

1 Introduction

The objective of this project is to enable the software model-checker, YASM [1], to efficiently detect potential buffer overflows in C programs. YASM currently provides precise verification of memory-safety properties in simple C programs, but is limited in terms of its run-time performance and language support. First, we discuss the architecture of YASM, which is required in order to understand the necessity and meaning of our proposed extensions and optimizations.

The major steps of the YASM analysis loop are:

- **Abstract:** Given a set of predicates E , compute a predicate program \mathcal{P} which is an abstraction of a given C program P .
- **Check:** Check whether \mathcal{P} satisfies its (memory-safety) assertions; the output can be *yes*, *no*, or *maybe*; in the case of *yes* or *no*, terminate, otherwise, continue.
- **Refine:** Add predicates to E and repeat the loop.

This project concentrates on improving the first step, abstraction.

2 Predicate Abstraction

A *predicate program* is, informally, a C program in which the only datatype is boolean. In [2], a predicate program \mathcal{P} is considered to be an *abstraction* of a C program P if all feasible executions of P are feasible in \mathcal{P} . YASM actually computes \mathcal{P} such that the converse of the definition holds as well [1].

Intuitively, the boolean variables in \mathcal{P} represent boolean expressions, or *predicates*, over variables in P ; e.g., a variable b in \mathcal{P} could represent the expression $(x > y)$, where x and y are `int` variables in P . Given a set of predicates E over which \mathcal{P} is defined, in order for \mathcal{P} to be an abstraction of P , \mathcal{P} must have the same control-flow structure as P and, for each statement s in P , there must be a corresponding statement s' in \mathcal{P} which captures the effect of s on each predicate in E .

For example, where $E = \{(x > 0), (y > 0)\}$, with $b_0 \equiv (x > 0)$ and $b_1 \equiv (y > 0)$, the statement $x = y$ could be abstracted as the parallel assignment $b_0 = b_1, b_1 = b_1$, since, intuitively, assigning y to x means $(x > 0)$ is true after the assignment if $(y > 0)$ holds beforehand, and $(x > 0)$ is false otherwise. Since the value of y is unchanged by the statement, $b_1 = b_1$ captures the (nil) effect of the statement on the predicate $(y > 0)$.

For a given set of predicates E , an abstraction \mathcal{P} is obtained by first computing the *weakest precondition* for each predicate in E with respect to each statement in P . The weakest precondition of a predicate ϕ with respect to a statement $x = e$, written $WP(x = e, \phi)$, is ϕ with all occurrences of x replaced with e : $\phi[e/x]$ ¹. Intuitively, $WP(x = e, \phi)$ yields the weakest condition which must hold before executing $x = e$ in order for ϕ to hold afterwards. Note that this WP computation is strictly syntactic. In the example above, $WP(x = y, (x > 0)) = (x > 0)[y/x] = (y > 0)$.

In general, $WP(x = e, \phi)$ may yield a formula which is not in E ; for example, where $E = \{(x > 0)\}$, $WP(x = y, (x > 0)) = (y > 0) \notin E$. Thus, the abstraction step relies on a theorem prover

¹YASM actually computes a slightly more complicated WP in order to account for the possibility of aliasing.

to compute what is called a *predicate strengthening* in [2], which, for a target formula ϕ , is a formula $\mathcal{F}_E(\phi)$ defined strictly over predicates in E such that $\mathcal{F}_E(\phi) \Rightarrow \phi$. For example, where $E = \{(x > 0), (y > 1)\}$, $WP(x = y, (x > 0)) = (y > 0) \notin E$, but $(y > 1) \Rightarrow (y > 0)$ and $(y > 1) \in E$; the theorem prover is used to establish the truth of the implication, i.e., that $\mathcal{F}_E((y > 0)) = (y > 1)$.

A naive predicate strengthening procedure generates $2^{|E|}$ cubes, each of which is a conjunction over all the (possibly negated) predicates in E . For each cube, the naive procedure calls the theorem prover to check whether the cube implies the target formula. The result is the disjunction of all satisfying cubes. In the example above, the cubes and their evaluations are as follows:

Cube	Eval.	Target
$(x > 0) \wedge (y > 1)$	\Rightarrow	$(y > 0)$
$(x > 0) \wedge \neg(y > 1)$	$\not\Rightarrow$	$(y > 0)$
$\neg(x > 0) \wedge (y > 1)$	\Rightarrow	$(y > 0)$
$\neg(x > 0) \wedge \neg(y > 1)$	$\not\Rightarrow$	$(y > 0)$.

The result is thus $(x > 0) \wedge (y > 1) \vee \neg(x > 0) \wedge (y > 1)$. Note that an additional non-trivial procedure is required to simplify the result to the logically equivalent expression, $(y > 1)$. YASM currently uses a standard constraint-satisfaction algorithm called conflict-directed back-jumping to attempt to prune the cube search space and to produce a simpler strengthening. For this example, YASM’s strengthening procedure produces $(y > 1)$ instead of the larger, redundant formula.

3 Challenge

Running YASM on small buffer-manipulating programs to check for buffer overflows shows that the theorem-prover dominates analysis time; it accounts for nearly all the time spent in the abstraction step, and around two-thirds of the overall analysis time. Moreover, the time spent in the theorem-prover seems to be proportional to the size of the buffer being checked. Thus, in order to improve YASM’s performance, we must reduce the amount of time spent in the theorem-prover by reducing the number of theorem-prover calls and the cost of each call. A secondary concern is to extend YASM’s language support and its interpretation of standard C library functions.

4 Completed Optimizations

Related predicates. In generating candidate cubes from a set of predicates E to compute $\mathcal{F}_E(\phi)$, it seems to be beneficial to first reduce E to only those predicates which mention a variable in ϕ or an alias of such a variable; these predicates are deemed to be *related* to ϕ . For example, if $\phi = (y > 0)$ and $E = \{(y = \&z), (z > 0), (y > 1), (w = 2), (x > 0)\}$, then the only predicates in E which are related to ϕ are $\{(y = \&z), (z > 0), (y > 1)\}$. Computing these related predicates potentially reduces the cube search-space and, as the computation is strictly syntactic, introduces negligible additional cost. In experiments on simple buffer-manipulating programs, this optimization reduced the number of queries and theorem-proving time by nearly a factor of three.

Reducing communication overhead. YASM communicates with an external theorem-prover process using pipes. As such, some of the theorem-proving time is spent in sending and receiving data. By shortening the names of identifiers and in-lining certain functions comprising the queries, the amount of data sent between YASM and the theorem-prover was reduced, yielding a modest improvement in theorem-proving time.

5 Proposed Optimizations

Caching strengthening computations. Currently, YASM’s strengthening procedure is stateless. Thus, given two sets of predicates E and E' , where $E \subseteq E'$, in computing $\mathcal{F}_{E'}(\phi)$ the procedure repeats much of the work performed in computing $\mathcal{F}_E(\phi)$ since it generates and checks many of the same cubes for E' as it does for E . By caching the inputs (E and ϕ) and results ($\mathcal{F}_E(\phi)$), we may eliminate many of these redundant computations as E grows. For example, if in two separate computations with predicate sets $E \subseteq E'$ none of the predicates in $E' \setminus E$ are related to the target ϕ , we can altogether avoid (re-)computing $\mathcal{F}_{E'}(\phi)$ since it is equivalent to $\mathcal{F}_E(\phi)$ in this case.

Using externally computed invariants. An invariant is a predicate which always holds at a program location. For example, alias analysis could provide the invariant $p \neq \&x$ for a statement $x = 3$, which means that p does not point to x at that statement. These invariants could be used to simplify WP formulas (by eliminating contradictory subformulas), which would in turn reduce the cost of the computing the strengthening. In general, it may be beneficial to use alternative preconditions, that is, preconditions which are not necessarily the weakest but which yield cheaper strengthening computations.

Limiting cube size. Whereas SLAM limits the size of cubes to three predicates, yielding a search space of size $(2|E|)^3$, YASM currently generates cubes as large as $|E|$, yielding a search space of size $2^{|E|}$. The results in [2] suggest that limiting the cube size to three is sufficient for most analyses to terminate conclusively, so it may be worthwhile to use the same heuristic in YASM.

Decomposing WP into atomic predicates. In general, WP may be non-atomic, containing conjunctions and disjunctions. Thus, instead of computing $\mathcal{F}_E(\phi)$ for a composite formula ϕ , it may reduce theorem-proving time to decompose WP into atomic predicates ϕ_i , compute each $\mathcal{F}_E(\phi_i)$ individually, and compose the results, assuming the theorem-prover can decide queries (implications) over an atomic formula significantly faster than a composite formula.

6 Proposed Extensions

Structs. YASM currently does not support structs. It should be straightforward to modify YASM’s front-end to faithfully translate structs to its internal representation language.

Heap and string functions. YASM currently does not provide any special interpretation of heap and string functions such as `malloc`, `strlen`, and `strcpy`. Thus, if the body of such a function is not provided, the function is *uninterpreted*, that is, assumed to have an indeterminate effect. In analyzing programs for buffer overflows, it may be sufficient to only capture some of the semantics of these functions, such as the effect of `strcpy` on the length of a string, or the effect of `strlen` on the value of a predicate ($*s == 0$). The particular semantics of these functions could be captured by special cases of the WP procedure.

References

- [1] Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A Software Model-Checker for Verification and Refutation. In Proceedings of CAV’06, LNCS 4144, pp. 170-174, August 2000.
- [2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In PLDI 2001.