* (disjunction) expressions (e.g., ANGR [44]). Moreover, our approach is agnostic to the underlying memory allocation scheme, and can work with a linear deterministic scheme as well as other schemes that are based on system API's (e.g., *malloc*). Therefore, we believe that our approach can be applied to other tools that generate address-dependent queries (e.g., ANGR and Manticore).

The idea of improving constraint solving by reusing previously solved results has been investigated in the past: KLEE [16] uses counter-example caching, which stores results into a cache that maps constraint sets to concrete variable assignments. Using these mappings, KLEE can solve several types of similar queries, involving subsets and supersets of the constraint sets already cached. Green [47] is a framework that enables reusing constraints results within a single run as well as across different runs and programs. To enable efficient reuse, the technique uses slicing to reduce the complexity of the constraints, and canonization to store the constraints in a normal form. GreenTrie [28] is an extension of Green that detects implications between constraints to improve caching for satisfiability queries. Cashew [15] is a caching framework for model-counting queries built on top of Green [47], which introduces an aggressive normalization scheme and parameterized caching. Eiers et al. [22] use subformula caching to improve the performance of model counting constraint solvers in the context of quantitive program analysis. These approaches do not solve the problem presented in this paper, as they fail to detect equisatisfiable address-dependent queries when they are syntactically distinct, due to address constants.

Other approaches have been proposed to scale constraint solving in the context of symbolic execution: splitting constraints into independent sets [17], [16], multiple solvers support [37], interval-based solving [20], and fuzzing-based solving [32], [38]. Perry et al [40] focus on reducing the cost of array theory constraints using several semantics-preserving transformations. These transformations attempt to eliminate array constraints as much as possible by replacing them with constraints over their indices and values. Modern SMT solvers such as CVC4 [13], Z3 [18], and Yices [21], have support for *incremental solving*, which enables learning lemmas that can be later reused for solving similar queries. These approaches are orthogonal to our approach, with which they could be combined.

State merging [41], [31], [12], [43] enables fusing multiple states into a single state, thus potentially increasing the chance for query cache hits. In contrast to state merging, our approach does not introduce *if-then-else* expressions, thus avoiding the negative effect on the SMT solver. In addition, our approach is control-independent, and therefore can reuse query results even when state merging is not applicable.

Segmented memory model (SMM) [30] is a technique for handling symbolic pointers that have multiple resolutions: The memory is partitioned into segments using static pointer analysis, such that each pointer refers to a single segment, thus avoiding forks when symbolic pointers are dereferenced. Similarly to DSMM [45], this approach mainly focuses on mitigating path explosion, rather than on optimizing constraint solving. SMM does not provide a solution to our problem for basically the same reason that DSMM does not: The objects can still be allocated in different segments, or allocated in different offsets within the same segment, which would result in the same problem as happens when base addresses differ. Also note that the challenge of caching address-dependent queries exists not only when symbolic pointers have multiple resolutions, but also when they are resolved to a single object.

VIII. Conclusion and Future Work

We proposed a novel query caching approach in the context of symbolic execution, which addresses the challenge of caching address-dependent queries. We formally discussed the correctness guarantees of our approach, and provided an additional empirical validation. We demonstrated in our experiments that our approach can lead to order-of-magnitude speedup on programs with address-dependent queries, while maintaining a relatively low overhead in other cases.

Our approach cannot be applied to queries which are not address-agnostic. Therefore, coming up with a scheme that can handle such queries can further improve performance, especially in programs where symbolic pointer resolution is expensive. Our approach can handle *satisfiability* and *validity* queries, but *model* queries remain unhandled. Applying a similar approach for such queries, including address-dependent ones, is another research direction.

Acknowledgements. The research leading to these results has received funding from the Pazy Foundation and the Israel Science Foundation (ISF) grant No. 1996/18.

References

[1] Apache Portable Runtime. https://apr.apache.org/.

[2] expat. https://libexpat.github.io.

[3] GNU Bash. https://www.gnu.org/software/bash.

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

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

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

[7] GNU oSIP. https://www.gnu.org/software/osip.

[8] json-c. https://github.com/json-c/json-c.

[9] libxml2. http://www.xmlsoft.org.

[10] libyaml. https://github.com/yaml/libyaml.

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

[12] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proc. of the 36th International Conference on Software Engineering (ICSE'14)*, May 2014.

[13] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proc. of the 23rd International Conference on Computer-Aided Verification (CAV'11)*, Jul. 2011.

[14] A. R. Bradley, Z. Manna, and H. B. Sipma, "What's decidable about arrays?" in *Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 427–442.

[15] T. Brennan, N. Tsiskaridze, N. Rosner, A. Aydin, and T. Bultan, "Constraint normalization and parameterized caching for quantitative program analysis," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 535–546.

[16] C. Cadar, D. Dunbar, and D. Engler, "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)*, Dec. 2008.

[17] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically Generating Inputs of Death," in *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, Oct.-Nov. 2006.

[18] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[19] L. De Moura and N. Bjorner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[20] O. S. Dustmann, K. Wehrle, and C. Cadar, "Parti: a multi-interval theory solver for symbolic execution." 2018.

[21] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[22] W. Eiers, S. Saha, T. Brennan, and T. Bultan, "Subformula caching for model counting and quantitative program analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 453–464.

[23] B. Elkarablieh, P. Godefroid, and M. Y. Levin, "Precise pointer reasoning for dynamic test generation," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)*, Jul. 2009.

[24] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 519–531. [Online]. Available: http://dl.acm.org/citation.cfm?id=1770351.1770421

[25] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux kernel extensions," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, p. 1069–1084.

[26] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[27] M. Hind, "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)*, Jun. 2001.

[28] X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)*, Jul. 2015.

[29] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, Jun. 2012.

[30] T. Kapus and C. Cadar, "A segmented memory model for symbolic execution," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. ACM, 2019, pp. 774–784.

[31] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'12)*, Jun. 2012.

[32] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: solving floating-point constraints using coverage-guided fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 521–532.

[33] P. D. Marinescu and C. Cadar, "KATCH: High-coverage testing of software patches," in *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[34] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 691–701.

[35] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[36] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[37] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *Proc. of the 25th International Conference on Computer-Aided Verification (CAV'13)*, Jul. 2013.

[38] A. Pandey, P. R. G. Kotcharlakota, and S. Roy, "Deferred concretization in symbolic execution via fuzzing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 228–238. [Online]. Available: https://doi.org/10.1145/3293882.3330554

[39] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis," Sep. 2013.

[40] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'17)*, Jul. 2017.

[41] D. Qi, H. D. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," 2011.

[42] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proc. of the 24th USENIX Security Symposium (USENIX Security'15)*, Aug. 2015.

[43] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. ACM, 2015, aCM SIGSOFT Distinguished Paper Award.

[44] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "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)*, May 2016.

[45] D. Trabish and N. Rinetzky, "Relocatable addressing model for symbolic execution," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 51–62. [Online]. Available: https://doi.org/10.1145/3395363.3397363

[46] A. Venet and G. Brat, "Precise and efficient static array bound checking for large embedded c programs," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004, p. 231–242.

[47] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: reducing, reusing and recycling constraints in program analysis," in *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12)*, Nov. 2012.