EPiC Series in Computing
Volume 57, 2018, Pages 363–380
LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing

Benjamin Farinier[1], Robin David[2], Sébastien Bardin[1], and Matthieu Lemerre[1]

[1] CEA, LIST, Software Safety and Security Lab, Université Paris-Saclay, France
firstname.lastname@cea.fr
[2] Quarkslab, Paris, France
rdavid@quarkslab.com

Abstract

The theory of arrays has a central place in software verification due to its ability to model memory or data structures. Yet, this theory is known to be hard to solve in both theory and practice, especially in the case of very long formulas coming from unrolling-based verification methods. Standard simplification techniques *à la read-over-write* suffer from two main drawbacks: they do not scale on very long sequences of stores and they miss many simplification opportunities because of a crude syntactic (dis-)equality reasoning. We propose a new approach to array formula simplification based on a new dedicated data structure together with original simplifications and low-cost reasoning. The technique is efficient, scalable and it yields significant simplification. The impact on formula resolution is always positive, and it can be dramatic on some specific classes of problems of interest, e.g. very long formula or binary-level symbolic execution. While currently implemented as a preprocessing, the approach would benefit from a deeper integration in an array solver.

1 Introduction

Context. Automatic decision procedures for Satisfiability Modulo Theory [4] are at the heart of almost all recent formal verification methods [11, 6, 10, 26]. Especially, the *theory of arrays* enjoys a central position in software verification as it allows to model memory or essential data structures such as maps, vectors and hash tables.

Intuitively, given a set $\mathcal{I}$ of indexes and a set $\mathcal{E}$ of elements, the theory of arrays describes the set Array $\mathcal{I}$ $\mathcal{E}$ of all arrays mapping each index $i \in \mathcal{I}$ to an element $e \in \mathcal{E}$. Actually, logical arrays can be seen as infinite updatable maps implicitly defined by a succession of writes from an initial map. These arrays are defined by the two operations read (select) and write (store), whose semantic is given in Fig. 1 by so-called read-over-write axioms (ROW-axioms).

Despite its simplicity, the satisfiability problem for the theory of arrays is NP-complete[1]. Indeed, it implies deciding (dis-)equalities between read and written indexes on *read-over-write*

[1]Reduction of the program equivalence problem in presence of arrays (sequential, boolean case) to the equivalence case without arrays but with if-then-else operators, to SAT [17].

$$\begin{array}{rlr} \mathsf{select} : & \text{Array } \mathcal{I}\ \mathcal{E} \to \mathcal{I} \to \mathcal{E} & \forall a\, i\, e.\, \mathsf{select}\,(\mathsf{store}\, a\, i\, e)\, i = e \\ \mathsf{store} : & \text{Array } \mathcal{I}\ \mathcal{E} \to \mathcal{I} \to \mathcal{E} \to \text{Array } \mathcal{I}\ \mathcal{E} & \forall a\, i\, j\, e.\, (i \neq j) \Rightarrow \mathsf{select}\,(\mathsf{store}\, a\, i\, e)\, j = \mathsf{select}\, a\, j \end{array}$$

Figure 1: The theory of arrays (ROW-axioms)

terms (ROW) of the form $\mathsf{select}\,(\mathsf{store}\, a\, i\, e)\, j$, potentially yielding nested case-splits. Standard decision procedures for arrays consist in eliminating as much ROW as possible through a preprocessing step [22], using axioms from Fig. 1 as rewriting rules, and then enumerating all possible (dis-)equalities in ROW, yielding a potentially huge search space — the remaining ROW-axioms can be introduced lazily to mitigate this issue [9].

Problem and challenge. Yet, this is not satisfactory when considering very long chains of writes, as can be encountered in unfolding-based verification techniques such as Symbolic Execution (SE) [10] or Bounded Model checking [11] — the case of Deductive Verification is different since user-defined invariants prevent the unfolding. The theory of arrays can then quickly become a bottleneck of constraint solving. Especially, the ROW-simplification step is often very limited, for two reasons. **First**, exploring for every read in a backward manner the corresponding list of all writes yields a *quadratic time cost* (in the number of array operations) and therefore *it does not scale to very long formulas.* This is a major issue in practice as, for example, Symbolic Execution over malware or obfuscated programs [27, 1, 29] may have to consider execution traces of several millions of instructions, yielding formulas with several hundreds of thousands of array operations. Note also that bounding the backward exploration misses too many ROW-simplifications. **Second**, *(dis)-equalities can be rarely decided during preprocessing* as standard methods rely on efficient but *crude approximate equality checks* (typically, syntactic term equality), limiting again the power of these approaches. With such checks, index equality may be sometimes proven, but disequality can never be — except in the very restricted case of constant-value indexes.

Our proposal. We present a novel approach to ROW-simplification named FAS (*Fast Array Simplification*), allowing to scale and to simplify much more ROW than previous approaches. The technique is based on three key components:

- A re-encoding of write sequences (total order) as sequences of packs of independent writes (partial order), together with a dedicated data structure (*map list*) ensuring scalability;
- A new simple normalization step (*base normalization*) allowing to amplify the efficiency of syntactic (dis-)equality checks;
- A lightweight integration of *domain-based reasoning* over packs yielding even more successful (dis-)equality checks for only a slight overhead.

Experimental results demonstrate that FAS scales over very large formulas (several hundreds of thousands of ROW) typically coming from Symbolic Execution and can yield very significant gains in terms of runtime — possibly passing from hours to seconds.

Contribution. Our contribution is two-fold:

- We present in detail the new FAS preprocessing step for scalable and thorough array constraint simplification (Sec. 4), along with its three key components: dedicated data structure (Sec. 4.1), base normalization (Sec. 4.2) and domain reasoning (Sec. 4.4);
- We experimentally evaluate FAS in different settings for three leading SMT solvers (Sec. 5). The technique is fast and scalable, it yields a significant reduction of the number of ROW with always a positive impact on resolution time. This impact is even dramatic for some key usage scenarios such as SE-like formulas with small TIMEOUT or very large size.

Discussion. In our view, FAS reaches a sweet spot between efficiency and impact on resolution. Experiments demonstrate that even major solvers benefit from it, with gains ranging from slight to very high depending on the setting. While presented as a preprocessing, FAS would clearly benefit from a deeper integration inside an array solver, in order to take advantage of more simplification opportunities along the resolution process.

2 Motivation

$$\begin{array}{l|l}
\texttt{esp}_0\ :\ \texttt{BitVec16} & \texttt{esp}_0\ :\ \texttt{BitVec16} \\
\texttt{mem}_0\ :\ \texttt{Array BitVec16 BitVec16} & \texttt{mem}_0\ :\ \texttt{Array BitVec16 BitVec16} \\
 & \\
\texttt{assert}\,(\texttt{esp}_0 > 61440) & \texttt{assert}\,(\texttt{esp}_0 > 61440) \\
\texttt{mem}_1 \triangleq \texttt{store mem}_0\,(\texttt{esp}_0 - 16)\,1415 & \texttt{assert}\,(\texttt{select mem}_0\ 1415 = 9265) \\
\texttt{esp}_1 \triangleq \texttt{esp}_0 - 64 & \\
\texttt{eax}_0 \triangleq \texttt{select mem}_1\,(\texttt{esp}_1 + 48) & \\
\texttt{assert}\,(\texttt{select mem}_1\ \texttt{eax}_0 = 9265) &
\end{array}$$

Figure 2: A formula in the theory of arrays (left) and its simplification with FAS (right)

Let us detail how the formula in the left part of Fig. 2 can be simplified into the formula in the right part using our new FAS simplification procedure for arrays. We focus on the last assertion which involves a read on $\texttt{mem}_1 \triangleq \texttt{store mem}_0\,(\texttt{esp}_0 - 16)\,1415$, i.e. a *read-over-write*. Let us denote $i \triangleq \texttt{esp}_0 - 16$. The read occurs at index $\texttt{eax}_0$, which is itself the result of a read on $\texttt{mem}_1$ at index $j \triangleq \texttt{esp}_1 + 48$. According to arrays semantics (Fig. 1), we must try to decide whether i and j (resp. $\texttt{eax}_0$ and i) are equal or different. *The standard syntactic equality check is not conclusive here.* But $\texttt{esp}_1 \triangleq \texttt{esp}_0 - 64$, therefore j can be rewritten into $\texttt{esp}_0 - 16$ (*base normalization* in FAS), which is exactly i. Hence $i = j$ is proven. By applying array axioms, we deduce that $\texttt{eax}_0 \triangleq 1415$, and the last assertion becomes $\texttt{select mem}_1\ 1415 = 9265$. We now try to decide whether i and 1415 are equal or different. *Again, the standard syntactic equality check fails.* Yet, by the first assertion we deduce that $i > 61424$ (*domain propagation* in FAS), leading to $i \neq 1415$. Therefore $\texttt{mem}_1$ is safely replaced by $\texttt{mem}_0$ in the last assertion which becomes $\texttt{select mem}_0\ 1415 = 9265$. Finally, as assertions in the formula now only refer to $\texttt{esp}_0$ and $\texttt{mem}_0$, we erase all the intermediate definitions to obtain the simplified formula.

This little mental gymnastic emphasises two important aspects of ROW-simplifications. First, simplifications often require (dis-)equality reasoning beyond pure syntactic equality. Second, simplifications involve a backward reasoning through the formula which may become prohibitive on large formulas if not treated with care (not shown here, up to 1h simplification time in Fig. 10). *Our proposal focuses especially on these two aspects.*

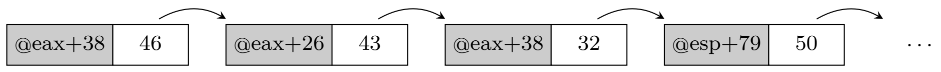

Figure 3: Arrays represented as sequences (lists) of writes

3 Background

The theory of arrays has been introduced in Fig. 1 (Sec. 1). As already stated, the main difficulty for reasoning over arrays comes from terms of the form $\mathsf{select}\,(\mathsf{store}\,a\,i\,e)\,j$, called *read-over-write* (ROW), since depending on whether $i = j$ holds or not, the term evaluates to e (SELECT-HIT) or to $\mathsf{select}\,a\,j$ (SELECT-MISS). *Array (formula) simplification* consists in removing as many ROW as possible before resolution by proving (when possible) the validity of the (dis-) equality of such pairs of indexes (i, j) and rewrite the term accordingly. Such simplification procedures critically depend on two factors: 1. the **equality check procedure**, and 2. the **underlying representation of an array** and its revisions arising from successive writes.

The **equality check** must be both *efficient* — simplifying a formula must be cheaper than solving it, and *correct* — all proven (dis-)equalities must indeed hold. It can thus only be *approximated*, i.e. it is incomplete and may miss some valid (dis-)equalities. The standard solution is to rely on *syntactic term equality checking*. Obviously this is a *crude approximation*: disequality can never be proven (but for constant-value indexes), and as exemplified in Sec. 2, small syntactic variations of the same value can hinder proving equalities.

We present now two (unsatisfactory) standard **array representations**, coming either from the decision procedure community (the *list* representation: *generic but slow*) or from the symbolic execution community (the *map* representation: *efficient but restricted*).

Arrays represented as lists. The standard representation of an array and its subsequent revisions is basically a "store-chain", the linked list of all successive writes in the array. Hence a fresh array is simply an empty list, while the array obtained by writing an element e at index i in array A is represented by a node containing (i, e) and pointing to the list representing A. Fig. 3 illustrates this encoding. This approach is very generic — it can cope with symbolic indexes, and it is the one implicitly used inside array solvers. In order to simplify a read at index j on array A, one must decide whether $i = j$ is valid for the pair (i, e) inside the head of the list representing A. If we succeed, then we can apply the ROW axiom and replace the read by value e. Otherwise, we try to decide whether $i \neq j$ is valid. If this is the case, then we use the second ROW axiom and move backward along the linked list. If not, the simplification process stops.

An inherent problem with this representation is the increase in the simplification cost as the number of writes rises. As mentioned in Sec. 2, this cost becomes prohibitive when dealing with large formulas. Indeed, one might be forced for each read to fully explore the write-list backward, yielding a *quadratic worst case time cost.* This is especially unfortunate because this worst case arises in situations where the simplification could perform the best, e.g. when all disequalities between indexes hold so that all reads could be replaced with accesses to the initial array (no more ROW). A workaround is to bound the backward exploration of the write-list, which reduces the worst case time cost to linear, but at the expense of limited simplifications (Fig. 10, Sec. 5.4).

Arrays represented as maps. In the restricted case where all indexes of reads and writes are constant values, a persistent map with logarithmic lookup and insertion can be used to simplify all ROW occurrences — yielding fast and scalable simplification. This representation is used in symbolic execution tools [10] with strong concretization policy [23, 14] during the formula generation step in order to limit the introduction of arrays, but it is not suited to general purpose array solvers as *it cannot cope with symbolic indexes*.

Here, a freshly declared array is represented by an empty map where indexes and elements sorts correspond to those of the array, and the array obtained after a write of element e at index i is simply represented by the map of the written array in which e is added at index i, as illustrated in Fig. 4. Then the simplification of a read at index j becomes its substitution by the element mapped to j. In the case where no such element is found, the read occurs on the initial array. Therefore, we can either replace the array by the initial one or replace the read by a fresh symbol. In the latter case, we have to ensure that two reads are replaced by the same symbol if and only if they occur at the same index.

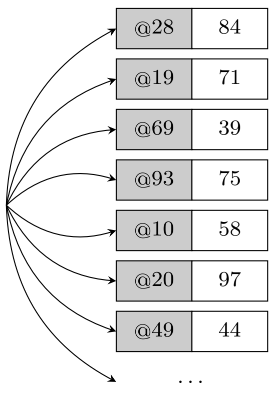

Figure 4: Arrays represented as maps of writes (constant write indexes)

4 Efficient simplification for *read-over-write*

We now present FAS (*Fast Array Simplification*), an efficient approach to *read-over-write* simplification. FAS combines three key ingredients: a new representation for arrays as a list of maps to ensure scalability, a dedicated rewriting step (base normalization) geared at improving the conclusiveness of syntactic (dis-)equality checks between indexes, and lightweight domain reasoning to go beyond purely syntactic checks.

4.1 Dedicated data structure: arrays represented as lists of maps

We look here for an array representation combining the advantages of the list representation (genericity) and the map representation (efficiency) presented in Sec. 3. As a preliminary remark, we can note that the map representation can be extended from the constant-indexes case to the case where all indexes of reads and writes are pairwise *comparable*. By comparable we mean that a binary comparison operator $\prec$ is defined and decidable for every pair of indexes in the formula. Yet, if such a hypothesis might sometimes be satisfied, it is not necessary the case, for example when indexes contain uninterpreted symbols.

The representation of arrays we propose, **lists of maps** (denoted *map lists*), aims precisely at combining advantages of maps when all indexes are pairwise comparable while being as general as lists in other situations. Our array representation can be thought of as a list of *packs of independent writes*. The idea is that sets of comparable (and proven different) indexes can be packed together into map-like data structures, allowing efficient (i.e. logarithmic) search on these packs of indexes during the application of ROW-like simplification rules. *While the idea is presented here in general, we instantiate it in Sec. 4.3, Fig. 7, and in Sec. 4.4, Fig. 8.*

In this representation, nodes of the list are maps from pairwise-comparable indexes to written elements, as illustrated in Fig. 5. A **fresh array** is represented as an empty list (of maps). The array obtained after the **write of element** e **at index** i is defined by:

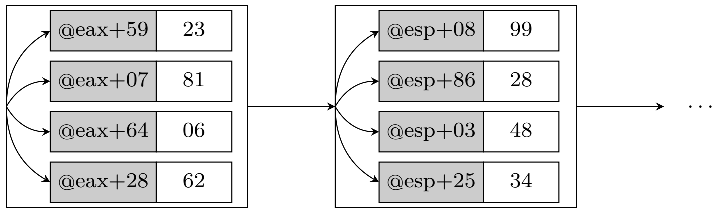

Figure 5: Arrays represented as sequences of packs of independent writes (map list)

- If i is comparable with all other indexes of elements already inserted in the map at head position, then we add the element e at index i into this map (STORE-HIT);
- Else we add to the list a fresh node containing the singleton map of index i to element e (STORE-MISS).

For a **read at index** j, the simplification of ROW is done as follows:

- If indexes in the head position map of the list representing the array are all comparable with j, then if j belongs to this map we substitute the read by the associated element (SELECT-HIT), else we re-iterate on the following node in the list (SELECT-MISS);
- Else, we abort (SELECT-ABORT).

A first version of the dedicated (dis-)equality checks we use is presented in Sec. 4.2. The whole FAS *procedure, together with the associated notion of comparable, is formally described in Sec. 4.3, and a refinement using more semantic checks is presented in Sec. 4.4.*

Intuitively, the benefit of this representation is that its behavior varies between the one of the list representation and the one of the map representation, depending on the proportion of indexes pairwise comparable. Indeed, when all indexes are pairwise comparable, the list only contains a single map of all indexes, which is equivalent to the map representation. And when none of the index pairs are comparable, the list is composed of singleton maps, which is equivalent to the list representation.

From a technical point of view, map lists enjoys several good properties:

Property 1 (Compactness)**.** By construction, all indexes in any map of a map list are pairwise comparable, while indexes from adjacent maps are never comparable.

Property 2 (Complexity)**.** Assuming that 1. we can decide efficiently (constant or logarithmic time) whether an index is comparable to all the other indexes of a given map, 2. that $\prec$ between comparable terms can also be efficiently decided (constant or logarithmic time), and 3. a decent implementation of maps (logarithmic time insertion and lookup), then:

- Array writes are computed in logarithmic time (map insertion) — where the standard list approach requires only constant time;
- Array reads are *also computed in logarithmic time* (map lookup) as SELECT-MISS can only led to SELECT-ABORT (Prop. 1) — where the standard list approach requires *linear time.*

In the case where all indexes are pairwise comparable, our representation contains a single map and simplification cost for r reads and w writes is bounded by $r \cdot ln(w)$, while the list approach requires a quadratic $r \cdot w$ time.

Finally, map lists allow to easily take into account some cases of *write-over-write* (a write masked by a later write at the same index can be ignored if no read happens in-between), while it requires a dedicated and expensive (w^2) treatment with lists.

4.2 Approximated equality check and dedicated rewriting

We consider as equality check a variation of syntactic term equality, namely **syntactic base/offset equality**, which is regarding two terms t_1 and t_2 defined as follows:

- If $t_1 \triangleq \beta_1 + \iota_1$ and $t_2 \triangleq \beta_2 + \iota_2$ — where β_1, β_2 are arbitrary terms (*bases*) and ι_1, ι_2 are constant values (*offsets*), and $\beta_1 = \beta_2$ (syntactically) then return the result of $\iota_1 = \iota_2$,
- Otherwise the check is not conclusive.

This equality check is *correct* and *efficient*, and it strictly extends syntactic term equality — the result is more often conclusive. Actually, in practice it turns out that this extension is significant. Indeed, a common pattern in array formulas coming from software analysis is reads or writes at indexes defined as the sum of a base and an offset (think of C or assembly programming idioms). Hence, dealing with such terms is particularly interesting for verification-oriented formulas.

Dedicated rewriting: base normalization. Yet, this equality check still suffers from the rigidity of syntactic approaches. Therefore it is worthwhile to normalizes indexes as much as possible by applying a dedicated set of rewriting rules called **base normalization** (*rebase*), cf. Fig. 6. These rules are essentially based on limited inlining of variables together with associativity and commutativity rules of $+/-$ operators, the goal being to minimize the number of possible bases in order to increase the "conclusiveness" of our equality check, as done in example Sec. 2.

$$\begin{array}{lccll}
\text{if } u \triangleq v \quad \text{then} & u + k & \rightsquigarrow & v + k & \text{alias inlining} \\
\text{if } u \triangleq v + l \text{ then} & u + k & \rightsquigarrow & v + (k + l) & \text{base/offset inlining} \\
 & -(x + k) & \rightsquigarrow & (-k) - x & \text{constant negation} \\
 & (x + k) + l & \rightsquigarrow & x + (k + l) & \text{constant packing} \\
 & (x + k) + y & \rightsquigarrow & (x + y) + k & \text{constant lifting} \\
 & (x + k) + (y + l) & \rightsquigarrow & (x + y) + (k + l) & \text{base/offset addition} \\
 & (x + k) - (y + l) & \rightsquigarrow & (x - y) + (k - l) & \text{base/offset subtraction} \\
 & & \dots & &
\end{array}$$

Figure 6: Example of base normalization rules. u, v are variables, k, l are constant values and x, y are terms. Non-inlining rules reduce either the number of operators or the depth of constant values, ensuring termination. Note that $(-k), (k + l), (k - l)$ are constant values, not terms.

Optimization: sub-term sharing. Sharing of sub-terms consists in giving a common name to two syntactically equal terms. This improvement is not new, but has an original implication is this context. Besides easing the decision of equality between terms, it remedies to an issue induced by the simplification of ROW. Indeed, the simplification of ROW can be seen as a kind of "inlining" stage, which may in some cases lead to terms size explosion. This problem arises when after a write of element e at index i, several reads at index i are simplified. It may result in numerous copies of term e, term which may contain itself other reads to simplify. By naming and sharing terms read and written in arrays, the sub-term sharing phase prevents this issue. *Experiments in Sec. 5.4 demonstrate the practical interest on very large formulas.*

4.3 The FAS procedure

Using the generic algorithm of Sec. 4.1 with equality check and normalization from Sec. 4.2, we formalize FAS as the set of inference rules presented in Fig. 7. Two terms will be said *comparable* when they share the same base β. STORE-HIT and STORE-MISS rules explain how to update the representation of an array on writes, and SELECT-HIT and SELECT-MISS rules explain how to simplify reads. STORE rules are presented as triples $\{\Lambda\}\,\mathsf{store}\,a\,i\,e\,\{\Lambda'\}$ where Λ' is the representation for $\mathsf{store}\,a\,i\,e$ when Λ is the representation for a. SELECT rules are presented as triples $\{\Lambda\} \vdash \mathsf{select}\,a\,i \rightsquigarrow e$ meaning that $\mathsf{select}\,a\,i$ can be rewritten in e when Λ is the representation for a.

$$\frac{i = \beta + \iota \qquad \iota \text{ a constant}}{\{\langle \Gamma, \beta, b\rangle :: \Lambda\}\,\mathsf{store}\,a\,i\,e\,\{\langle \Gamma\left[\iota \leftarrow e\right], \beta, b\rangle :: \Lambda\}}\ \text{STORE-HIT}$$

$$\frac{i = \alpha + \iota \qquad \alpha \neq \beta}{\{\langle \Gamma, \beta, b\rangle :: \Lambda\}\,\mathsf{store}\,a\,i\,e\,\{\langle \varnothing\left[\iota \leftarrow e\right], \alpha, a\rangle :: \langle \Gamma, \beta, b\rangle :: \Lambda\}}\ \text{STORE-MISS}$$

$$\frac{\Gamma\left[\iota\right] = e \qquad i = \beta + \iota}{\{\langle \Gamma, \beta, b\rangle :: \Lambda\} \vdash \mathsf{select}\,a\,i \rightsquigarrow e}\ \text{SELECT-HIT}$$

$$\frac{\{\Lambda\} \vdash \mathsf{select}\,b\,i \rightsquigarrow e \qquad \Gamma\left[\iota\right] = \varnothing \qquad i = \beta + \iota}{\{\langle \Gamma, \beta, b\rangle :: \Lambda\} \vdash \mathsf{select}\,a\,i \rightsquigarrow e}\ \text{SELECT-MISS}$$

Figure 7: Inference rules for select and store using the *map list* representation

The representation $\langle \Gamma, \beta, b\rangle :: \Lambda$ we use is a specialized version of the map list representation that we just defined, where Γ is a map, β is the common base of indexes present in Γ, b the last revision of the array written at a different index than β, and where Λ is the tail of the list. Assuming that all indexes have been normalized, if the base of the write index is equal to β, then the STORE-HIT rule applies and we add the written element into Γ. If the base of the write index is not equal to β, then the STORE-MISS rule applies. We add as a new node of the list a singleton map containing only the written element, the new base and the written array. For ROW-simplification, the SELECT-HIT rule states that if the base of the read index is equal to β, and if there is an element in Γ mapped to this index, then we return this element. Finally the SELECT-MISS rule states that if there is no such element, then we return the simplified read on b at the same index, using Λ as the representation.

4.4 Refinement: adding domain-based reasoning

While our equality check performs well for deciding (dis-)equalities between indexes with a same base, it behaves poorly with different bases. So we extend FAS in Fig. 8 with *domain-based reasoning* abilities. Basically, maps are now equipped with abstract domains over-approximating their sets of (possible) concrete indexes, and the data structure is now a **list of sets of maps**, all maps in a set having different bases but disjoint sets of concrete indexes. When syntactic base/offset equality check is not conclusive, domain intersection may be used to prove disequality.

We borrow ideas from Abstract Interpretation [13]. Given a concrete domain D, an abstract domain is a complete lattice $\left\langle D^\sharp, \sqsubseteq, \sqcup, \sqcap, \top, \bot \right\rangle$ coming with a monotonic concretization function $\gamma : D^\sharp \mapsto \mathcal{P}\left(D\right)$ such that $\gamma\left(\top\right) = D$ and $\gamma\left(\bot\right) = \varnothing$. An element of an abstract domain is called an abstract value. In the following the concrete domain is the set of array indexes.

The representation is now a list of sets of tuples $\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle$ where Γ, β and b are a map, a base and an array as previously described, and where $\Gamma^\sharp$ is the joined abstract value of indexes

$$\frac{i = \beta + \iota \qquad \iota \text{ a constant} \qquad \begin{array}{l}\Theta = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \beta \wedge \Sigma^\sharp \sqcap i^\sharp = \bot\right\} \\ \Xi = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \beta \wedge \Sigma^\sharp \sqcap i^\sharp \neq \bot\right\}\end{array}}{\left\{\left(\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle \oplus \Theta \uplus \Xi\right) :: \Lambda\right\} \mathsf{store}\, a\, i\, e \left\{\left(\left\langle \Gamma\left[\iota \leftarrow e\right], \beta, b, \Gamma^\sharp \sqcup i^\sharp \right\rangle \oplus \Theta\right) :: \Xi :: \Lambda\right\}} \text{ STORE-HIT}$$

$$\frac{i = \alpha + \iota \qquad \iota \text{ a constant} \qquad \begin{array}{l}\Theta = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \alpha \wedge \Sigma^\sharp \sqcap i^\sharp = \bot\right\} \\ \Xi = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \alpha \wedge \Sigma^\sharp \sqcap i^\sharp \neq \bot\right\}\end{array}}{\left\{(\Theta \uplus \Xi) :: \Lambda\right\} \mathsf{store}\, a\, i\, e \left\{\left(\left\langle \varnothing\left[\iota \leftarrow e\right], \alpha, a, i^\sharp \right\rangle \oplus \Theta\right) :: \Xi :: \Lambda\right\}} \text{ STORE-MISS}$$

$$\frac{\Gamma\left[\iota\right] = e \qquad i = \beta + \iota}{\left\{\left(\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle \oplus \Xi\right) :: \Lambda\right\} \vdash \mathsf{select}\, a\, i \rightsquigarrow e} \text{ SELECT-HIT}$$

$$\frac{\{\Lambda\} \vdash \mathsf{select}\, b\, i \rightsquigarrow e \qquad \Gamma\left[\iota\right] = \varnothing \qquad i = \beta + \iota}{\left\{\left(\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle \oplus \Xi\right) :: \Lambda\right\} \vdash \mathsf{select}\, a\, i \rightsquigarrow e} \text{ SELECT-MISS}$$

$$\frac{\{\Lambda\} \vdash \mathsf{select}\, b\, i \rightsquigarrow e \qquad i = \beta + \iota \qquad \Theta = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \beta \wedge \Sigma^\sharp \sqcap i^\sharp = \bot\right\}}{\{\Theta :: \Lambda\} \vdash \mathsf{select}\, a\, i \rightsquigarrow e} \text{ SELECT-SKIP}$$

Figure 8: Inference rules for select and store using domains

in Γ. Given a write at index i, the set at head position in the list is split into: 1. Θ the set of tuples whose map abstract value does not overlap with $i^\sharp$, the abstract value of i, 2. Ξ the set of tuples whose map abstract value overlap with $i^\sharp$, and if it exists, 3. the tuple $\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle$ where β is after normalization the base of i. If this tuple exists, then the STORE-HIT rule applies. We update Γ as previously and its associated abstract value becomes the join value of $\gamma^\sharp$ and $i^\sharp$. We append first Ξ alone onto the list, and then Θ together with the updated tuple. Else, the STORE-MISS rule applies. Again we first append Ξ alone, then Θ together with a new singleton map, the new base, the written array and the write index abstract value. Finally, SELECT-HIT and SELECT-MISS are similar to previous ones, but we add a new rule SELECT-SKIP. This rule states that, if the read index abstract value do not overlap with maps abstract values in the set at head position, then we drop the head and reiterate on the tail of the list.

Note that if abstract values in these rules are set to $\top$, then Θ is always empty and we get back to the previous inference rules. Also the complexity of reads becomes linear in the list size, as domains can prove disequality at each node of the list. Yet, it is not a problem in practice, as demonstrated by experimental evaluation in Sec. 5.

Domain propagation. So far, we did not explained how abstract values are computed. The literature on abstract domains is plentiful [28]. Nevertheless we present in Fig. 9 propagation rules for a specific abstract domain, the well-known domain of (multi-)intervals — used in our implementation (note that operations are performed over bitvectors of a known size N, and $+$ is the wraparound addition). The general difficulty is to find a sweet spot between the potential gain (more checks become conclusive) and the overhead of propagation. As a rule of thumb, *non-relational domains* should be tractable and useful. Especially, combining multi-intervals with congruence (e.g. $x \equiv 5 \bmod 8$) or bit-level information (e.g. the second bit of x is 1) [3] is a good candidate for refining our method at an affordable cost.

Let i, j two bitvectors of size N, with $i^\sharp = [m_i, M_i], j^\sharp = [m_j, M_j]$ where $0 \leq m_{i,j} \leq M_{i,j} \leq 2^N$,

$$\begin{array}{rll}
c^\sharp & = [\mathrm{c}, \mathrm{c}] & \text{for any constant } c \\
v^\sharp & = [m_i, M_j] & \text{if } i \leq v \leq j \\
(\mathrm{extract}_{l,h}\, i)^\sharp & = [0, 2^{h-l+1} - 1] & \text{if } (M_i \gg l) - (m_i \gg l) \geq 2^{h-l+1} \\
& = [\mathrm{extract}_{l,h}(m_i), \mathrm{extract}_{l,h}(M_i)] & \text{if } \mathrm{extract}_{l,h}(M_i) \geq \mathrm{extract}_{l,h}(m_i) \\
& = [0, \mathrm{extract}_{l,h}(M_i)] & \text{otherwise} \\
& \quad \sqcup [\mathrm{extract}_{l,h}(m_i), 2^{h-l+1} - 1] & \\
(i+j)^\sharp & = [m_i + m_j, M_i + M_j] & \text{if } M_i + M_j < 2^N \\
& = [m_i + m_j - 2^N, M_i + M_j - 2^N] & \text{if } m_i + m_j \geq 2^N \\
& = [m_i + m_j - 2^N, 2^N - 1] & \text{otherwise} \\
& \quad \sqcup [0, M_i + M_j - 2^N] &
\end{array}$$

Figure 9: Examples of propagation for intervals. These propagations are extended to multi-intervals by distribution for unary operators and pairwise distribution for binary operators.

5 Implementation and experimental evaluation

5.1 Implementation

In order to evaluate the efficiency of our approach, we implemented FAS (with the different representations presented so far and the abstract domain of multi-intervals) as a preprocessor for SMT formulas belonging to the QF_ABV logic (*quantifier-free formulas over the theory of bitvectors and arrays*) — as typical choice in software verification. In that setting, all bitvector values and expressions have statically known sizes, arithmetic operations are performed *modulo* and values can "wraparound". For reproducibility purposes source code and benchmarks are available online[2]. The implementation comprises 6,300 lines of OCaml integrated into the TFML SMT formula preprocessing engine [19], part of the BINSEC symbolic execution tool [15]. It comprises all simplifications and optimizations described in Sec. 4, including map lists, base normalization, sub-term sharing and domain propagation (multi-intervals) over bitvectors. Note that our normalization rules (Sec. 4.2) and domain propagators (Sec. 4.4) correctly handle possible arithmetic wraparounds.

An advantage operating as a preprocessor is to be independent of the underlying solver used for formula resolution, and therefore allow us to evaluate the impact of our approach with several of them. A drawback is that we do not have access to various internal components of the solver, like accessing the model under construction, and cannot use them to refine our approach. In the long term, a deeper integration into a solver would be more suitable.

5.2 Experimental setup

We evaluated FAS performances under three criteria : 1. *simplification thoroughness*, measured by the reduction of the number of ROW terms; 2. *simplification impact*, measured by resolution time before and after simplification; 3. *simplification cost*, measured by the total time of simplification.

We devise three sets of experiments corresponding to three different scenarios: mid-sized formulas generated by the SE-tool BINSEC [15] from real executables programs — typical of test generation and vulnerability finding (cf. Sec. 5.3), very large formulas generated by BINSEC from

[2]http://benjamin.farinier.org/lpar2018/

very long traces — typical of reverse and malware analysis (cf. Sec. 5.4), and formulas taken from the SMT-LIB benchmarks (cf. Sec. 5.5). Regarding experiments over SE-generated formulas, we also consider three variants corresponding to standard concretization / symbolization policies [14] (cf. Sec. 5.3), as well as different TIMEOUT values. Experiments are carried out on an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2,60GHz. We consider three of the best SMT solvers for the QF_ABV theory, namely Boolector [8], Yices [18] and Z3 [16].

Note that the impact of map lists (w.r.t. a list-based representation) and sub-term sharing will be evaluated only in Sec. 5.4, as they are interesting only on large enough formulas. Moreover, the map list representation impacts only preprocessing time, not its thoroughness: assuming preprocessing does not time out (and rebase and domains are used), FAS *and* FAS*-list will carry out the same simplifications.*

A note on problem encoding. As already stated, we consider quantifier-free formulas over the theory of bitvectors and arrays coming from the encoding of low-level software verification problems. Arithmetic operations are performed *modulo* and values can "wraparound". Also, since memory accesses in real hardware are performed at word-level (reading 4 or 8 bytes at once), they are modelled here by successive byte-level reads and writes — allowing to take properly into account misaligned or overlapping accesses. Finally, memory is often modelled as a single logical array of bytes (i.e., bitvector values of size 8), without any *a priori* distinction betweeen stack and heap (this is the case for all examples from BINSEC).

5.3 Medium-size formulas from SE

We consider here typical formulas coming from symbolic execution over executable codes. While mid-sized (max. 3.42 MB, avg. 1.40 MB), these formulas comprise quite long sequences of nested ROW (max. 11,368 ROW, avg. 4,726 ROW) as there is only one initial array (corresponding to the initial memory of the execution, i.e. a flat memory model). More precisely, we consider 6,590 traces generated by BINSEC [15] from 10 security challenges (e.g. `crackme` such as Manticore or Flare-On) and vulnerability finding problems (e.g. GRUB vulnerability), and from these traces we generate 3 x 6,590 formulas depending on the concretization / symbolization policies used in BINSEC to generate them: **concrete** (all array indexes are set to constant values), **symbolic** (symbolic array indexes), and **interval** (array indexes bound by intervals). We consider two different TIMEOUT: 1,000 seconds (close to SMT-LIB benchmarks setting) and 1 second (typical of program analysis involving a large number of solver calls, e.g. deductive verification or symbolic execution).

The whole results are presented in Table 1 (TIMEOUT 1,000 sec.) and Table 2 (TIMEOUT 1 sec.). Note that resolution time does not include TIMEOUT. Columns FAS and FAS-itv represents respectively our technique (map list, rebase and sharing) potentially improved with domain reasoning based on intervals (FAS-itv). The **default** column represents a minimal preprocessing step consisting of constant propagation and formula pruning, *without any array simplification.*

We can see that:

- *Simplification time* is always very low on these examples (340 sec. for 3 x 6,590 formulas, in avg. 0.017 sec. per formula). Moreover, it is also very low w.r.t. resolution time (taking TIMEOUT into account: Boolector 6%, Yices 4% and Z3 0.3%) and largely compensated by the gains in resolution, but for one case where Boolector performs especially well (concrete formulas: cost of 118% — not compensated by gains in resolution).
- Formula simplification is indeed *thorough*: as a whole, the number of ROW is reduced by a factor 5 (2.5 without interval reasoning). The simplification performs extremely well, as

Table 1: 6,590 x 3 medium-size formulas from SE, with TIMEOUT = 1,000 sec.: simplification time (in seconds), number of ROW after simplification, number of TIMEOUT and resolution time (in seconds, without TIMEOUT)

		simpl. time	#TIMEOUT and resolution time						#ROW
			Boolector		Yices		Z3		
concrete	default	61	0	163	2	69	0	872	866,155
	FAS	85	0	94	2	68	0	244	1,318
	FAS-itv	111	0	94	2	68	0	224	1,318
interval	default	65	0	2,584	2	465	31	155,992	866,155
	FAS	99	0	2,245	2	487	25	126,806	531,654
	FAS-itv	118	0	755	2	140	14	37,269	205,733
symbolic	default	61	0	6,173	3	1,961	65	305,619	866,155
	FAS	91	0	6,117	3	1,965	66	158,635	531,654
	FAS-itv	111	0	4,767	2	1,108	43	80,569	295,333
total	default	187	0	8,922	7	2,495	96	462,484	2,598,465
	FAS	275	0	8,458	7	2,520	91	285,686	1,064,626
	FAS-itv	340	0	5,616	6	1,317	57	37,573	502,384

Table 2: 6,590 x 3 medium-size formulas from SE, with TIMEOUT = 1 sec.

		#TIMEOUT and resolution time					
		Boolector		Yices		Z3	
concrete	default	2	93	2	3.12	2	655
	FAS	2	24	2	2.54	2	39
	FAS-itv	2	23	2	2.51	2	40
interval	default	1,230	730	57	184	480	751
	FAS	593	1,213	58	181	483	773
	FAS-itv	52	602	6	66	273	665
symbolic	default	1,947	575	2,771	307	3,497	438
	FAS	1,888	618	2,723	310	3,470	442
	FAS-itv	1,597	647	1,473	528	2,895	504
total	default	3,179	1,399	2,830	494	3,979	1,845
	FAS	2,483	1,856	2,783	495	3,955	1,254
	FAS-itv	1,651	1,273	1,481	597	3,170	1,210

expected, on *concrete* formulas, where almost all ROW instances are solved at preprocessing time. On *interval* formulas, the number of ROW is sliced by a factor 4, and a factor 3 in the case of *full symbolic* formulas.

- The *impact* of the simplification over resolution time (for a 1,000 sec. TIMEOUT) varies greatly from one solver to another, but it is always significant: factor 1.5 for Boolector, factor 1.9 for Yices with one fewer TIMEOUT, up to a factor 3.8 and 32 fewer TIMEOUT for Z3. Especially, on interval formulas FAS with domain reasoning yields a 3.4 (resp. 3.3) speed factor for Boolector (resp. Yices), while Z3 on this category enjoys a 4.1 speedup together with 14 fewer TIMEOUT. Interestingly, domain reasoning is useful also in the case of fully symbolic formulas, i.e. with no explicit introduction of domain-based constraints.

Results for a 1 sec. TIMEOUT follows the same trend but they are much more significant (number of TIMEOUT: Boolector -48%, Yices -47% and Z3 -21%), and they become especially dramatic on interval formulas (number of TIMEOUT: Boolector -96%, Yices -90% and Z3 -44%).

Focus on specific cases. We highlight now a few interesting scenarios where FAS performs very well, especially formulas generated from the GRUB vulnerability (Table 3, 753 formulas) and formulas representing the inversion of a crypto-like challenge (Table 4, 139 formulas). Regarding GRUB, while basic FAS does not really impact resolution time, adding domain-based reasoning does allow a significant improvement — Boolector, Yices and Z3 becoming respectively 4.1x, 4.7x and 7x faster. Regarding UNGAR, again FAS alone does not improve resolution time (for Z3, we even see worse performance), but adding interval reasoning yields dramatic improvement: Boolector becomes 18.8x faster, Yices becomes 48.2x faster (with -1 TIMEOUT) and Z3 does not time out anymore (-12 TIMEOUT).

Table 3: GRUB (interval), 753 formulas — Number of TIMEOUT and resolution time (in seconds, without TIMEOUT)

	#TIMEOUT and resolution time					
GRUB	Boolector		Yices		Z3	
default	0	508	0	258	0	31,322
FAS	0	505	0	257	1	26,809
FAS-itv	0	123	0	54	0	4,481

Table 4: UNGAR (symbolic), 139 formulas — Number of TIMEOUT and resolution time (in seconds, without TIMEOUT)

	#TIMEOUT and resolution time					
UNGAR	Boolector		Yices		Z3	
default	0	359	3	627	12	926
FAS	0	373	3	624	12	1,130
FAS-itv	0	19	2	13	0	569

Conclusion. On these middle-size formulas coming from typical SE problems, we can draw the following conclusion: **Speed.** FAS is extremely efficient and does not yield any noticeable overhead; **Thoroughness.** Formula simplification is significant — even on fully symbolic formulas, and it becomes (as expected) dramatic on "concrete" formulas; **Impact.** The impact of FAS varies across solvers and formulas categories, yet it is always positive and it can be dramatic in some settings (low TIMEOUT, interval formulas, etc.).

5.4 Very large formulas

We now turn our attention to large formulas (max. 458 MB, avg. 45 MB) involving very long sequences of nested ROW (max. 510,066 ROW, avg. 49,850 ROW), as can be found for example in symbolic deobfuscation. We consider 29 benchmarks taken from a recent paper on the topic [27] representing execution traces over (mostly non crypto-) hash functions (e.g. `MD5`, `City`, `Fast`, `Spooky`, etc.) obfuscated by the Tigress tool [12]. We also consider a trace taken from the ASPack packing tool.

Results are presented in Table 5, where FAS-list represents our simplification method where the map list is replaced by a normal list — getting an improved version of the standard list-based ROW-simplification (the goal being to evaluate the gain of our new data structure). Again, simplification is significant with a strong impact on the number of time outs and on resolution time, especially in the concrete case and for Z3. Impact in the symbolic case is more mixed but positive (-1 TIMEOUT for Boolector and Z3, no impact for Yices). In term of size, FAS reduces formulas to max. 86.49MB, avg. 6.98MB, and FAS-itv to max. 86.45MB, avg. 6.17MB. If sub-term sharing is disabled, formulas size jumps to max. 591.99MB, avg. 14.95MB for FAS and max. 591.71MB, avg. 16.35MB for FAS-itv. Regarding simplification time, FAS-list suffers from scalability issues on these formulas (5x slower than FAS).

Table 5: 29 x 3 very large formulas from SE, with TIMEOUT = 1,000 sec.: simplification time (in seconds), number of ROW after simplification, number of TIMEOUT and resolution time (in seconds, without TIMEOUT)

		simpl. time	#TIMEOUT and resolution time						#ROW
			Boolector		Yices		Z3		
concrete	default	44	10	159	4	1,098	26	3.33	1,120,798
	FAS-list	1,108	8	845	4	198	10	918	456,915
	FAS	196	8	820	4	196	10	922	456,915
	FAS-itv	210	4	654	1	12	4	1,120	0
interval	default	44	12	131	12	596	27	0.19	1,120,798
	FAS-list	222	12	129	12	595	26	236	657,594
	FAS	231	12	129	12	597	26	291	657,594
	FAS-itv	237	12	58	12	28	19	81	651,449
symbolic	default	40	12	1,522	12	1,961	27	0.13	1,120,798
	FAS-list	187	11	1,199	12	2,018	26	486	657,594
	FAS	194	11	1,212	12	2,081	26	481	657,594
	FAS-itv	200	11	1,205	12	2,063	26	416	657,594

The ASPack example. We now turn our attention to the formula generated from a trace of a program protected by ASPack (96 MB and 363,594 ROW, *concrete* mode). Solving the formula is highly challenging: while Yices succeeds in a decent amount of time (69 seconds), Z3 terminates in 2h36min while Boolector needs 24h. Table 6 presents our results on this particular example. FAS performs extremely well (Table 6), turning resolution time from hours to a few seconds (Boolector) or minutes (Z3). Yices also benefits from it. Especially, all ROW instances are simplified away. FAS and FAS-itv reduce the ASPack formula size to 3.81MB, while it jumps to 443.54MB when sub-term sharing is disabled. Interestingly, this example clearly highlights the scalability of FAS w.r.t. a standard list-based approach, passing roughly from 1h (list) to 1 minute (FAS). Fig. 10 proposes a detailed view of the performance and impact of the standard list-based simplification method (Boolector only), depending on the bound for backward reasoning (the standard method has no bound). For comparison, the two horizontal lines represent simplification and resolution time with FAS. We can see that bounding the list-based reasoning has no tangible effect here, as we need at least a 3,000 seconds (50 minutes) simplification time to get a resolution time under 3,000 seconds.

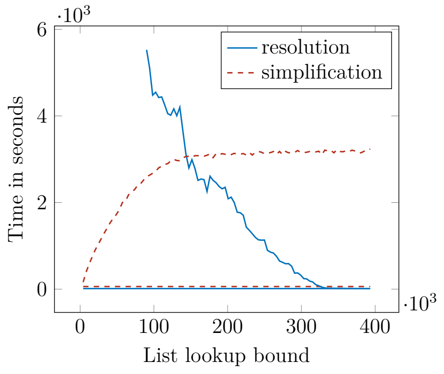

Figure 10: Boolector on ASPack

Table 6: ASPack formula, without TIMEOUT

ASPack	simpl. time	resolution time			#ROW
		Boolector	Yices	Z3	
default	15 sec.	≈ 24h	69 sec.	2h36	360,991
FAS-list	53 min.	9.7 sec.	3.4 sec.	183 sec.	0
FAS	61 sec.	9.7 sec.	3.4 sec.	183 sec.	0
FAS-itv	63 sec.	9.8 sec.	3.4 sec.	182 sec.	0

Conclusion. Once again FAS appears to be fast and to have a significant impact on resolution time, especially in the concrete case where the difference can be from several hours to a few seconds (total resolution + simplification: a few minutes). Moreover, it appears clearly that on very long traces FAS scales much better than the standard list-based ROW-simplification method.

5.5 SMT-LIB formulas

We consider now the impact of FAS on formulas taken from the SMT-LIB benchmarks. These formulas are notably different from the ones considered in the two previous experiments: while most of them do come from verification problems, they may involve complex Boolean structure (rather than "mostly conjunctive" formulas) and they do not necessarily exhibit very deep chains of ROW. These kinds of formulas are not our primary objective, yet we seek to evaluate how our technique performs on a "bad case". We evaluate FAS on all the 15,016 SMT-LIB formulas from QF_ABV theory. TIMEOUT is set to 1,000 seconds. Results are reported in Table 7. Note that, again, resolution time does not include TIMEOUT.

Table 7: 15,016 formulas from SMT-LIB benchmarks, with TIMEOUT = 1.000 sec.: simplification time (in seconds), number of ROW after simplification, number of TIMEOUT and resolution time (in seconds, without TIMEOUT)

SMT-LIB	simpl. time	#TIMEOUT and resolution time						#ROW
		Boolector		Yices		Z3		
default	87	59	20,126	151	28,156	158	41,925	548,176
FAS	378	54	19,922	148	26,657	147	43,090	469,815
FAS-itv	378	55	19,843	146	28,703	149	40,873	469,567

Conclusion. FAS is again very efficient on these formulas (avg. 0.025 sec. per formula), and reduces the number of ROW by -14%. Yet the impact of simplifications, while slight, is clearly positive on both TIMEOUT (Boolector -8%, Yices -2% and Z3 -7%) and resolution time (for Yices, only when taking TIMEOUT time into account). Such gains are not anecdotal as the best SMT solvers are highly tuned for SMT-LIB. Since the number of TIMEOUT is the main metric for SMT-LIB, Boolector with FAS would have won the last edition for QF_ABV theory. Finally, domain reasoning does not add anything here (but for Yices) — either the benchmark formulas do not exhibit such interval constraints, or our propagation mechanism is too crude to take advantage of it.

5.6 Conclusion

Our experiments demonstrate that our approach is *efficient* (the cost is almost always negligible w.r.t. resolution time) and *scalable* (compared to the list-based method). The simplification is *thorough*, removing a large fraction of ROW. The *impact is always positive* (both in resolution time and number of time outs), and it is *dramatic for some key usage scenarios* such as SE-like formulas with small TIMEOUT or very large size.

Finally, we can note that domain reasoning is usually helpful (though, not on SMT-LIB formulas) and that it shows a powerful synergy with the "interval C/S policy" in SE — yielding a new interesting sweet spot between tractability and genericity of reasoning.

6 Related work

A preliminary *work in progress* version of this work was published in a French workshop [20], in French (6 pages). The current article adds a much more refined description, the domain reasoning part and a much more systematic and thorough experimental evaluation (including SMT-LIB, long traces over packed hash functions, etc.).

Decision procedures for the theory of arrays. Surprisingly, there have been relatively few works on the efficient handling of the (basic) theory of arrays. Standard symbolic approaches for pure arrays complement symbolic read-over-write preprocessing [22, 5, 2] with enumeration on (dis-)equalities, yielding a potentially huge search space. New array lemmas can be added on-demand or incrementally discovered through an abstraction-refinement scheme [9]. Another possibility is to reduce the theory of arrays to the theory of equality by systematic "inlining" of the array axioms to remove all store operators, at the price of introducing many case-splits The encoding can be eager [24] or lazy [9]. Our method generalizes previous preprocessing [22, 2] and is complementary to complete resolution methods [9, 24]. Note also that our approach could benefit from being integrated directly within such a complete resolution method, allowing incremental simplification all along the resolution process.

Decision procedures have also been developed for expressive extensions of the array theory, such as arrays with extensionality (i.e. equality over whole arrays) or the array property fragment [7], which enables limited forms of quantification over indexes and arithmetic constraints. These extensions aim at increasing expressiveness and they do not focus so much on practical efficiency. Our method can also be applied to these settings (as ROW are still a crucial issue), even though it will not cover all difficulties of these extensions.

Optimized handling of arrays inside tools. Many verification and program analysis tools and techniques ultimately rely on solving logical formulas involving the theory of arrays. Since the common practice is to re-use existing (SMT) solvers, these approaches suffer from the limitations of the current solvers over arrays. As a mitigation, some of these tools take into account knowledge from the application domain in order to generate relevant (but usually not equivalent) and simpler formulas [25, 21] — see also the specific case of SE over concrete indexes discussed in Sec. 3. Our method is complementary to these approaches as it operates on arbitrary formulas and the simplification keeps logical equivalence.

7 Conclusion

The theory of arrays has a central place in software verification due to its ability to model memory or data structures. Yet, this theory is known to be hard to solve because of *read-over-write* terms (ROW), especially in the case of very large formulas coming from unrolling-based verification methods. We have presented FAS, an original simplification method for the theory of arrays geared at eliminating ROW, based on a new dedicated data structure together with original simplifications and low-cost reasoning. The technique is efficient, scalable and it yields significant simplification. The impact on formula resolution is always positive, and it can be dramatic on some specific classes of problems of interest, e.g. very long formula or binary-level symbolic execution. These advantages have been experimentally proven both on realistic formulas coming from symbolic execution and on SMT-LIB formulas.

Future work includes a deeper integration inside a dedicated array solver in order to benefit from more simplification opportunities along the resolution process, as well as exploring the interest of adding more expressive domain reasoning.

References

[1] S. Bardin, R. David, and J. Marion. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 633–651. IEEE, 2017.

[2] S. Bardin and A. Gotlieb. Fdcc: A combined approach for solving constraints over finite domains and arrays. In *Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'12, pages 17–33, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] S. Bardin, P. Herrmann, and F. Perroud. An alternative to sat-based approaches for bit-vectors. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010*. Springer, 2010.

[4] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking.*, pages 305–343. Springer, 2018.

[5] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. A write-based solver for SAT modulo the theory of arrays. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*. IEEE, 2008.

[6] A. R. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification (Chapters 5, 6 and 12)*. Springer, 2007.

[7] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 174–177, 2009.

[9] R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201, 2009.

[10] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[11] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS, Barcelona, Spain, March 29 - April 2, 2004*, pages 168–176, 2004.

[12] C. S. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 319–328, 2012.

[13] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1977.

[14] R. David, S. Bardin, J. Feist, L. Mounier, M. Potet, T. D. Ta, and J. Marion. Specification of concretization and symbolization policies in symbolic execution. In *ISSTA, Saarbrücken, Germany, July 18-20, 2016*, pages 36–46, 2016.

[15] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *SANER, Suita, Osaka, Japan, March 14-18, 2016*, pages 653–656, 2016.

[16] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] P. J. Downey and R. Sethi. Assignment commands with array references. *J.ACM*, 25(4):652–666, 1978.

[18] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[19] B. Farinier, S. Bardin, R. Bonichon, and M. Potet. Model generation for quantified formulas: A taint-based approach. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 294–313. Springer, 2018.

[20] B. Farinier, S. Bardin, and R. David. Simplification efficace pour la théorie des tableaux. In *Journées Francophones des Langages Applicatifs, January 24-27, Banyuls-sur-Mer, France*, 2018.

[21] A. Fromherz, K. S. Luckow, and C. S. Pasareanu. Symbolic arrays in symbolic pathfinder. *ACM SIGSOFT Software Engineering Notes*, 41(6):1–5, 2016.

[22] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV, Berlin, Germany, July 3-7, 2007*, pages 519–531, 2007.

[23] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[24] D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[25] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 68–78. ACM, 2017.

[26] J. Rushby. *Verified software: Theories, tools, experiments*, chapter Automated Test Generation and Verified Software. Springer, 2008.

[27] J. Salwan, S. Bardin, and M. Potet. Symbolic deobfuscation: From virtualized code back to the original. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, pages 372–392. Springer, 2018.

[28] A. Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities.* Springer, 2008.

[29] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 674–691. IEEE, 2015.