Expression Reduction from Programs in a Symbolic Binary Executor

Anthony Romano and Dawson Engler

Stanford University

Abstract. Symbolic binary execution is a dynamic analysis method which explores program paths to generate test cases for compiled code. Throughout execution, a program is evaluated with a bit-vector theorem prover and a runtime interpreter as a mix of symbolic expressions and concrete values. Left untended, these symbolic expressions grow to negatively impact interpretation performance.
We describe an expression reduction system which recovers sound, context-insensitive expression reduction rules at run time from programs during symbolic evaluation. These rules are further refined offline into general rules which match larger classes of expressions. We demonstrate that our optimizer significantly reduces the number of theorem solver queries and solver time on hundreds of commodity programs compared to a default ad-hoc optimizer from a popular symbolic interpreter.

1 Introduction

The importance of program reliability, robustness, and correctness, has fueled interest for automated, unassisted program analysis and bug detection. As bug finding systems improve, they find complex, deep bugs which are hard to explain and expensive to discover. It is often difficult or impractical to confirm these error reports by hand; instead, test cases establish soundness by serving as a certificate against false positives. Likewise, amortizing the cost of the analysis process, so bugs may be found in the first place, is subject to considerable study.

Symbolic execution is a popular technique [17] for sound, automated test-case generation. These test cases are created with a goal of finding paths to bugs or interesting program properties in complicated or unfamiliar software. Conceptually, variant data (e.g., file contents) in a program is marked as symbolic and evaluated abstractly. When the program state reaches a control decision based on a symbolic condition, a satisfiability query is submitted to a theorem prover backed solver. If the symbolic condition is satisfiable, but not valid, the state is forked into two states, and a corresponding predicate is made into a *path constraint* which is added to each state's *constraint set*. Solving for the state's constraint set creates an assignment, or test case, which follows the state's path.

The convenience of applying dynamic analysis to unmodified program binaries led to the development of symbolic binary execution. Under symbolic binary execution, compiled executables are symbolically evaluated as-is; there is no need

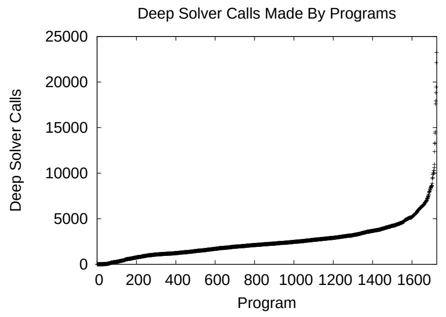

Fig. 1: Total calls into external SMT solver.

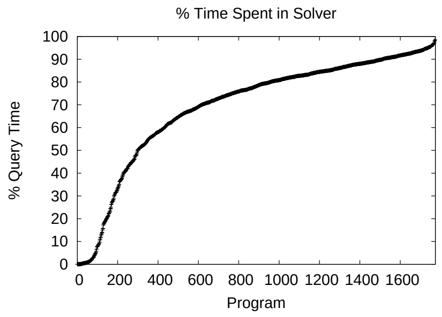

Fig. 2: Query time out of total time.

for recompilation, annotations, special linking, or limited languages. This contrasts with older systems [4, 12, 18, 19] which were built to process source code or a metadata-rich byte code. A typical system pairs a symbolic interpreter with a dynamic binary translation front-end [5, 10, 13, 20, 15]. The binary translator converts a program's machine code into instructions for the symbolic interpreter which proceeds to run the program.

Shifting from specialized symbolic interpretation to general binary translation imposes a performance challenge. First, compilers target machine architectures. Fast code on hardware may lead to the interpreter generating suboptimal solver queries. Additionally, translation introduces an intrinsic overhead from resource mismatches. For instance, register access may be translated into an intermediate value access plus an access into a region of memory representing a register file. Furthermore, type information, control structures, and other metadata from the source code, which is useful for inferring execution traits, is often unavailable at the binary level. Finally, simple ad-hoc tuning fails to scale to the variety of programs, compilers, and optimization configurations in the wild.

In practice, solver requests dominate symbolic execution running time. Figure 2 illustrates query overhead; of several hundred programs after running five minutes, 80% spend more time solving for satisfiability than dispatching instructions. Hence, reducing or eliminating solver calls can be expected to yield gains.

A typical symbolic execution session submits thousands of solver queries. Figure 1 shows the total number of external calls made to the solver for the same programs over a five minute period. Calls to the solver are frequent since they are used to determine branch satisfiability for symbolic conditionals (e.g., whether a state should fork). At an average rate of 500 queries per minute on a modest machine, there is ample opportunity to optimize expressions.

Expressions from symbolic evaluation are redundant and suitable for optimization. For a simple example, a loop that increments a symbolic expression x every iteration produces the bulky expression $(+ \ldots (+ (+\ x\ 1)\ 1) \ldots 1)$ which should fold into a svelte $(+\ x\ c)$. Binary code exacerbates the problem because translation overhead leads to more operations and hence larger expressions.

We propose an expression optimizer which learns reduction rules from symbolic evaluation of binary programs. During the learning phase, rules are created on-demand by symbolic program evaluation and validated using the theorem prover from the symbolic interpreter. These rules are further processed into generalized rules suitable for subsequent symbolic execution with no solver overhead.

We implement and evaluate the expression optimizer in a symbolic binary executor, KLEE-MC, an unassisted program error detector which finds hundreds of bugs in third party binaries. The optimizer is evaluated at scale with over two thousand commodity programs from a stock desktop Linux system. Rules are collected during a brief learning period and are shown to lessen the total number of queries dispatched during symbolic execution. Furthermore, rules improve running and solver time by at least 10% on average over the baseline interpreter.

2 Symbolic Execution Expressions

Symbolic expressions are a byproduct of symbolic execution. A program is symbolically executed by marking its inputs as symbolic and evaluating abstractly. This evaluation emits *expressions* as the result of arithmetic and bitwise operations on symbolic data.

The symbolic executor, KLEE-MC, is a machine code extension of the KLEE [4] symbolic LLVM interpreter. KLEE-MC simulates a program from the host machine by dynamically translating a snapshot's machine code into LLVM for the KLEE interpreter. First, a process snapshot is built from a running program binary through data gleaned from the operating system's debugging facilities (e.g., `ptrace`). KLEE-MC loads the snapshot where it is on-demand translated into LLVM operations. Machine code is translated by VEX [16] into VEX-IR super blocks (i.e., multiple exit basic blocks) which model the target machine architecture as operations on memory, registers, and control flow. Each super block is rendered into unary LLVM functions of the form $f : \mathit{RegisterFileAddress} \rightarrow \mathit{JumpAddress}$. Finally, KLEE-MC loops, translates code on-the-fly, interprets the LLVM, and models a symbolic Linux system call interface.

A mix of concrete values and symbolic expression data are manipulated by evaluating LLVM operations. LLVM operations are those defined by the LLVM IR, such as arithmetic, logical, and memory operations, as well as a handful of specialized LLVM intrinsics. Expressions are a subset of the SMTLIB [3] language, but operators will be written in shorthand notation when convenient. For instance, the expression to add two bit-vector expressions a and b is $(+\ a\ b)$.

Large, redundant expressions negatively influence performance. A large expression slows query serialization to the solver and is costly to evaluate into a constant on variable assignment. Expanded tautologies (e.g., $(x \vee \neg x)$), or expressions that evaluate to one value for all interpretations, lead to cache pollution and unnecessary calls to the expensive theorem prover. Intermediate solvers which rely on the syntactic structure suffer as well. Worse, the expression may linger as a path constraint, affecting future queries through the constraint set.

There are two strategies for shedding expression bloat: optimization and concretization. Expression optimization applies sound identities to reduce an expression to fewer terms. For instance, the bit-wise `or` expression (`or` $0\ x$) is identical to x. The drawback is vital identities are program-dependent, and thus, unpredictable. Alternatively, concretization of symbolic terms reduces expressions to constants but at the cost of completeness; x becomes $\{c\}$ instead of $x \subset \mathbb{N}$.

The original KLEE interpreter addressed optimization and concretization. A hand-written optimizer reduces excess terms from common redundancies. Implied value concretization handles concretization; path constraints are inspected to find and concretize variables which are constant for all valid interpretations.

We consider a problem of expression minimization. We define the expression node minimization problem as follows: given an expression e the minimized expression e' is such that for all valid (never unsatisfiable) expressions $(=\ e\ e_i)$, the number of nodes in e', $|e'|$, satisfies $|e'| \leq |e_i|$. It is worth noting e' is not unique under this definition. To solve this we define an ordering operator on expressions $\leq$ where $e_i \leq e_j$ when there are fewer nodes, $|e_i| < |e_j|$, or by lexical comparison, $|e_i| = |e_j| \wedge lex(e_i) \leq lex(e_j)$. Uniqueness is given by the expression minimization problem of finding e' where $e' \leq e_i$.

A minimizing optimizer has several advantages. Theoretically, it is bounded; reductions are only applied until the expression stops shrinking. Implied value concretization, a method to use path constraints to replace variables (e.g., the constraint $(=\ x\ 1)$ replaces x with 1 in the program state), discerns implications better from smaller expressions. If a conditional expression reduces to a constant, a solver call may be avoided. Conditionals which are not constant benefit in the solver as well, such as through more accurate independence analysis.

3 Rules from Programs

Expressions minimizations form a *reduction relation*, $\rightarrow$. The theory of contractions is a classic formalization for converting terms in lambda calculus [6]. This theory was developed further in abstract rewriting systems as reduction relations under the notion of confluence [11]. We use reduction relations as a theoretical framework for reasoning about properties of the expression rewriting system.

The problem of discovering elements of $\rightarrow$ is handled with a database of reducts. The database, referred to as the EquivDB, is globally populated by expressions made during symbolic execution of binary programs. As expressions are created, they matched against the EquivDB with assistance from the theorem prover to find smaller expressions with equivalent semantics. If equivalent, the expression reduces to the smaller expression and is related under $\rightarrow$.

Information about $\rightarrow$ is maintained as a set of rewrite rules. Each rule has a *from-pattern* and a *to-pattern* which describe classes of elements in $\rightarrow$ through expression templates. Once shown to be sound by the theorem prover, rules are used as the expression optimization directive format in the symbolic interpreter.

3.1 Reductions on Expressions

A reduction from one expression to another is cast in terms of reduction relations. A reduction relation is used to convert one λ-term to another λ-term. We omit the trivial proof of the existence of the correspondence from expressions to λ-terms, $\Lambda_E : \mathit{Expr} \to \Lambda$, but note all expressions for our purposes are in the set of closed sentences Λ^0. All variables are bound; arguments to $\Lambda_E(e)$ are De Bruijn indexes of 8-bit symbolic array read operations (`select` in SMTLIB) from e.

The reduction relation $\to$ is defined as the binary relation

$$\to = \{(\Lambda_E(e), \Lambda_E(e')) \mid (e, e') \in \mathit{Expr}^2\ ,\ e' \leq e\ \wedge\ (=\ e'\ e)\}$$

The immediate reduction relation $\twoheadrightarrow$ is defined as

$$\twoheadrightarrow = \{(e,\ e') \in \to\ \mid \forall (e, e'') \in \to\ .\ \Lambda_E^{-1}(e') \leq \Lambda_E^{-1}(e'')\}$$

An expression is reduced by $\to$ through β-reduction. The reduction $a \to b$ is said to reduce the expression e when there exists an index assignment σ for $\Lambda_E(e)$ where $\Lambda_E(e)_\sigma$ is syntactically equal to a. β-reducing b with the terms in e substituted by σ on matching variable indices yields the shorter expression $[a \to b][e]$. The new $[a \to b][e]$ is guaranteed by referential transparency to be semantically equivalent to e and can safely substitute occurrences of e.

For instance, consider the following equivalent 8-bit expressions e and e'.

$$e \equiv (\texttt{bvand}\ \texttt{bv}128[8]\ (\texttt{sign\_extend}[7]\ (=\ \texttt{bv}0[8]\ (\texttt{select}\ a\ \texttt{bv}0[32]))))$$

$$e' \equiv (\texttt{concat}\ (=\ \texttt{bv}0[8]\ (\texttt{select}\ a\ \texttt{bv}0[32]))\ \texttt{bv}0[7])$$

Expressions e and e' are semantically equivalent; both return the value 127 when index 0 of the symbolic array a is zero. Applying Λ_E yields λ-term functions,

$$\Lambda_E(e) \equiv (\lambda x_1.(\texttt{and}\ 128\ (\texttt{sgnext7}\ (=\ 0\ x_1))))$$

$$\Lambda_E(e') \equiv (\lambda x_1.(\texttt{concat}\ (=\ 0\ x_1)\ 0_7))$$

Any expression syntactically equivalent to e up to the variable term $(\texttt{select}\ a\ \texttt{bv}0[32])$ is reducible by $\Lambda_E(e) \to \Lambda_E(e')$. For instance, suppose the variable term were replaced with $(*\ 3\ (\texttt{select}\ b\ 1))$. Applying the reduction rule with a β-reduction replaces the variable with the new term,

$$\Lambda_E(e')(*\ 3\ (\texttt{select}\ b\ 1)) \to_\beta (\texttt{concat}\ (=\ 0\ (*\ 3\ (\texttt{select}\ b\ 1)))0_7)$$

Finally, the new λ-term becomes an expression for symbolic interpretation,

$$(\texttt{concat}\ (=\ \texttt{bv}0[8]\ (\texttt{bvmul}\ \texttt{bv}3[8]\ (\texttt{select}\ b\ \texttt{bv}1[32]))\ \texttt{bv}0[7])$$

.

3.2 EquivDB

Elements of $\to$ are discovered by observing expressions made during symbolic execution. Each expression is stored to a file in a directory tree, the EquivDB, to facilitate a fast semantic lookup of expression history across programs. The stored expressions are shorter candidate *reducts*. The expression and reduct are checked for semantic equivalence, then saved as legal reduction rule.

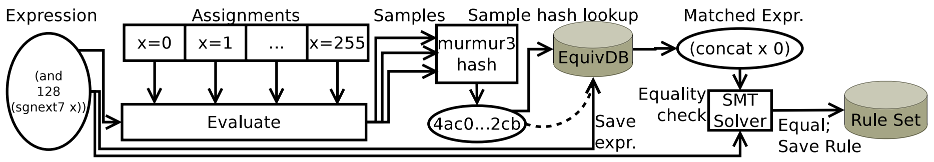

Fig. 3: Storing and checking an expression against the EquivDB.

Generating Candidate Reducts The expressions generated by programs are clues for reduction candidates. The justification is several programs may have similar local behavior where some constant specialization triggers a compiler optimization, but not for all. A path on symbolic execution reintroduces specializations on general code which makes expressions with equivalent semantics to those belonging to the specialized code.

In the rule learning phase, candidate reducts are collected by the interpreter's expression builder, which are submitted to the EquivDB. Only top-level expressions are considered to avoid excess overhead from intermediate expressions which are generated by optimization rewrites during construction. To store an expression into the EquivDB, it is sampled, the values are hashed, and is written to the file path <bit-width>/<number of nodes>/<sample hash>. Entries are capped at 64 nodes maximum to avoid excessive space utilization.

Samples from expressions are found by assigning constant values to all array `select` accesses. The set of array assignments include all 8-bit values, strided non-zero values of up to 17 bytes (i.e., $> 2\times$ the 64-bit architecture word width to reduce aliasing), and strided zero values of up to 17 bytes. The expression is evaluated for each array assignment and the sequence of samples is combined with a fast hashing algorithm [1]. It is worth noting this has obviously poor collision properties; for instance, the 32-bit comparisons ($= \ x$ 12345678) and ($= \ x$ 12345679) would have the same sample hashes because neither constant appears in the assignment set. Presumably, a larger set of assignments would improve the valid reduct hit rate at the expense of additional compute time.

The EquivDB storage and lookup facility is illustrated by Figure 3. At the top of the diagram, an expression from the interpreter is sampled with a set of assignments and the values are hashed. The expression is looked up by the sample hash in the EquivDB and saved for future reference. A lookup match is found and checked against the starting expression for semantic equality. Finally, the equality is found valid and the corresponding rule is stored into the rule set.

Reduction by Candidates Before an expression e is stored in the EquivDB, the learning phase attempts to construct a reduction rule. Based on e's sample hash, the EquivDB is scanned for matching hashes with the same bit-width and fewer terms. Expression equivalence is checked using the theorem prover and, if valid and contracting, a rule is saved and applied to the running program.

A smaller expression is loaded from the EquivDB based on matching sample hash and bit-width. The candidate reduct e^* is parsed from an SMT file in the EquivDB to an expression and assigned temporary arrays for each symbolic read. The temporary arrays in e^* are replaced by index in $\Lambda_E(e^*)$ with the matching index terms from $\Lambda_E(e)$ to get e'. If the reduct is contracting, $e' < e$, and ($= \ e \ e'$)

is valid by the solver, then $(e, e') \in \rightarrow$ and $e \rightarrow e'$ is saved as a rule for future reference. If $e \rightarrow e'$ is valid, the shorter e' is produced instead of e.

```
:extrafuns ((x Array[32:8]))
:formula
(= (ite (= bv0xffffffff00[40] (extract[63:24]
    (bvadd bv0xffffffff00000001[64]
    (zero_extend[56] (select x bv0[32])))))
  bv1[1] bv0[1]) bv0[1])
```

Fig. 4: A translation check query. The to-expression $\texttt{0xffffffff00}_{40}$ is compared with the from-expression $extract(63, 24, \texttt{0xffffffff00000001}_{64} + x_8)_{40}$. Negation of the translation equality is unsatisfiable, hence the translation is valid.

An example query for a candidate rule validity check is given in Figure 4. The equality expression on an arithmetic expression (e) and a constant value (e') is sent to the solver and a "sat" or "unsat" string is returned. To determine soundness of the rule $e \rightarrow e'$ with one query, the equality is negated so that validity is given by unsatisfiability.

Care is taken to handle several edge cases. It is often the case e is equivalent to a constant; the sample hash predicts e is constant if it takes only one value for the entire sample set of assignments. The predicted constant value is used as a first candidate reduct to avoid the overhead of storing constants in the EquivDB. To avoid infinite recursion, expressions built in the solver are queued for rule checking until the interpreter builds an expression outside the solver. For reliability, if the solver fails or takes too long to check a rule's validity, then the query is aborted and e is returned for symbolic interpretation.

3.3 Rewrite Rules

Reductions in $\rightarrow$ are represented through rewrite rules. Every rewrite rule can be traced back to a primordial expression equality from EquivDB cache. To keep the number of rules within reason, each rewrite rule covers a class of reductions with expression template *patterns*. These rules direct expression optimization and are managed through persistent storage.

Once a candidate rule has been checked by the EquivDB, all further processing is performed solely on rules, with no need to invoke the interpreter. The equivalent expressions e, from the interpreter, and e', from the EquivDB, are converted into a from-pattern a and to-pattern b which are combined to make a rule $a \rightarrow b$. The from-pattern is later generalized to a wider range of expressions.

Patterns in rules are flattened expressions with extra support for labeling replacement slots. Symbolic array reads are labeled as 8-bit slots which correspond to the variables from the expression transform Λ_E. These slots are used in pattern matching which is described in Section 4 and are important for correspondence between the from-pattern and to-pattern. Dummy variables, which are only matched on expression width, are supported and introduced through rule generalization (Section 6). Likewise, there are constrained slots for constants, also introduced through generalization, which matches when a constant is a satisfying assignment for an expression bundled with the rule.

Rules are sent to persistent storage with a binary format and may be serialized into files and read in by the interpreter. Serialization flattens expressions into patterns by a pre-order traversal of all nodes. On disk, each rule is given a header which lets the rule loader gracefully recover from corrupted rules, specify version features, and overlook deactivated tombstone rules.

Manipulating rules, such as for generalization or other analysis, often requires expression *materialization*. Rules can be materialized into a validity check or by individual pattern into expressions. The validity check is a query which may be sent to the solver to verify that the relation $a \to b$ holds. Each expression is assigned independent temporary arrays for symbolic data to avoid assuming properties from state constraint sets.

4 Rule-Directed Optimizer

The optimizer applies rules to a target expression to produce smaller, equivalent expressions. A set of reduction rules is loaded from persistent storage at interpreter initialization for the rule-directed expression builder. There are two phases for applying reduction rules when building a target expression. First, efficient pattern matching finds the arguments for a β-reduction from a rule's from-pattern to a target expression. When a rule match is found, the β-reduction applies the arguments to the rule's to-pattern to make a smaller expression.

4.1 Pattern Matching

Over the length of a program path, a collection of rules is applied to every expression. The optimizer analyzes every expression seen by the interpreter, so finding a rule must be fast and never call to the solver. Furthermore, thousands of rules may be active at any time, so matching rules must be efficient.

The optimizer has three ways of finding a rule r which reduces an expression e. The simplest, linear scan, matches e against one rule at a time until reaching r. Hashing a skeletalization of all from-pattern materializations and the expression strictly matches r by hash table. Flexible matching against the entire rule collection, which includes subexpression replacement, is handled with a backtracking trie which is traversed in step with e. Both skeletal hashing and the trie are used by default for lookup to mitigate unintended rule shadowing.

Linear Scan. The expression and from-pattern are scanned and pre-order traversed with tokens checked for equality. Every pattern variable token assigns its label to the current subexpression and skips its children. If a label has already been assigned, the present subexpression is checked for syntactic equivalence to the labeled subexpression. If distinct, the variable assignment is inconsistent and the rule is rejected. All rules must match through linear scan; it is always applied after rule lookup to double-check the result.

Skeletal hashing. Expressions and from-patterns are skeletal hashed [8] by ignoring `select` subexpressions. A rule is chosen from the set by the target expression's skeletal hash. The hash is invariant with respect to indexing and leads to ambiguity; matching by hash may lead to a rule won't reduce the expression. Lookup is made sound by checking a potential match with a linear scan.

Backtracking Trie. The tokenization for every from-pattern is stored in a trie. The expression is scanned and the trie matches on traversal. As nodes are matched to pattern tokens, subexpressions are collected to label the symbolic read slots. Choosing between labeling or following subexpressions is tracked with a stack and is backtracked on match failure. On average, an expression is scanned about 1.1 times, so the cost of backtracking is negligible.

A majority of expressions are never optimized. Since few expressions will match against the rule collection, rejected expressions are fast-pathed to avoid unnecessary rule lookups. Constants are the most common type expression and are universally normal forms; they are passed through as-is by the optimizer. Non-constant expressions are hashed and only accepted if no expression with a matching hash missed in the rule set; misses are memoized.

4.2 β-reduction

Given a rule $a \to b$ which reduces expression e, a β-reduction contracts e to the b pattern structure. Subexpressions labeled by a on the linear scan of e serve as the variable index and term for substitution in b. There may be more labels in a than variables in b; superfluous labels are useless terms. On the other hand, more variables in b than labels in a indicates an inconsistent rule. To get the β-reduced, contracted expression, the b pattern is materialized and its `select`s on temporary arrays are substituted by label with subexpressions in e.

5 Building Rule Sets

Rules are gathered into rule sets for offline refinement. Rule sets are managed with a special program called `kopt` which uses expression and solver infrastructure from the interpreter. The `kopt` program checks rules for integrity and builds new rules by reapplying the rule set to pattern materializations.

A rule set is checked for integrity at several points. Without integrity, the expression optimizer could be directed by a faulty rule to corrupt the symbolic computation. Worse, if a bogus rule is used to make more rules, such as by transitive closure, the error propagates, poisoning the entire rule set.

Additional processing refines a rule set's translations when building expressions. When all rules are applied at once, a rule's materialization may not match its pattern, which can introduce expression permutations unknown to the rule set. This is handled by creating rules to fill out the rule set's transitive closure until reaching a fixed point. Further, to-patterns are normalized to limit production of distinct expressions with equivalent semantics to improve rule efficiency.

5.1 Integrity

Rules are only applied to a program when they are verified to be correct by the solver. Output from the learning phase is marked as pending and is verified by the solver independently. Rules are further refined past the pending stage into new rules which are checked as well. At program run time the rule set translations can be cross-checked against the baseline builder for testing composition.

Rule sets are processed for correctness. A rule set is loaded into `kopt` and each rule is materialized into an equivalence query. The external theorem prover verifies the equivalence is valid. Syntactic tests follow; components of the rule are constructed and analyzed. If the rule was not effective when materialized through the optimizer, it is thrown out.

A rule must be contracting. When expressions making up a rule are heavily processed, such as serialization to and from SMT or rebuilding with several rule sets, the to-expression may have more nodes than the from-expression. In this case, although the rule is valid, it is non-contracting and therefore removed. The rule can be recovered by reversing the patterns and checking validity.

As an end-to-end check, rule integrity is optionally verified at run time for a program under the symbolic interpreter. The rule directed expression builder is *cross-checked* against the default expression builder. Whenever a new expression is created from an operator $\circ$ and arguments $\bar{x}$, the expression $(\circ\ \bar{x})$ build under both builders for e and e' respectively. If $(=\ e\ e')$ is not valid according to the solver, then one builder is wrong and the symbolic state is terminated with an error and expression debugging information. Cross-checking also works with a fuzzer to build random expressions which trigger broken translations.

5.2 Transitive Closure

Rules for large expressions may be masked by rules from small expressions. Once rules are applied to a program's expressions, updated rules may be necessary to optimize the new term arrangement. Fortunately, rules are contracting, and therefore expression size monotonically decreases; generating more rules through transitivity converges to a fixed point minima.

An example of how bottom-up building masks rules: consider the rules $r_1 = [(+\ a\ b) \to 1]$ and $r_2 = [a \to c]$. Expressions are built bottom-up, so a in $(+a\ b)$ will be reduced to c, yielding $(+\ c\ b)$. Rule r_1 no longer applies since r_2 eagerly modified a subexpression. However, all rules are contracting, so $|(+\ c\ b)| < |(+\ a\ b)|$. Hence, new rules may be generated by applying known rules, then fed back to the system with the expectation of convergence to a fixed point rule set.

New rules are *inlined* as they are observed. For every instance of a pattern materialization not matching the pattern, a new rule is created from the expected pattern. Following the example, r_3 will materialize into $(+\ a\ c)$, so we create a new rule, $r_3 = [(+\ a\ c) \to 1]$.

Convergence in general is affected by the EquivDB. The database may hold suboptimal translations that bubble up into rules through learning. Hence, it is advantageous to flush the equivalences to cut down searches for fixed point rules. On the other hand, the database improves as rules improve, since smaller expressions are stored. Hence, a database of rule derived expressions continues to have good reductions even after discarding the initial rule set.

5.3 Normal Form Canonicalization

Expressions of same size may take different forms. Consider, $(=\ 0\ a)$ and $(=\ 0\ b)$ where $a = reverse(b)$. Both are equivalent and have the same number of nodes but will not be reducible under the same rule because of syntactic mismatch. Instead, a *normal form* condition is imposed by selecting for the minimum of the expression ordering operator $\leq$ on semantic partitions on to-patterns. With normal forms, fewer rules are necessary because equivalent to-patterns materialize to the same literal minimal representation.

The to-pattern materializations are collected from the rule set and partitioned by sample hashes. Each partition P of to-expressions is further divided by semantic equivalence by choosing the minimum expression $e_\perp \in P$, querying for valid equality over every pair $(e_\perp, e)$ where $e \in P$. If the pair is equivalent, the expression e is added to the semantic partition $P(e_\perp)$. Once $P(e_\perp)$ is built, a new $e'_\perp$ is chosen from $P \backslash P(e_\perp)$ and the process is repeated until P is exhausted.

Rules are replaced by their normal forms. Once the to-expressions are partitioned, the rule set is scanned for rules with to-expressions e where there is some $P(e_\perp)$ with $e \in P(e_\perp)$ where $e_\perp \neq e$. The rule's to-pattern is replaced with the to-pattern for $e_\perp$ and the old rule is removed from the rule set file.

6 Rule Generalizations

The class of expressions a rule matches may be extended by selectively relaxing terms in the from-pattern. The process of generalization goes beyond transitive closure by inserting new variables into expressions. Useless subterms are relaxed with dummy variables by subtree elimination. Constants with a set of equisatisfiable values are relaxed by assigning constraints to a constant label.

6.1 Subtree Elimination

Useless terms in from-expressions are marked as dummy variables in the from-pattern through subtree elimination. A rule's from-expression e has its subexpressions post-order replaced with dummy, unconstrained variables. For each new expression e', the solver finds for the validity of $(=\ e\ e')$. If e' is equivalent, the rule's from-pattern is rewritten with e' so that it has the dummy variable.

As an example, let

$$(=\ 0\ (\texttt{or}\ 1023\ (\texttt{concat}\ (\texttt{select}\ 0\ x)\ (\texttt{select}\ 1\ x))))$$

be the from-expression e. Since the `or` term is always non-zero, $e \twoheadrightarrow 0$. Traversal will try to mark the 0, `or`, and 1023 terms as dummy variables but the solver rejects equivalence. The `concat` term, however, can take any value so it is marked as a 16-bit dummy variable `v16` to get the pattern $(=\ 0\ (\texttt{or}\ 1023\ \texttt{v16}))$ which matches any 16-bit term instead of only the `concat` term.

6.2 Constant Relaxation

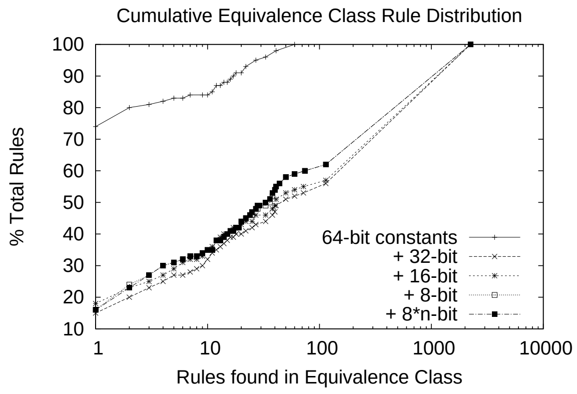

Fig. 5: Acceptance of constant widths on expression equivalence class hits.

A large class of expressions generalize from a single expression by perturbing the constants. In a rule, constant slots serve as constraints on the expression. Consider the 16-bit expression e, `(and 0x8000 (or 0x7ffe (ite (x) 0 1))`. The values of the `ite` if-then-else term never set the 15th bit, so $e \twoheadrightarrow 0$. By marking $0x8000$ as a labeled constant c, this reduction generalizes to the rule `(and 0x8000 (or c (ite (x) 0 1))` where $c <$ `0x8000` is the *constant constraint*, which expands the rule's reach from one to thousands of elements in $\twoheadrightarrow$.

To find candidates for constant relaxation, rules are partitioned by from-pattern expression materialization into constant-free equivalence classes. The constant-free syntactic equivalence between expressions e and e' is written as $e \equiv_c e'$. Let the function $\alpha_c : Expr \rightarrow Expr$ α-substitute all constants with a fixed sequence of distinct free variables. When the syntactic equivalence $\alpha_c(e) \equiv \alpha_c(e')$ holds, then constant-free equivalence $e \equiv_c e'$ follows.

A cumulative distribution of equivalence class sizes in $\equiv_c$ from hundreds of rules is given in Figure 5. Constants in rules are α-substituted with a dummy variable by bit-width from 64-bit only to all byte multiples. Singleton equivalence classes are unique rules that are syntactically unique modulo constants. Likewise, rules belonging to large classes exhibit a high degree of similarity. Aside from admitting more rules total, distribution is relatively insensitive to constant width past 64-bits; few rules are distinct in $\equiv_c$ and one large class holds nearly a majority of rules.

Constants are selected from a rule one at a time. The constant term t is replaced by a unique variable c. The variable c is subjected to various constraints to find a new rule which matches a *set* of constants on c. This generalizes the base rule where the implicit constraint is $(= \ c \ t)$.

Constant Disjunction The simplest way to relax a constant is to constrain the constant by all values seen for its position in a class of rules in $\equiv_c$. A constant is labeled and the constraint is defined as the disjunction of a set of observed values for all similar rules. The resulting rule is a union of observed rules with similar parse trees pivoted on a certain constant slot.

The disjunction is built by greedily augmenting a constant set. The first in the set of values S is the constant c from the base rule. A new constant value v is taken from the next rule and a query is sent to the solver to check if v can be substituted into the base rule over c. If the validity check fails, v is thrown away as a candidate. If v is a valid substitution, it is added to S. When all candidate values from the rule equivalence class are exhausted, the constraint on the labeled constant slot c is $\bigvee_{s \in S}(= \ c \ s)$

Ranges Range constraints restrict a constant to a contiguous region of values. The values for the range $[a, b]$ on the constant substitution x are computed through binary search in the solver. The constant from the base rule c is used as the initial pivot for the search so $c \in [a, b]$ to match the base rule. Starting from c, one binary search finds a from $[0, c]$ and another finds b from $[c, 2^n - 1]$. The constraint $a \leq x \leq b$ is placed on the new rule and the solver verifies equivalence to the from-expression from the base rule.

Bit masks A constant in a rule may only depend on a few bits being set or zeroed, leaving all other bits unconstrained. Ranges on constants only support contiguous ranges, so it is necessary to introduce additional constraint analysis. Constant constraints on a constant x's bits are found by creating a mask m and value c which is valid for a predicate of the form $x \ \& \ m \ = \ c$.

The solver is used to find the mask m bit by bit. Since the base rule is valid, the rule's constant value a must satisfy $a \ \& \ m \ = \ c$. Bit k of the mask is computed by solving for the validity of $(= \ x \ (a \ \& \ 2^k))$ when x is constrained by the base rule. Each set bit k implies bit k of x must match bit k of a.

7 Evaluation

The expression optimizer is evaluated in terms of performance, effects on queries, and system characteristics on two thousand programs. Foremost, rules improve running time and solver performance on average. Total queries are reduced on average from baseline by the optimizer. The space overhead and expression distribution of the EquivDB illustrate properties of the learning phase. Rule effectiveness is measured by number of rules used and rate of sharing.

7.1 Implementation

The expression reduction system was written on top of a modern symbolic execution stack. We used a heavily modified version of KLEE (KLEE-MC), LLVM-3.1, valgrind-3.8.1, and the latest SVN of STP [9]. To attain a degree of fidelity, the symbolic interpreter is cross-checked with the LLVM JIT on replay. For system stability, a separate process solves queries intended for STP.

Component	Lines of Code
EquivDB/Learning	645
Rule Builder	1900
`kopt`	2519
Hand-written Builder	1315

Table 1: Lines of code for expression optimization.

Table 1 shows the lines of C++ code for major components of the expression handling system. Qualitatively, the code for the new optimizer represents a modest effort compared to the ad-hoc version. The largest component is `kopt`, the offline rule analysis program, where the cost of complexity is low. The rule builder, which applies the vetted rules as expressions are built inside the interpreter, is primarily focused on the fast matching trie. The EquivDB learning builder uses the least code since creating candidate rules is relatively simple.

7.2 Test System

Programs All experiments are performed over a set of approximately 2300 programs. The programs are from the system binary directories `/{usr/,}{sbin,bin}` of an up-to-date x86-64 Gentoo Linux system. Programs are breakpointed at the entry point and snapshotted. Every snapshot includes the entire process image from memory, including linked shared libraries, such as the C library `glibc`. Subsequent runs of program reuse the snapshot for reproducibility purposes.

Programs are set to run under the symbolic interpreter for at most five minutes. Each program is given five minutes on one core of an 8-core desktop chip with 16GB of memory. There is minor additional processing and book keeping; overall, one symbolic run of the program set takes slightly more than a day.

Path Replay Two sets of runs are taken for basis of comparison: one with only the ad-hoc optimizer and the other with the rule-directed optimizer as well. The same paths must be followed to give an accurate comparison between the baseline symbolic interpreter and the optimizer. Paths are selected by having the shortest known path to a distinct basic block. Since paths are known a priori, persistent query caches are disabled to avoid distorted times.

KLEE-MC supports two kinds of path replay: concrete test cases and branch paths. A concrete test case is a feasible solution to a path's final constraints which is used to seed symbolic values. A branch path is a list of branch decisions.

Each branch path is a log of taken branch indexes (e.g., true, false) for some completed state. Branch replay reads an entry from the log for every branch decision and directs the replaying state toward the desired path. As an optimization, if a branch replay forks off a state with a branch log which is a prefix for another branch path, the branch path replay begins at the forked state.

Branch path equivalence is not guaranteed between paths with different rules. Mismatched branch decisions arise between distinct rule sets when the interpreter syntactically checks for constant expressions to avoid extra work. A concrete test case, on the other hand, is a semantic interpretation, and therefore insensitive to expression structure. Concrete test cases preserve paths across rule sets so they are used to rebuild branch paths.

7.3 Performance

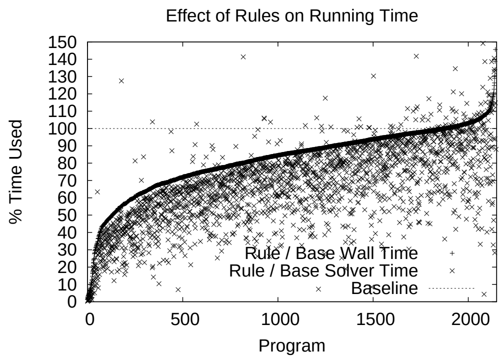

Fig. 6: Sorted percentage run time for expression optimization over baseline.

The effect of rules on running time and solver time is given sorted in Figure 6. Overall, there are significant performance gains made on average with the rule directed optimizer. Additionally, the correlation between run and solver time is evident by solver improvements closely following run time gains.

On average, the optimizer improves performance of the symbolic interpreter on a wide variety of programs. The optimizer improved times by producing shorter expressions and syntactic structures favorable to solver optimizations. The fastest 50th percentile decreases running time by at least 10%. Limited to the fastest 25th percentile, programs see decreased running time of at least 27%.

A few programs do not benefit from the optimizer. Either no improvement or a performance loss were observed in slightly fewer than 13% of all programs. Only five programs (0.2%) took more than 2× of baseline execution time. There is no requirement, however, to run the optimizer, so applications which exhibit a performance penalty with rules can simply go without and retain baseline speed.

As expected, less time is spent waiting on the solver. A 94% majority of the programs spent less time in the solver by using the optimizer. A sizeable 36% of all programs are at least twice as fast in the solver. The 6% minority of programs, like for running time, incurred additional solver overhead. Solver time improvement and running time improvement appear related; only 5% of faster programs had running time decrease more than solver time.

7.4 Queries

The percent change in queries submitted to the solver is shown ordered in Figure 7. On average, the total number of solver queries dispatched for consideration is lower with the optimizer than without. Within the best 50th percentile, at least 17% of queries submitted to the solver were eliminated. In total, fewer queries were dispatched for 87% of the programs. The query histogram in Figure 8 illustrates a shift toward faster queries from slower queries.

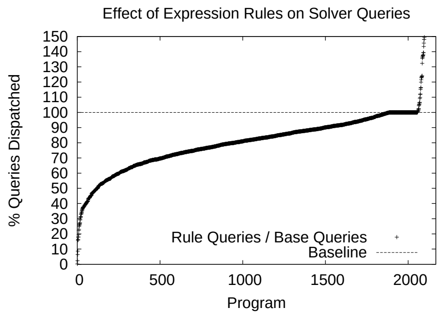

Fig. 7: Percentage of queries submitted when using rule sets.

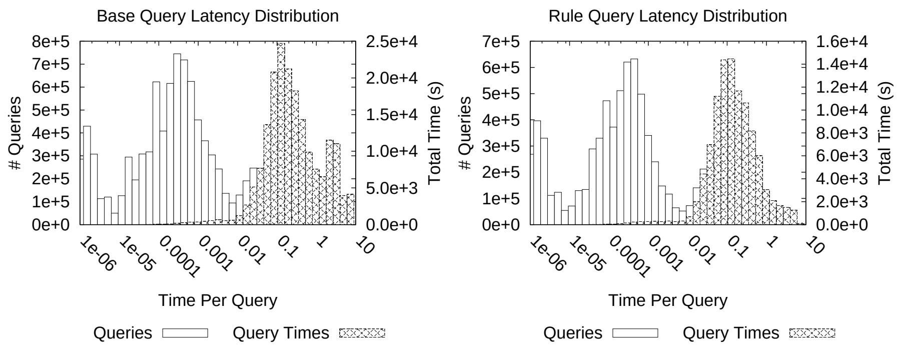

Fig. 8: Query time distribution for baseline and rule test cases.

7.5 EquivDB

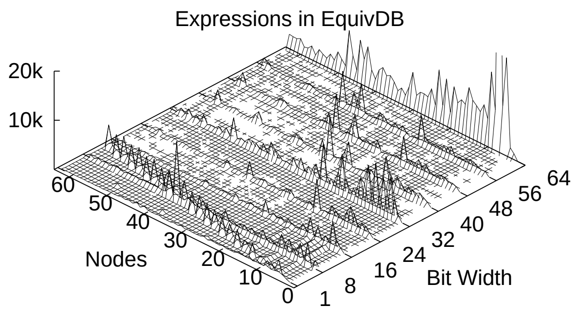

Fig. 9: Distribution of expressions by number of nodes and bitwidth in EquivDB.

The EquivDB distribution is given in Figure 9. Data for the EquivDB was collected from a single run of all programs with a five minute learning period. On the file system, the EquivDB uses approximately 4GB of storage and contains 1.2 million expressions, a modest overhead. Ridges appear at 8 bit multiples, indicating expressions are often byte aligned; possibly because symbolic arrays have byte granularity and most machine instructions are byte-oriented. Some ridges appear to extend past the node limit, suggesting the cut-off could be raised. Blank areas, such as those between 32 and 40 bits indicate no entries

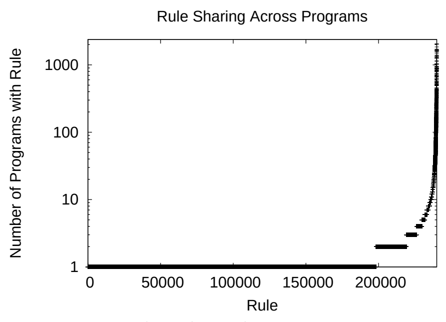

Fig. 10: Rules shared among programs.

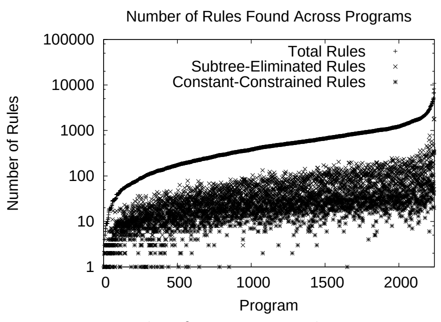

Fig. 11: Rules for measured programs.

for a bit-width and node count. As an outlier, there are 63485 expressions with seven nodes at 64 bits.

7.6 Rules

A large number of rules is indicative of poor rule efficiency. If rules are applied as one-off translations, then it is unlikely a fixed rule set will be effective in general. However, as illustrated in Figure 11, most programs have fewer than a few thousand rules and fewer than a hundred generalized rules. Rule explosion is from only a few programs interacting poorly with the optimizer.

Code sharing is common through shared libraries, so it is reasonable to expect rules to be shared as well. Figure 10 counts the frequency of rules found across programs and confirms sharing. There are 41795 shared rules out of a total of 240052 rules; 17% of rules are shared. The most common rule, with 2035 programs, is an equality lifting rule: $(=\ -1_{32}\ (\texttt{concat}\ -1_{24}\ x_8)) \rightarrow (=\ -1_8\ x_8)$.

8 Related Work

Peephole optimizers using a SAT solver to find optimal short instruction sequences on machine code is a well-known technique [2, 14]. Our expression optimization is similar because it seeks minimal operations. The benefit of applying optimization at the expression level over the instruction level is rules can path-specialize expressions regardless of the underlying code.

For compilers, HOP [8] automatically generated peephole rules by using a formal specification of the target machine. PO simulates runs of register transfers symbolically, using primitive abstract interpretation. It speculatively translates the combined effects back into assembly code. If successful, it replaces the original with the shorter code segment. HOP improved PO by demonstrating skeletal hashes to memoize the rewrite rules for a faster peephole optimizer.

Prior work on symbolic execution uses rule based term rewriting for optimizing solver queries. F-Soft [19] applies a term writing system [7] to expressions generated through symbolic interpretation of C sources. The term rewrite system is seeded with a specification of a few hundred handwritten rules from formal systems (e.g., Presburger arithmetic, equational axiomatization), was applied to a handful of programs, and found improved solver times. However, from the number of rules observed in our system and poor performance of hand-written rules in KLEE, we believe manual rule entry alone is best suited to limited workloads.

References

1. Appleby, A. murmurhash3. http://sites.google.com/site/murmurhash, Nov 2012.
2. Bansal, S., and Aiken, A. Binary translation using peephole superoptimizers. OSDI'08, pp. 177–192.
3. Barrett, C., de Moura, L., and Stump, A. Design and results of the first satisfiability modulo theories competition. *Journal of Automated Reasoning* (2006).
4. Cadar, C., Dunbar, D., and Engler, D. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI'08, pp. 209–224.
5. Chipounov, V., Kuznetsov, V., and Candea, G. S2e: a platform for in-vivo multi-path analysis of software systems. ASPLOS '11, pp. 265–278.
6. Church, A., and Rosser, J. B. Some Properties of Conversion. *Transactions of the American Mathematical Society 39*, 3 (1936), 472–482.
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. The maude 2.0 system. RTA'03, pp. 76–87.
8. Davidson, J. W., and Fraser, C. W. Automatic generation of peephole optimizations. In *SIGPLAN Symposium on Compiler Construction* (1984), pp. 111–116.
9. Ganesh, V., and Dill, D. L. A decision procedure for bit-vectors and arrays. In *CAV'07* (Berlin, Germany, July 2007), Springer-Verlag.
10. Godefroid, P., Levin, M. Y., and Molnar, D. Automated whitebox fuzz testing. In *NDSS '08*.
11. Huet, G. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM 27*, 4 (Oct. 1980), 797–821.
12. Khurshid, S., Pasareanu, C. S., and Visser, W. Generalized symbolic execution for model checking and testing. In *TACAS'03* (2003).
13. Martignoni, L., McCamant, S., Poosankam, P., Song, D., and Maniatis, P. Path-exploration lifting: hi-fi tests for lo-fi emulators. In *ASPLOS* (New York, NY, USA, 2012), ASPLOS '12, ACM, pp. 337–348.
14. Massalin, H. Superoptimizer: a look at the smallest program. ASPLOS-II, pp. 122–126.
15. Molnar, D., Li, X. C., and Wagner, D. A. Dynamic test generation to find integer bugs in x86 binary linux programs. SSYM'09, USENIX, pp. 67–82.
16. Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. PLDI '07, pp. 89–100.
17. Păsăreanu, C., and Visser, W. A survey of new trends in symbolic execution for software testing and analysis. *STTT* (2009), 339–353.
18. Sen, K., Marinov, D., and Agha, G. CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13* (Sept. 2005), pp. 263–272.
19. Sinha, N. Symbolic program analysis using term rewriting and generalization. FMCAD '08, IEEE Press, pp. 19:1–19:9.
20. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. Bitblaze: A new approach to computer security via binary analysis. ICISS '08, pp. 1–25.