PARTI: A Multi-interval Theory Solver for Symbolic Execution

Oscar Soria Dustmann
Comm. and Distributed Systems
RWTH Aachen University, Germany
dustmann@comsys.rwth-aachen.de

Klaus Wehrle
Comm. and Distributed Systems
RWTH Aachen University, Germany
wehrle@comsys.rwth-aachen.de

Cristian Cadar
Department of Computing
Imperial College London, UK
c.cadar@imperial.ac.uk

ABSTRACT

Symbolic execution is an effective program analysis technique whose scalability largely depends on the ability to quickly solve large numbers of first-order logic queries. We propose an effective general technique for speeding up the solving of queries in the theory of arrays and bit-vectors with a specific structure, while otherwise falling back to a complete solver.

The technique has two stages: a learning stage that determines the solution sets of each symbolic variable, and a decision stage that uses this information to quickly determine the satisfiability of certain types of queries. The main challenges involve deciding which operators to support and precisely dealing with integer type casts and arithmetic underflow and overflow.

We implemented this technique in an incomplete solver called PARTI ("PARtial Theory solver for Intervals"), directly integrating it into the popular KLEE symbolic execution engine. We applied KLEE with PARTI and a state-of-the-art SMT solver to synthetic and real-world benchmarks. We found that PARTI practically does not hurt performance while many times achieving order-of-magnitude speedups.

CCS CONCEPTS

• **Theory of computation → Constraint and logic programming**; • **Software and its engineering → Software testing and debugging**;

KEYWORDS

Constraint solving, Symbolic execution, Multi-intervals

ACM Reference Format:
Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. 2018. PARTI: A Multi-interval Theory Solver for Symbolic Execution. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238179

1 INTRODUCTION

Symbolic execution is a popular testing and analysis technique which has the ability to explore multiple paths in the program under testing with the goal of detecting software errors [1, 7, 8, 17, 18]. At a high level, symbolic execution runs the program on *symbolic input*, which is initially not constrained to any particular value. Whenever execution reaches a branch that depends—directly or indirectly—on the symbolic input, execution may be split in two: on the *then* path, the input is constrained to satisfy the branch condition (e. g., $x > 2$), while on the *else* path it is constrained to satisfy the negation of the branch condition (e. g., $x \le 2$). On each path, a constraint solver is used to determine whether the current conjunction of constraints collected at each branch—called the *path condition*—is satisfiable. If it is not, that path is terminated as it is infeasible. Finally, when a path finishes execution, a constraint solver can provide a solution for the path condition, which represents a test input that can be used to replay the corresponding path, e. g., for debugging purposes.

The effectiveness of symbolic execution depends directly on the effectiveness of the underlying constraint solver, as often more than 90% of the total execution time of a symbolic execution engine is spent in the solver [26]. Modern symbolic execution tools rely on existing Satisfiability Modulo Theory (SMT) solvers, which can answer satisfiability problems in a given theory, e. g., linear arithmetic. Perhaps the most popular theory employed by modern symbolic execution engines is that of Quantifier Free Arrays and Bit-Vectors (QF_ABV), as it can be used [1, 6, 16, 18, 25] to precisely model the semantics of popular programming languages such as C, C++ or Java, and we will therefore consider only this theory in the remainder of this paper.

State-of-the-art SMT solvers, such as Boolector [4], STP [16], and Z3 [11], generally start with a pre-processing and optimisation stage, in which an instance of QF_ABV is simplified using various strategies, followed by a bit-blasting stage, in which the query is translated from the level of the theory to that of a boolean satisfiability (SAT) problem. While the pre-processing and optimisation stage tries to exploit the characteristics of the theory, it is oblivious to those of the queries generated during symbolic execution. As a result, symbolic execution engines perform their own pre-processing and optimisation stage before invoking the underlying SMT solver. Examples include performing simple arithmetic simplifications [6, 7], caching solutions [7, 33, 34], exploiting logical implications [6, 24] and rewriting complex array constraints [27].

In this paper, we introduce a new constraint solving optimisation technique for symbolic execution, implemented in an incomplete solver called PARTI ("PARtial Theory solver for Intervals"), which aims to exploit constraint solving queries whose solutions can be expressed as the union of a small number of intervals. Hence, PARTI provides a mechanism to solve only a subset of SMT problems fast and resorts to an off-the-shelf solver for other queries. For

ASE '18, September 3–7, 2018, Montpellier, France
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238179

instance, such queries are generated by inequality conditions involving a symbolic input and a constant value, and are prevalent in real programs as indicated by our evaluation (cf. §4). Furthermore, they often occur in some symbolic execution implementations in the pointer resolution stage, when determining valid objects of a symbolic pointer value.

At a high-level, when deciding a query, PARTI keeps track of all possible values for each symbolic variable. For instance, the solution set of an unsigned 8-bit integer symbolic variable x could be $[1, 3] \cup [120, 140]$. If later x is constrained to be greater than 5 by following the *then* side of the branch `if (x > 5)`, its solution set becomes just $[120, 140]$, as none of the values in $[1, 3]$ satisfy `x > 5`. With an efficient representation of the solution set, PARTI can quickly solve certain types of queries and bypass the more expensive complete solver. E. g., it can quickly determine that the query $x > 200$ is unsatisfiable, since the values for x are in $[120, 140]$.

The first challenge is to precisely deal with integer type casts and arithmetic underflow and overflow. To better understand this challenge, consider our previous example where valid solutions are $x \in [120, 140]$. If x is now type-cast to a signed 8-bit variable y (with a range of $[-128, 127]$), the solution set for y is $[120, 127] \cup [-128, -116]$ as some values in $[120, 140]$ overflow.

The second challenge is the selection of supported operands and an efficient representation of the solution set in order to balance applicability and performance: The sketched approach clearly cannot efficiently solve every single SMT query as the explicit derivation of the solution set of all variables is impossible in the general case and even when possible, it may lead to exponential blowup depending on the representation of the solution set.

Our contribution is firstly a scheme for effectively reasoning about such interval queries in the context of symbolic execution, together with a comprehensive analysis of the operations that we decide to handle, secondly an implementation of the presented technique and a thorough evaluation of the performance gains it can achieve.

2 PRELIMINARIES

We denote a QF_ABV SMT query q as a pair (e, C), where e is a query expression whose satisfiability we try to determine and C is the set of constraints that are known to hold. For example, the query $(\texttt{Eq}(a, 3), \{\texttt{Ule}(1, a), \texttt{Ule}(a, 5)\})$ is an SMT query that is satisfiable but not a tautology (cf. Table 1). If the solution set for a variable x is independent from other variables, we denote its solutions as $[\![x]\!]$, e. g., $[\![a]\!] = \{1, 2, 3, 4, 5\}$.

Each expression has a bit-width. E. g., e and all expressions in C have bit-width 1, i. e. they are boolean expressions. We represent an expression by an Abstract Syntax Tree (AST). This tree consists of (1) internal vertices, representing operators combining one or more sub-expressions, e. g., `Add`, and, (2) leaf vertices, representing either a constant, e. g., `Const` (42), or a read from a free variable, e. g., `Read` (x). For readability, we may omit explicitly specifying the `Read` and `Const` operators. Note that there may be several reads with different offsets and bit-widths from a free variable in one AST, but we restrict PARTI to such queries that have no partially overlapping reads for any given variable.

Table 1: The relevant QF_ABV operators for this paper. Full SMT encompasses a longer list, including, e. g., modulo.

Operator	Children	Semantics
`Const`	0	leaf: constant value
`Read`	0	leaf: free variable
`LShr`	1	logical right-shift
`Not`	1	negation
`ZExt`, `SExt`	1	unsigned/signed type conversion
`Add`, `Sub`	2	addition/subtraction
`Eq`	2	equality
`Mul`	2	multiplication
`Ule`, `Sle`	2	unsigned/signed less-than-or-equal-to
`Ult`, `Slt`	2	unsigned/signed less-than

The list of relevant operators discussed in this paper is depicted in Table 1. Note that all queries are assumed to be normalised to exclude operators such as 'not-equal' as these can be trivially lowered to the given operators.

3 CORE DESIGN

PARTI has been designed to enable very fast solving of a subset of SMT queries with a structure that many programs produce frequently during symbolic execution. That is, PARTI attempts to solve a query but may give up at any time while processing the expression or the constraints. This will be communicated to a complete SMT solver which acts as a fall-back. We distinguish between *complete solvers*, that, given enough resources, can solve all SMT queries, and *incomplete* solvers, such as PARTI, that can only solve (efficiently) a subset of SMT. In this section, we characterise the subset supported by our incomplete solver and describe its operation in detail.

3.1 Overview

We assume that the solver is presented with a query q consisting of an expression and a set of constraints, as defined in §2. The result of the solver can either be $[1, 1]$ ('always-true'), $[0, 0]$ ('always-false'), $[0, 1]$ ('true-or-false') or $\perp$ ('unknown'). If $\perp$ is returned, the query could not be solved and a complete solver must be consulted.

As PARTI is an incomplete solver by design, it has some limitations as to which queries it can answer. This is reflected in the operators it can process. For instance, the modulo operator is entirely unsupported, while other operators may be supported only in some cases. Hence, at any moment, when we encounter a part of the query that cannot be processed, PARTI immediately aborts processing the query and returns $\perp$.

The operation of PARTI is divided into two stages: learning and decision. Before we discuss the individual steps in detail in §3.3 and §3.4, we first illustrate the general idea of these stages by processing the following example query:

$$(\texttt{Eq}(\texttt{Add}(x, 2), y),$$
$$\{\ \texttt{Ule}(x, 7), \texttt{Ule}(4, x), \texttt{Ule}(\texttt{Sub}(y, 3), 7)\ \})$$

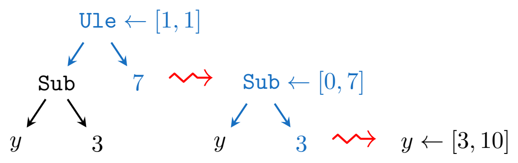

Figure 1: Processing the constraint $\mathtt{Ule}(\mathtt{Sub}(y, 3))$ in the learning stage, where constraints are processed top-down in order to derive information about the solution sets of variables (y in this case.)

3.1.1 *Learning Stage.* During the learning stage, PARTI extracts information about all involved symbolic variables from the constraints. We assign to each variable x a structure $\langle\!|x|\!\rangle$, which represents an approximation of the solution set $[\![x]\!]$. With each processed constraint, $\langle\!|x|\!\rangle$ is refined, reaching an exact representation of the admissible values for x once all constraints have been processed. Hence, until all constraints have been processed, $\langle\!|x|\!\rangle$ remains an approximation of x's solutions $[\![x]\!]$.

For the subset of constraints $\{\mathtt{Ule}(x, 7), \mathtt{Ule}(4, x)\}$, we have $[\![x]\!] = [4, 7]$. While learning the first constraint, we would record $\langle\!|x|\!\rangle = [0, 7]$. Then, when processing the second constraint this approximation would be refined to $\langle\!|x|\!\rangle = [4, \mathtt{max}] \cap [0, 7] = [4, 7] = [\![x]\!]$, where $\mathtt{max}$ corresponds to the maximal value of x's type.

Each constraint is processed top-down, that is, the initial implicit truth value of the expression is recursively pushed down along the AST: As illustrated in Figure 1, the expression $\mathtt{Ule}(\mathtt{Sub}(y, 3), 7)$ must evaluate to $[1, 1]$, as the constraint is known to be true. Therefore, it can be reasoned that its left operand $\mathtt{Sub}(y, 3)$ must evaluate to $[0, 7]$. This in turn implies that y must evaluate to $[3, 10]$. As y is a leaf expression that cannot be taken apart any further, we have learnt that $\langle\!|y|\!\rangle = [3, 10]$ and necessarily $[\![y]\!] = [3, 10]$.

The result of the learning stage is an exact solution set for all relevant variables.

3.1.2 *Decision Stage.* In the second stage, all occurrences of variables in the query expression are substituted with their solution set and the tree is subsequently collapsed, as depicted in Figure 2. Given the constraints over x and y learned above, the expression $\mathtt{Eq}(\mathtt{Add}(x, 2), y)$ is first substituted with $\mathtt{Eq}(\mathtt{Add}([4, 7], 2), y)$. Afterwards, by evaluating the solution sets of the $\mathtt{Add}$ expression and substituting y, we obtain $\mathtt{Eq}([6, 9], [3, 10])$ which in turn may be true as $[6, 9]$ and $[3, 10]$ have a non-empty intersection (e. g., 8) or false as we can choose different representatives from each of the sets (e. g., 6 and 10). This leaves us with the solution set $[0, 1]$, meaning that the query expression can be both true or false. In summary, we fold vertices in the AST in a bottom-up fashion, until the top-most operator is evaluated. A simple linear-time post-order walk through the AST is sufficient to decide the given expression.

The result of the decision stage and the overall result of PARTI is a value in $\{[0, 0], [1, 1], [0, 1], \bot\}$, as discussed above.

3.2 Representation of Solution Sets

After having presented the general overview of the solving algorithm, we now motivate the design of its primary data structure,

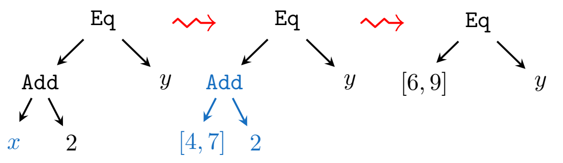

Figure 2: Two steps in evaluating the query expression in the decision stage. The expression will eventually evaluate to $[0, 1]$. As opposed to the learning stage (cf. Figure 1), an expression is processed bottom-up in the decision stage.

multi-intervals, by studying the requirements that arise when trying to represent a simple solution set.

3.2.1 *Signedness.* In our setting, symbolic variables are untyped with regard to their signedness and have a fixed bit-width (cf. §2). Hence, their semantic value depends on the operator using them or their sub-expressions and might therefore implicitly change depending on the context. Assume for instance a one-byte symbolic variable with a solution set with the bit-vector representation $\{11111111_b, 00000000_b\}$. When used in a signed context this can be expressed with the interval $[-1, 0]$ as 11111111_b is interpreted as -1 and 00000000_b as 0 and we have $-1 \leq 0$ and no integer between -1 and 0. However, if in an unsigned context, 11111111_b would be interpreted as 255 for which holds that $255 > 0$ and thus the interval $[255, 0]$ would be ill-formed, while $[0, 255]$ would be incorrect. In an unsigned context, the same solution set cannot be represented as a single interval. The same problem can arise analogously when casting from unsigned to signed.

3.2.2 *Multi-Intervals.* We observe that keeping a single interval is insufficient even for very simple problems. As shown above, sign conversion alone is sufficient to illustrate this limitation, but it also applies to all operators that can result in integer overflows. For instance, the operation $\mathtt{Add}(x, y)$ for $[\![x]\!] = \{253, 254, 255\}$ and $[\![y]\!] = \{1\}$ results in $[\![\mathtt{Add}(x, y)]\!] = \{0, 254, 255\}$ for 8-bit variables in an unsigned context.

A sufficiently expressive representation is to, rather than using a single interval, keep a set of intervals for each variable: a *multi-interval*. However, this representation allows for exponential blowup if the set of supported operations is not chosen carefully. For this reason, we add some further restrictions in the design of the algorithm, and choose to explicitly not implement some operators that in theory could be represented by this data structure.

We encode multi-intervals as sorted sets of pairs of integers such that the individual intervals never overlap. For instance, the unsigned addition depicted above would result in the multi-interval $\langle[0, 0], [254, 255]\rangle_{u8}$. Note that multi-intervals may be typed: We denote the type as subscript, e. g., $u8$, as in this example, indicating an 8-bit unsigned integer. Analogously, an s indicates a signed integer. We omit annotating the type if it is inconsequential. The cardinality $|H|$ of a multi-interval H, denotes the number of intervals it contains, for example $|\langle[10, 50], [100, 200]\rangle| = 2$. We use the previously introduced $\langle\!|x|\!\rangle$ notation in order to refer to x's multi-interval $\langle[\xi_{1,0}, \xi_{1,1}], \dots, [\xi_{n,0}, \xi_{n,1}]\rangle$.

In the following, we present all supported operators in the learning stage (§3.3) and the decision stage (§3.4), discussing both the

Table 2: Cost incurred while processing each supported operator. In the learning stage, we process an expression Op $(e, c) \longleftarrow v$, with n the cardinality of the multi-interval representing v and m the cardinality of the current solution set. In the decision stage, n_i is the cardinality of the multi-interval for the ith argument. Operations marked with an asterisk are only partially supported.

Section	Operator	Output size	Computation
		Learning Stage	
§3.3.1	Eq, Not	≤ 2	$O(1)$
§3.3.2	Ult, Slt, Ule, Sle	1	$O(1)$
§3.3.3	Add, Sub	$\leq n+1$	$O(n)$
§3.3.4	Mul	$\leq n$	$O(n)$
§3.3.5	ZExt*, SExt*	$\leq n$	$O(n)$
§3.3.6	Read*	$\leq \max(n, m)$	$O(n+m)$
		Decision Stage	
§3.4.1	Eq	1	$O(n_1+n_2)$
§3.4.2	Not	1	$O(1)$
§3.4.3	Ult, Slt, Ule, Sle	1	$O(1)$
§3.4.4	ZExt, SExt	$\leq n_1+1$	$O(n_1)$
§3.4.5	Add, Sub	$\leq n_1 \cdot n_2+1$	$O(n_1 \cdot n_2)$
§3.4.6	LShr*, Div*	$\leq \max(n_1, n_2)$	$O(n_1+n_2)$
§3.4.7	Mul*	$\leq \max(n_1, n_2)$	$O(n_1+n_2)$

computational complexity of each operator and the sizes of the resulting multi-intervals. We summarise all operators in Table 2. In §3.5 we end our analysis with an overview of the algorithm's limitations.

3.3 Operations for the Learning Stage

To obtain the admissible multi-interval for each symbolic variable, we extract it from all constraints during the learning stage. Here, each constraint is processed independently, refining the previous approximation. After the last constraint is successfully processed, the extracted multi-interval will exactly, as opposed to approximately, reflect the solution set.

The top-down approach presented here is restricted to binary caterpillar trees [21] containing a single Read. Such binary caterpillar trees, i. e. trees where each branch has one side with depth 1, occur often in practice, as shown in our evaluation. However, if any constraint with a different type of tree is encountered, the learning stage will be aborted and $\perp$ returned.

While we descend such a tree, for each binary operator Op we always have a constant side c and a side with an arbitrary expression e, as well as an assignment of a value v, denoted as Op $(e, c) \longleftarrow v$. We derive the admissible values for e by inspecting c, v, and Op. Initially, every constraint is implicitly *true* at the top-most level, therefore it is of the form $\mathtt{Op}_0(\mathtt{Op}_1(e, c_1), c_0) \longleftarrow v_0$ with $v_0 = [1, 1]$. From c_0 we can then derive a new value v_1 depending on the semantics of $\mathtt{Op}_0$ and recursively process $\mathtt{Op}_1(e, c_1) \longleftarrow v_1$. Finally, we have $e \longleftarrow v_2$ with v_2 derived from v_1, c_1 and the semantics of $\mathtt{Op}_1$.

3.3.1 *Equality and Logical Negation.*

Consider, w. l. o. g., Eq $(e, c) \longleftarrow v$, where v may be either $[0, 0]$, $[1, 1]$, or $[0, 1]$. For $v = [0, 1]$, the remainder of the constraint can be ignored as it is irrelevant whether it holds or not. If v is $[1, 1]$, we can simply continue recursively with $e \longleftarrow c$. If v is $[0, 0]$, the solution set is $\bar{c}$, i. e. a multi-interval containing exactly all values that c does not contain. As the multi-interval $\langle\!|c|\!\rangle$ must be of the form $\langle[c, c]\rangle$, this set-minus operation can be computed in constant time and its size is bound by 2.

The logical negation Not $(e) \longleftarrow v$ can be lowered to Eq $(e, v) \longleftarrow [0, 0]$, as e and v must be one-bit values. Thus, if Not (e) is assigned v, then this is equivalent to e not being equal to v, which in turn is expressed by assigning $[0, 0]$ to Eq (e, v). The same bounds as for Eq apply.

3.3.2 *Comparison.*

Assume a comparison L $(e, c) \longleftarrow v$ with L $\in$ {Ult, Ule, Slt, Sle}. The inverse case L $(c, e) \longleftarrow v$ is analogous. Again, $v = [0, 1]$ may be disregarded as the expression would be a tautology. If we have $v = [1, 1]$, we may recursively continue by learning $e \longleftarrow [a, b]$ where a is the minimum of the respective data type of L, e. g., -128 for an 8-bit signed type. This is because the constraint L (e, c) only imposes some upper bound c on the sub-expression e, while not imposing any lower-bound. For the same reason, b is either c or $c-1$ depending on whether or not the operation is a less-than or less-than-or-equal-to operation. So, $[a, b]$ is the interval matching the type of L and indicating that e's upper bound is c. The size of the resulting multi-interval is trivially exactly 1 and is computed in constant time.

3.3.3 *Addition and Subtraction.*

The subtraction operator is analogous to the addition operator, so we will only investigate Add $(e, c) \longleftarrow v$. Observe that now v is not limited to a boolean value but may be an arbitrarily complex multi-interval. In order to continue the recursion with $e \longleftarrow w$ we must determine a suitable w. Logically, we have $e + c = v \Rightarrow e = v - c = w$, so w follows as the difference between the multi-interval representing v and c. As c is still limited to a constant, the operation $v - c$ can be computed in linear time, by subtracting c from all intervals in v. This may cause at most one interval to span the underflow point and hence be split into two. All other intervals $[a, b]$ in v can trivially be transformed to $[a - c, b - c]$. The expression is then continued to be processed recursively: $e \longleftarrow w = v - c$. Hence, the resulting multi-interval w is computed in linear time and is bound by $|v| + 1$.

3.3.4 *Multiplication.*

By the same logic as with addition and subtraction operators, Mul $(e, c) \longleftarrow v$ is resolved by a division of v by c. Due to the nature of multiplication, some intervals may be merged or dropped, leading to the resulting multi-interval having $|\langle\!|w|\!\rangle| \leq |\langle\!|v|\!\rangle|$. As with Add, Mul has linear complexity.

On the other hand, a Div $(e, c) \longleftarrow v$ removes information from e. Hence, this operation cannot be reversed efficiently in this context, as representing all correct solutions in a multi-interval would potentially lead to a large number of intervals, so Div is not supported by PARTI.

3.3.5 *Type Conversion.*

There are two kinds of type conversions, a *sign cast* changing the signedness of an expression and a *type cast* changing the bit-width of an expression. For each of these, we distinguish two cases. For a sign cast, we can be casting from signed to unsigned or vice-versa, while for a type cast we might be casting down, that is reducing the number of bits, or casting up, that is extending the number of bits. W. l. o. g. we only discuss unsigned type casts as signed type casts are analogous and can be lowered to unsigned type casts. Hence we assume an operation $\mathtt{ZExt}(e) \leftarrow v$, where the expression $\mathtt{ZExt}(e)$ has size k and e has size k'.

- In the case of an upcast, the operator extends the number of bits. When traversing it backwards, the argument v' of the recursive call $e \leftarrow v'$ is restricted in the number of bits, i. e. $k' < k$. This can be easily computed by truncating the intervals that exceed $2^{k'}$ such that they are bound by $2^{k'}$. The resulting multi-interval v' is computed in linear time and $|v'|$ had an upper bound of $|v|$.
- In the case of a downcast, the operator would remove information. Therefore, the multi-interval that would correspond to e cannot be computed from the multi-interval that corresponds to $\mathtt{ZExt}(e)$ without risking an exponential blowup. As an illustrating example, from the information that an expression e of type $u16$ satisfies $\mathtt{Eq}(\mathtt{ZExt}_{u8}(e), 42)$, we can derive the solution set $\{42, 298, 554, \ldots, 65322\}$, which cannot be efficiently represented by a multi-interval. Hence this operation is not supported.

3.3.6 Leaf Operators. Once the tree has been descended to the one non-constant leaf, we have an operation of the form $\mathtt{Read}(x) \leftarrow v$ with v guaranteed to match the width of the read as type casts are always explicit. Hence, if this is the first processed constraint touching variable x, the multi-interval of v can be directly recorded as $\langle\!|x|\!\rangle$, the current approximation of the solution set of x. In case x has already been read by a previously processed constraint, we ensure that the offset and size of the previous read(s) either match the new read's offset and size exactly or read disjoint bytes (cf. §2).

Any partial solution set cannot be extended by future constraints, but only restricted. The result of applying the information from the new multi-interval corresponds to the intersection of the current solution set with the new multi-interval. As a multi-interval is a sorted sequence of intervals, the intersection of two can be computed in linear time.

3.4 Operations for the Decision Stage

We next describe the operations that are employed during the bottom-up decision stage, after the learning stage has finished. A summary of all operators supported in the decision stage, together with the size of the output multi-interval and the associated computational cost is given in the bottom part of Table 2.

3.4.1 Equality. When two symbolic variables a, b are compared for equality, $\mathtt{Eq}(a, b)$, it is insufficient for them to merely share the exact same solution set to return $[1, 1]$. They must both have the same *singleton* solution set, as otherwise they might evaluate to differing values. Hence, only if both multi-intervals are of the form $\langle[k, k]\rangle$ (for some constant k) the result is $[1, 1]$. Conversely, if their non-singleton solution sets have a non-empty intersection, the equality is satisfiable, but not a tautology, and the result is $[0, 1]$. Finally, if they have no solution in common, the result is $[0, 0]$.

3.4.2 Logical Negation. The logical negation $\langle\!|r|\!\rangle = \langle\!|\mathtt{Not}(a)|\!\rangle$ of a multi-interval $\langle\!|a|\!\rangle$ can be computed by inverting the respective truth values. That is, if $\langle\!|a|\!\rangle$ contains 0, then $\langle\!|r|\!\rangle$ must contain 1. And if $\langle\!|a|\!\rangle$ contains 1, then $\langle\!|r|\!\rangle$ must contain 0. This implies that if $\langle\!|a|\!\rangle = [0, 1]$, then $\mathtt{Not}$ is idempotent and the result is $\langle\!|r|\!\rangle = [0, 1]$. As a must have width 1, no value outside of $[0, 1]$ can be contained in $\langle\!|a|\!\rangle$. Clearly, $|\langle\!|r|\!\rangle| = 1$.

3.4.3 Comparison. For a comparison $\mathtt{L}(a, b)$ with $\mathtt{L} \in \{\mathtt{Ult}, \mathtt{Ule}, \mathtt{Slt}, \mathtt{Sle}\}$, we must determine if indeed for all $\alpha \in [\![a]\!]$ and all $\beta \in [\![b]\!]$, we have $\mathtt{L}(\alpha, \beta)$. To solve this problem efficiently, we rely on the representation of multi-intervals and extract the closure $\mathrm{cl}(\langle\!|a|\!\rangle) = [\alpha_{\min}, \alpha_{\max}]$ of $\langle\!|a|\!\rangle$ and the closure $\mathrm{cl}(\langle\!|b|\!\rangle) = [\beta_{\min}, \beta_{\max}]$ of $\langle\!|b|\!\rangle$. The closure $\mathrm{cl}(\langle\!|x|\!\rangle)$ of a non-empty multi-interval $\langle\!|x|\!\rangle = \left\langle[\xi_{1,0}, \xi_{1,1}], \ldots, [\xi_{n,0}, \xi_{n,1}]\right\rangle$ can be computed in constant time by extracting the lower bound of the first interval and the upper bound of the last interval: $\mathrm{cl}(\langle\!|x|\!\rangle) = [\xi_{1,0}, \xi_{n,1}]$. Then $\mathtt{L}(a, b)$ evaluates to $[1, 1]$ if $\mathtt{L}(\alpha_{\min}, \beta_{\max})$, to $[0, 0]$ if we have $\neg\mathtt{L}(\alpha_{\max}, \beta_{\min})$ and to $[0, 1]$ if both conditions are satisfied.

3.4.4 Type Conversion. An extending type cast from the multi-interval $\langle\!|a|\!\rangle = \left\langle[\alpha_{1,0}, \alpha_{1,1}], \ldots, [\alpha_{k,0}, \alpha_{k,1}]\right\rangle_{mi}$ for $m \in \{u, s\}$ to $\langle\!|r|\!\rangle = \left\langle[\varrho_{1,0}, \varrho_{1,1}], \ldots, [\varrho_{l,0}, \varrho_{l,1}]\right\rangle_{mj}$ with $j > i$ is trivial, as none of the intervals are affected and we have $|\langle\!|r|\!\rangle| = l = k = |\langle\!|a|\!\rangle|$ with all intervals being identical. For $j < i$, the resulting solution set is bound to some $\langle[min, max]\rangle$ and those intervals $[\alpha_{i,0}, \alpha_{i,1}]$ with $\alpha_{i,1} < min$ or $max < \alpha_{i,0}$ are dropped from $\langle\!|r|\!\rangle$. Any intervals including the min or max bound are truncated to fit inside $[min, max]$. As each interval is either shrunk or dropped we have $|\langle\!|r|\!\rangle| \leq |\langle\!|a|\!\rangle|$.

A sign cast must comply with two's-complement semantics: For an unsigned-to-signed cast, every interval strictly smaller than the boundary $h = 2^{i-1}$ is preserved, while every interval strictly greater than or equal to h is also preserved but moved left in the sorted list of intervals, to preserve the strict order. Finally, if there exists an interval that includes both h and $h - 1$, it is split in two, one with the upper bound $h - 1$ and one with the lower bound h. A signed-to-unsigned cast is performed analogously, while the boundary h is set at $h = 0$ and intervals are moved to the right, not the left. As up to one interval might be split in two and every other interval is preserved, we have $|\langle\!|r|\!\rangle| \leq |\langle\!|a|\!\rangle| + 1$.

3.4.5 Addition and Subtraction. The plus operator $\mathtt{Add}(a, b)$ is implemented by successively adding individual intervals of b to a akin to a cross product. For this algorithm, we break down the operation to adding two intervals $[\alpha_0, \alpha_1] + [\beta_0, \beta_1] = [\alpha_0 + \beta_0, \alpha_1 + \beta_1] = s$. If s happens to overflow or underflow, it is split in two as described in §3.4.4. Finally, all resulting intervals are merged into one multi-interval. For each pair of intervals from $\langle\!|a|\!\rangle$ and $\langle\!|b|\!\rangle$ we may get up to two intervals. However, as all intervals that lead to an underflow or overflow will be merged, we have in total $|\langle\!|r|\!\rangle| \leq |\langle\!|a|\!\rangle| \cdot |\langle\!|b|\!\rangle| + 1$. Subtraction is analogous to addition.

3.4.6 Non-Rotating Shift. From the two potential shift operators, left-shift and right-shift, the left-shift operator behaves analogously to a multiplication, which we discuss in §3.4.7. A right-shift with a non-constant shift amount may lead to exponential blow-up and thus is not supported. A right-shift by a constant, $\langle\!|r|\!\rangle = \mathtt{LShr}(a, c)$, potentially condenses the multi-interval, i. e. $|\langle\!|r|\!\rangle| \leq |\langle\!|a|\!\rangle|$, so we support it. For instance, the right-shift by 2 of the multi-interval $\langle[0, 9], [11, 20], [30, 40]\rangle_{u8}$ results in the first two intervals merging: $\langle[0, 5], [7, 10]\rangle_{u8}$.

3.4.7 Multiplication. Left shift, and by extension integer multiplication, presents us with a potential exponential blow-up. For instance, consider the multi-interval $\langle[0, 42]\rangle_{u8}$. A left-shift of 1 or a multiplication with 2^1 would result in a solution set containing all the even integers from 0 to 84, i. e., it would have the form $\langle[0,0],[2,2],[4,4],\dots,[84,84]\rangle_{u8}$. However, our problem analysis shows that multiplication with constants of the form 2^i are very common, especially stemming from pointer resolution logic (see §4.1.1). Therefore, we extend multi-intervals in the decision stage to carry a left-shift attribute (LSA). That is, the operation $\texttt{Mul}\,(\langle[0,42]\rangle_{u8}\,,2)$ will result in the multi-interval $\langle[0,42]\rangle_{u7}$ with an LSA of 1. In the general case, this gives us $|\langle\!|r|\!\rangle| \leq |\langle\!|a|\!\rangle|$. Subsequent binary operations like `Add` will then still abort if presented with two operands with differing LSAs, but may succeed when the LSAs are the same. Similarly, inequality operators can apply the LSA when computing the closure without incurring blow-up.

3.5 Limitations

By design, our approach has some limitations. In the learning stage, we cannot process constraints with more than one `Read` as we need to descend on the central path from the root to the singular variable. (On the other hand, we can process multiple reads in the decision stage.) In both stages, some operators are not supported, as they may lead to an exponential blowup.

For every query, PARTI decides on-the-fly whether it can continue to process it or has to abort. This results in an overall complexity that is at worst quadratic in the size of the entire query and exponential in the number of `Add`/`Sub` operators in the expression (cf. Table 2).

4 EVALUATION

We evaluate our approach by comparing a complete state-of-the-art SMT solver, namely Z3, with PARTI in conjunction with the same complete solver. We first validate PARTI's results with those of Z3, finding no discrepancy over the entire course of our several CPU-weeks long evaluation. Hence, the remainder of this section will assess the performance of PARTI.

We implemented the PARTI algorithm directly in the KLEE symbolic execution engine [6] in version 1.4.0, intercepting queries before they are being handed to the back-end solver. This is facilitated by the solver chain of KLEE, which allows to inject an incomplete solver and then automatically consult the next solver in the chain once the injected solver fails to solve the given query. By default, KLEE also invokes the final, complete solver by forking the current process. This is done to enforce the solver timeout, i. e., to terminate the solver, should it consume too much time. We deactivate this mechanism, as forking would create a substantial overhead and bias the evaluation in favour of PARTI. However, that results in solver failures of Z3, which occur quite infrequently, to be propagated up to KLEE, requiring one rerun every approximately 1000 runs.

We choose Z3 as it has finer time-out control built in than KLEE's default SMT solver STP. Running the latter would incur the overhead stemming from process forking or make it impossible to reliably measure the solver time due to extremely long-running queries being prevalent in many runs. However, we also compare the performance of PARTI under both solvers on a smaller benchmark which does not produce such time-out breaking queries in §4.2.3.

Our evaluation was run on an Intel Xeon [23] with 12 threads at 3.4 GHz with 0.25 TB RAM and with hyper-threading disabled. We ran no more than 11 processes in parallel and no swapping occurred during any run.

We structure the evaluation into two main parts: A functional analysis (§4.1) and a performance evaluation (§4.2).

In both cases, each experiment is repeated several times as time measurements always result in noisy data. The exact number of repetitions and the semantics of the depicted confidence intervals is denoted on each figure where it applies. To also minimise noise as much as possible, we use the DFS deterministic searcher, which descends on one execution path, only pursuing other paths after the current one terminates. As this will generate both short and long, simple and complex queries, it provides a good collection of queries. We also deactivated query caches in KLEE to enhance determinism.

4.1 Functional Analysis

As a means of obtaining an insight into the strengths and limitations of PARTI, we first perform a functional analysis of mostly synthetic test programs. Here, we evaluate the termination time for a complete coverage of the input program, imposing no timeout for KLEE. Each run is repeated manifold with PARTI+Z3 and with Z3.

4.1.1 Object Resolution. In KLEE, object resolution for symbolic pointers (including array indexes) depends on a series of comparisons, checking if the pointer value could point inside each of the memory objects on the current path. For instance, when KLEE encounters the access `a[i]`, where i is an unconstrained symbolic index (which could thus refer to other arrays, too), it compares the symbolic value `a + i` against the bounds of all possible objects on that path. This causes a large number of comparison queries, slowing down such operations to the extent that users of KLEE repeatedly complain about this shortcoming.

We analyse this scenario with respect to different array types and lengths in Figure 3. The performance here heavily depends on the support of multiplication in the decision stage: In this scenario, we can identify most queries to have the form $\texttt{L}\left(b, \texttt{Mul}\left(x, 2^i\right)\right)$ where b and i are constants, and x is a symbolic pointer. If we deactivate the support for LSAs (cf. §3.4.7), everything but the speedup for the 8 bit runs drops to 1. With LSAs we observe a huge speedup well above a factor of 200x for all array types—as to be expected, the array size (horizontal axis) has no influence on the execution time. As our example only performs the indexing operation, no query is produced that cannot be solved by PARTI directly.

4.1.2 Worst Case Scenario. We analyse PARTI's behaviour in a worst case scenario. To create an adversarial program, we heavily exploit the multi-interval explosion introduced by an `Add` operation during the decision stage. First, we create a series of N variables v_i and constrain their solution set to the values $\left\{0, 2^i\right\}$ for $i = 1, \dots, N$. Second, we compute the sum over all v_i and multiply with an additional variable with the solution set $\{0, 1\}$. This creates a value with the solution set $\left\{0, 2, 4, 6, \dots, 2^{N+1} - 2\right\}$, which is correctly represented by PARTI but incurs an exponential blowup. Finally, PARTI has to abort computation because of the unsupported non-constant multiplication. For a control experiment, we omit the

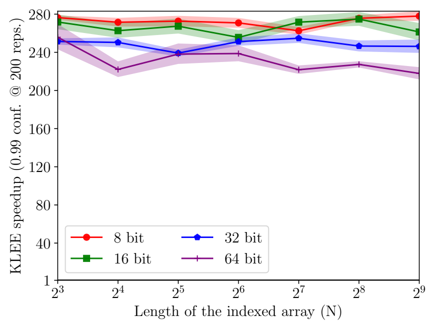

Figure 3: Speedup when indexing an array of size N and various data types with a symbolic 64-bit index. Here, the relative time does not depend on the size or the type of the array, as constant multiplication is supported due to LSAs.

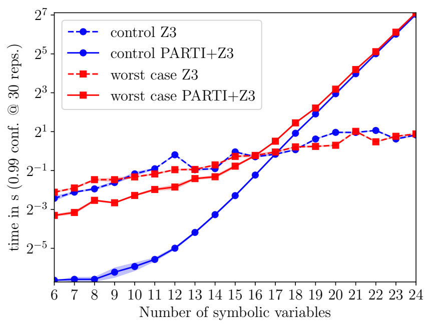

Figure 4: Runtime for the symbolic execution of the worst case and control programs (cf. §4.1.2). PARTI performs well for up to 16 Add expressions. Afterwards the exponential blowup due to additions in the decision stage take over.

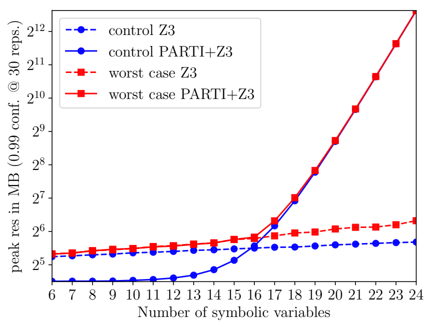

Figure 5: Peak resident memory for symbolic execution of the worst case and control programs. PARTI performs well up to a number of 16 Add expressions, when the number of individual intervals in one multi-interval surpasses 2^{16}.

multiplication, allowing PARTI to succeed instead of having to additionally invoke Z3.

Note that we only analyse one path in this experiment, artificially constraining the solution set of v_i. In a real setting, we would experience path explosion of the magnitude 4^i, with most of the paths not incurring worst case queries in PARTI.

Figure 4 shows absolute runtimes of KLEE for an execution of this worst case program for different values of N, while Figure 5 depicts the peak resident memory for the same experiment. With a small number of symbolic values, the control run shows a noticeable speed up, while the worst case run shows that PARTI can only decide the queries generated while still adding constraints. Around 17 variables, the overhead begins to overtake the speedup and the exponential blowup dominates the overall performance of KLEE, both in the control as well as in the worst case. A very similar behaviour can be observed for the memory consumption.

This also means that while the cardinality of all involved multi-intervals remains below 2^{16}, the overhead of PARTI remains negligible compared to that of KLEE and Z3.

4.1.3 Sorting Algorithms. To get an insight into the efficacy of PARTI in specific scenarios, we investigate its behaviour on a suite of classical sorting algorithms, such as merge sort and quick sort. Our test function sorts an array of concrete data and a fixed number of symbolic entries, placed at random positions within the array. For each of six sorting algorithms, we measure the time until KLEE explores all paths and terminates on its own. Depending on the number of symbolic array entries and the tested algorithm, the time-to-finish may range from fractions of a second up to over an hour. Figures 6 and 7 show that depending on the settings, different algorithms produce more favourable queries for PARTI: For instance, when analysing a single symbolic entry, both the merge and heap sort algorithms show a speedup of around 10x, slowly declining with larger input arrays, while selection sort achieves speedups well above 40.

When analysing the same algorithms with two symbolic entries, instead of one, the speedup drops significantly for all algorithms. While the variance is greatly increased for three algorithms, we still measure speedups ranging from 1.5x to 10x. The different performance in Figure 7 is due to the order in which these algorithms compare the data-to-sort. When an algorithm such as bubble sort touches the first two entries in its very first comparison and they are both symbolic, in the second query there will already be a constraint that contains reads from more than one symbolic value, barring PARTI from processing any subsequent query on that path. Hence, no subsequent query can be answered by PARTI as it will abort each time during the learning stage. So, the speedup depends on the algorithm's memory access patterns and the positions of the symbolic entries.

This demonstrates that the success of PARTI is heavily dependent on the nature of the queries generated by the target program. For instance, if there is only one symbolic value present, the generated constraints will have the structure of a binary caterpillar tree with only one symbolic read, such that the learning phase can succeed. In the second scenario, the symbolic execution introduces a comparison between two symbolic values in every path at some point. Therefore, PARTI can be applied only for a portion of the queries, as can be seen by the starkly reduced speedup in Figure 7 (approx. 1.5x to 10x) compared to Figure 6 (approx. 10x to 50x).

4.2 Performance Evaluation

We evaluate PARTI's performance on a number of real-world programs, to understand the speedup we would see in the wild. Again, we run each experiment with PARTI+Z3 and with Z3.

4.2.1 Coreutils. The GNU Coreutils [15] are widely used for the analysis of the efficacy and efficiency of symbolic execution. We run the Coreutils suite version 8.29, using 98 individual programs as a benchmark set with a configuration corresponding to [31].

To compare the runtime of KLEE configured with PARTI+Z3 versus just Z3, we first ran the respective utility, e. g., sum, for a fixed amount of time, e. g., 900 s and logged the number of instructions, I, KLEE managed to process in that time. Then, we rerun the experiment, without an imposed time-out, but instruct KLEE

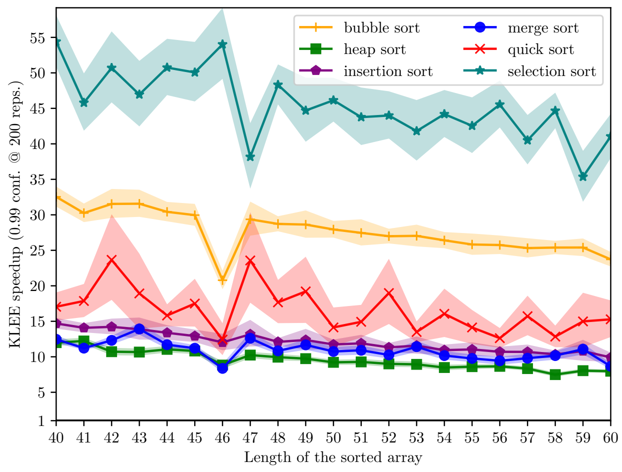

Figure 6: Speedup for sorting algorithms with 1 symbolic field and N-1 concrete fields when using PARTI+Z3 versus Z3 alone. Compared to Figure 7, this experiment offers significantly more opportunities for our solver as there is no comparison between two symbolic values.

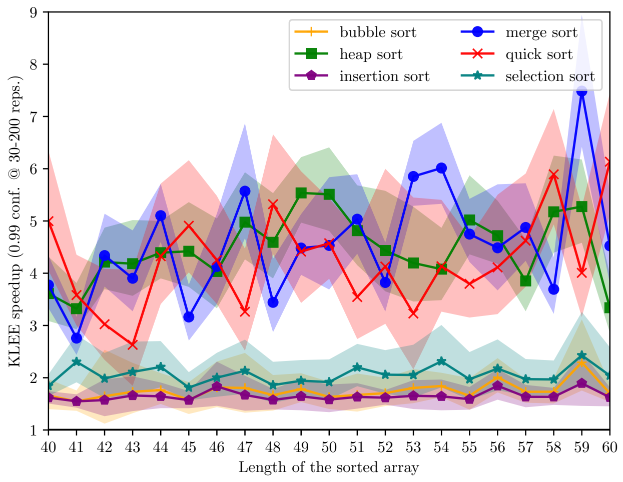

Figure 7: Speedup for sorting algorithms with 2 symbolic field and N-2 concrete fields when using PARTI+Z3 versus Z3 alone. As bubble, selection and insertion sort have very long runtime, they are run with much fewer repetitions, while still achieving adequate confidence in the mean.

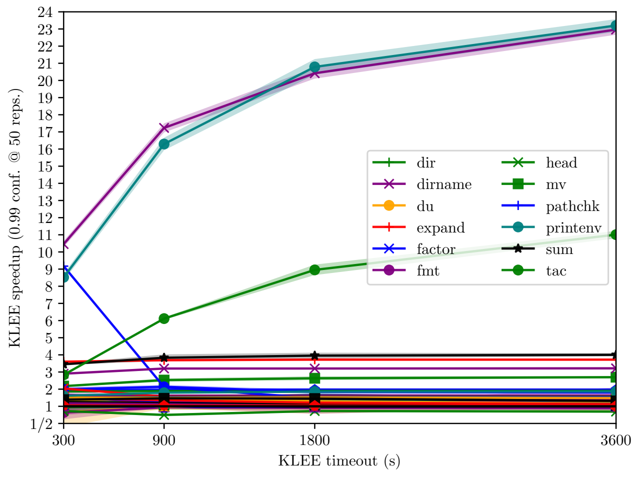

Figure 8: Speedup when executing 98 Coreutils tools. The legend shows those tools with the highest and lowest speedup. The confidence intervals of some tools include portions below the 1x line, indicating that small slowdowns are possible. Many times however, we measure a significant speedup.

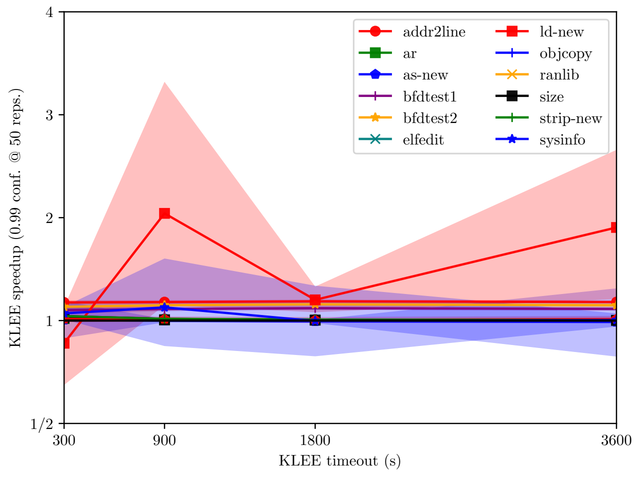

Figure 9: Speedup when executing 17 Binutils tools. The legend shows those tools with the highest and lowest speedup. We observe that the variance is significantly higher than for the Coreutils experiments and PARTI achieves little to no overall speedup.

to terminate after processing *I* instructions. We then record the time KLEE takes to terminate when using PARTI+Z3 versus Z3 alone. Due to non-determinism in KLEE (cf. [26]) the variance of the measured times can vary greatly, especially since we decided to include all Coreutils. Hence, some results may exhibit higher variance and therefore show larger confidence intervals.

Figure 8 shows the measured speedup of PARTI+Z3 versus just Z3. It shows that many programs benefit from a speedup up to 2x, while only seldom falling slightly below the neutral mark of 1x. Around ten programs run at least twice as fast with PARTI, while two, namely dirname and printenv, even climb above a speedup of 20x. So, while in many cases only moderate speedup can be achieved, in some instances KLEE's performance can be improved drastically, from an hour down to a few minutes.

4.2.2 *Binutils.*

For a set of programs with different characteristics, we analyse GNU Binutils [14]—a suite of tools for manipulation and creation of binary files. We use the Binutils suite version 2.28,

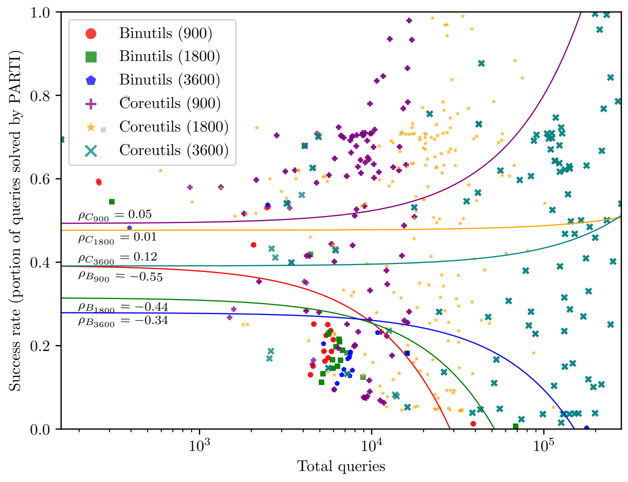

Figure 10: A comparison of PARTI's success rate on the Coreutils and Binutils suites. Each dot represents one run of one tool. The curves represent linear best fits of the data points.

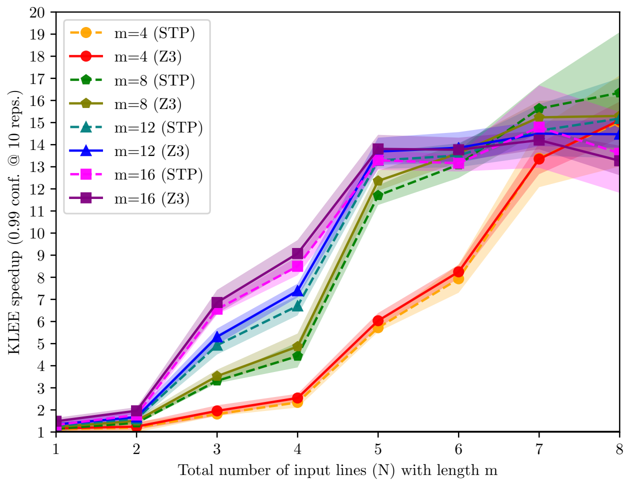

Figure 11: Completing GNU sort `-C` and N alternating concrete and symbolic input lines of fixed length m. The speedup grows both with the length of lines and with the number of lines.

including 17 individual programs. We measure the time for the processing of a pre-determined set of instructions, as with the Coreutils evaluation (cf. §4.2.1).

As depicted in Figure 9, there is no significant speedup for the Binutils suite. The number of exploitable queries here is smaller, as PARTI does not support bitwise operators. These are more prevalent when processing binary data and hence to be expected in greater number in the Binutils suite compared to the Coreutils suite.

This indicates that although PARTI can often result in an advantage, as could be observed with the Coreutils evaluation, where a number of programs experience a speedup of well over 2x, other program structures are less suited for our choice of supported operators. Furthermore, we see a greater variance of execution times in this suite, such that we can only state that there is never a slowdown worse that 1.5x and never a speedup better than 3x.

4.2.3 GNU sort. We analysed GNU sort from Coreutils in more detail. We pass a partially symbolic file and instruct sort to check whether the file is already sorted. Attempts to actually sort a symbolic file with KLEE, whether with or without PARTI, resulted in hundreds of GB of RAM being used within minutes, preventing a proper investigation of anything but the most trivial cases.

We tested `sort -C` on a file with a total of N alternating symbolic and random concrete lines of uniform length m, showing our findings in Figure 11 for values of $m = 4, 8, 12, 16$.

Note that the complexity of the generated queries grows with the length of the symbolic lines, as more bytes need to be compared. In this experiment, we observe that the more complicated the queries are for different values of m, the better PARTI performs relative to Z3. In addition, the more queries generated by symbolic execution, the more queries are sped up, yielding an approximately linear speedup, as all queries can be solved by PARTI in this setup. Therefore, especially in settings with a significant amount of queries, PARTI's ability to quickly resolve some queries results in a high overall speedup. Further, the overall speedup increases with the length of the file's lines, as the complexity of the queries increases.

For this experiment, we also investigated the performance gains between using Z3 and STP as the baseline solver. The results show that with .99 confidence no difference can be identified between the two solvers, even though the plotted average appears to indicate a slight advantage for STP.

4.2.4 Success Rate. The total execution time is most relevant to the user of a constraint solver or symbolic execution engine. However, it provides no information about the ratio of queries that can be answered by PARTI. Figure 10 shows the distribution of the success rate on the Coreutils and Binutils suites with respect to the total number of queries issued by the respective tool.

This distribution matches the speedup we see in §4.2.1 and §4.2.2, seeing a significantly smaller success rate for most of the Binutils tools and a wide spread for the Coreutils tools. Additionally, we can observe that there appears to be no significant relation between the number of Coreutils queries and the success rate of solving them. However, the negative correlation for the Binutils explains the ineffectiveness on that suite: With more queries and a more complex program issuing unsupported operations, fewer queries fall in the supported set, and only short runs are sped up. The success rate of the GNU sort evaluation in §4.2.3 is not included in Figure 10 as it reaches 1 for all runs.

5 RELATED WORK

As discussed in the introduction, our approach is similar in spirit to other constraint solving optimisations employed by modern symbolic execution engines. For instance, prior work has used caching of satisfiability queries [7] and counterexamples [6, 33, 34], expression simplifications [7], logical implications [6, 24] or rewriting of complex array constraints [27] to either avoid calling the SMT solver or call it with simpler queries. Similar to these approaches, PARTI aims to speed up a certain class of constraint solving queries.

Many SMT solvers for QF_ABV are built on top of SAT solvers, and may perform theory-specific optimizations before bit-blasting the query into a SAT formula. For instance, the STP solver [16] performs optimizations such as arithmetic expression simplifications and array-based refinement before bit-blasting the formula to SAT. Solvers like Z3 [10, 11] or Yices [12, 13] also use a conjunction of various theory solvers together with the SAT solver.

Lazy solvers have also been proposed by Bruttomesso et al. [5], which proposes a layered solver design, MathSAT. It uses various rewriting rules to simplify a given query, applying faster passes which support a smaller set of problems before resorting to more complex ones. The second layer is similar to PARTI, as it employs a number of deduction rules pertaining to the properties of various operators. Contrary to PARTI, however, these apply to relations between different variables, such as the transitivity of <, and yields not a definite solution but a simplified query. Similar simplifications are also part of KLEE's internal query optimisation.

Hadarean et al. [20] discuss a staged solver that leverages equality, inequality, and bit-blasting theories with a core solver. Similarly to PARTI, all solvers in this chain are incomplete with respect to SMT but can solve subsets in polynomial time and are called in order of their complexity, beginning with the fastest. The theory solvers are called repeatedly from the main solver loop and some, such as the bit-blasting theory solver, may employ a SAT solver.

Approaches like StratEVO [28] concern themselves with the adjustment and selection of solving strategies within a solver such as Z3 [11]. Here, genetic algorithms are employed to evolve the selection process towards the fastest strategies. Conversely, the multi-solver version of KLEE presented in [26] proposes to run a collection of complete SMT solvers in parallel such that differences in their performance can be exploited.

Interval Arithmetic (IA) has been employed for several decades now [2, 3, 19, 22]. IA operates in the domain of real numbers ($\mathbb{R}$), represented by IEEE floating-point machine numbers and was in its infancy applied to provide a better means of avoiding floating-point rounding errors in Prolog [2, 9]. It solves the task of finding the solution set, represented as an interval, of a list of variables given a finite number of constraints. To this end, it repeatedly attempts to approach the solution set by an enclosing of cartesian products of intervals. This is quite different from the approach presented in this paper, as PARTI represents the solution set of each variable as a set of intervals, while IA represents the total solution set as a set of products of singular intervals. This makes the representation more powerful, but also its computation more complex. Additionally, IA's aim is to tackle non-linear problems on floating-point numbers. Although it has been applied to non-negative integral numbers, it is not designed for the behaviour of two's complement integer arithmetic. Solvers like iSAT3 [29, 30] and raSAT [32] employ IA and constraint propagation techniques to sharpen approximations of solution sets. These operate similar to the merging step in the learning stage and the procedure of the decision stage, but as optimisations on approximations of real-value sets lacking multiplication-gaps and overflows.

6 CONCLUSION

In this paper, we approached the challenge presented by the reliance of symbolic execution on SMT solvers and the corresponding bottleneck. The fundamental complexity of the underlying $\mathcal{NP}$-complete problem severely impacts the efficacy with which symbolic execution can analyse programs and detect defects. We demonstrated how our approach improves the performance of solving SMT queries in the context of symbolic execution.

By designing a lightweight incomplete solver, PARTI, we are capable of exploiting the structure of many queries that can be solved efficiently, leaving the remainder for a complete solver to handle. We discussed the theoretical complexity of our solver, and verified empirically that its overhead is negligible. Hence, the overall solver performance is not impaired, while often being improved significantly, exhibiting order-of-magnitude speedups in several cases. For instance, tools from the Binutils suite, such as sysinfo, experience no speedup, while tools from the Coreutils suite, such as dirname and printenv, are sped up by more than 20x.

Our solver relies on a two stage approach, first extracting information about symbolic variables from the given constraints, and second substituting reads from these variables with the extracted information. We find our proposed multi-interval data structure to be a suitable representation of this information as it lends itself to an efficient manipulation of results while consuming very little memory.

Experimental results indicate the viability of this approach in quickly answering queries that can be solved efficiently by our solver, but would require more time to be solved by a complete, state-of-the-art, SMT solver such as STP or Z3. For some Coreutils tools, we observe that practically every single query can be answered by PARTI, yielding a high speedup. In general, while we can observe a number of examples without significant speedups, we often measure consistent and reproducible speedups well above a factor of 2x and up to a factor of 20x.

ACKNOWLEDGMENTS

This research has received funding from the European Research Council under the EU's Horizon2020 Framework Programme / ERC Grant Agreement no. 647295 (SYMBIOSYS) and from the EPSRC under the grant EP/L002795/1.

REFERENCES

[1] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF–SE: A Symbolic Execution Extension to Java PathFinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, pages 134–138. Springer Berlin Heidelberg, 2007.

[2] Frédéric Benhamou, Laurent Granvilliers, and Frédéric Goualard. Interval Constraints: Results and Perspectives. In *In Proceedings of the Joint ERCIM/Compulog NetWorkshop on New Trends in Constraints*, 1999.

[3] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *The Journal of Logic Programming*, 32(1):1 – 24, 1997.

[4] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, pages 174–177. Springer Berlin Heidelberg, 2009.

[5] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, pages 547–560. Springer Berlin Heidelberg, 2007.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, pages 209–224. USENIX Association, 2008.

[7] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, December 2008.

[8] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[9] J. G. Cleary. Logical Arithmetic. *Future Computing Systems*, pages 125–149, 1987.

[10] Leonardo de Moura and Nikolaj Bjørner. Efficient E-Matching for SMT Solvers. In *Proceedings of the 21st International Conference on Automated Deduction (CADE-21)*, pages 183–198. Springer Berlin Heidelberg, 2007.

[11] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340. Springer Berlin Heidelberg, 2008.

[12] Bruno Dutertre. Yices 2.2. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*, pages 737–744. Springer International Publishing, 2014.

[13] Bruno Dutertre and Leonardo De Moura. The YICES SMT Solver. volume 2, pages 1–2, 2006.

[14] Free Software Foundation. Binutils, 2018-07-19. URL: http://www.gnu.org/software/binutils.

[15] Free Software Foundation. Coreutils - GNU core utilities, 2018-07-19. URL: http://www.gnu.org/software/coreutils.

[16] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, pages 519–531. Springer Berlin Heidelberg, 2007.

[17] Patrice Godefroid, Michael Y Levin, and David Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'08)*, volume 8, pages 151–166, 2008.

[18] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[19] Laurent Granvilliers and Frédéric Benhamou. Algorithm 852: RealPaver: An Interval Solver Using Constraint Satisfaction Techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, 2006.

[20] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. A Tale of Two Solvers: Eager and Lazy Approaches to Bit-Vectors. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*, pages 680–695. Springer International Publishing, 2014.

[21] Frank Harary and Allen J. Schwenk. The Number of Caterpillars. *Discrete Mathematics*, 6(4):359 – 365, 1973.

[22] T. Hickey, Q. Ju, and M. H. Van Emden. Interval Arithmetic: From Principles to Implementation. *J. ACM*, 48(5):1038–1068, 2001.

[23] Intel Corporation. Intel®Xeon®Processor E5-2643 v4, 2018-07-19. URL: http://ark.intel.com/products/92989/.

[24] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*, pages 177–187. ACM, 2015.

[25] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, pages 609–615. Springer Berlin Heidelberg, 2011.

[26] Hristina Palikareva and Cristian Cadar. Multi-solver Support in Symbolic Execution. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, pages 53–68. Springer Berlin Heidelberg, 2013.

[27] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2017)*, pages 68–78, 2017.

[28] N. G. Ramírez, Y. Hamadi, E. Monfroy, and F. Saubion. Evolving SMT Strategies. In *Proceedings of the 28th International Conference on Tools with Artificial Intelligence (ICTAI'16)*, pages 247–254, 2016.

[29] Karsten Scheibler and Bernd Becker. Implication Graph Compression inside the SMT Solver iSAT3. In *MBMV*, pages 25–36, 2014.

[30] Karsten Scheibler and Bernd Becker. Using Interval Constraint Propagation for Pseudo-Boolean Constraint Solving. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD'14)*, pages 32:203–32:206, 2014.

[31] KLEE Team. OSDI'08 Coreutils Experiments, 2018-07-19. URL: https://klee.github.io/docs/coreutils-experiments/.

[32] Vu Xuan Tung, To Van Khanh, and Mizuhito Ogawa. raSAT: an SMT solver for polynomial constraints. *Formal Methods in System Design*, 51(3):462–499, 2017.

[33] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 58:1–58:11. ACM, 2012.

[34] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Memoized Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*, pages 144–154. ACM, 2012.