Automated Differential Program Verification for Approximate Computing

Shuvendu K. Lahiri
Microsoft Research, Redmond, WA, USA
shuvendu@microsoft.com

Arvind Haran, Shaobo He, Zvonimir Rakamarić
School of Computing, University of Utah, UT, USA
{haran,shaobo,zvonimir}@cs.utah.edu

***Abstract*—Approximate computing is an emerging area for trading off the accuracy of an application for improved performance, lower energy costs, and tolerance to unreliable hardware. However, care has to be taken to ensure that the approximations do not cause significant divergence from the reference implementation. Previous research has proposed various metrics to guarantee several relaxed notions of safety for the design and verification of such approximate applications. However, current approximation verification approaches often lack in either precision or automation. On one end of the spectrum, type-based approaches lack precision, while on the other, proofs in interactive theorem provers require significant manual effort.**

In this work, we apply automated differential program verification (as implemented in SymDiff) for reasoning about approximations. We show that mutual summaries naturally express many relaxed specifications for approximations, and SMT-based checking and invariant inference can substantially automate the verification of such specifications. We demonstrate that the framework significantly improves automation compared to previous work on using Coq, and improves precision when compared to path-insensitive analysis. Our results indicate the feasibility of applying automated verification to the domain of approximate computing in a cost-effective manner.

I. Introduction

Continuous improvements in per-transistor speed and energy efficiency are fading, while we face increasingly important concerns of power and energy consumption, along with ambitious performance goals. The emerging area of *approximate computing* aims at lowering the computational effort (e.g., energy) of an application through controlled (small) deviations from the intended results [1]. Low-level approximation mechanisms include, for example, approximating digital logic elements or arithmetic; high-level mechanisms include approximating loop computations or generating multiple approximate candidate implementations. In addition, many of these studies also show that large classes of applications can in fact tolerate small approximations (e.g., machine learning, web search, multimedia, sensor data processing).

There is a growing need to develop *formal* and *automated* techniques that allow approximate computing trade-offs to be explored by developers. Prior research has ranged from the use of types [2], static reliability analysis [3] or interactive theorem provers [4] to study the effects of approximations while also providing formal guarantees. While these techniques have significantly increased the potential to employ approximate computing in practice, a drawback is that they either lack the required level of precision or degree of automation. More importantly, these works do not harness the continuous advances in Satisfiability Modulo Theories (SMT) [5] based automatic software verification [6]. SMT-based approaches have the potential of providing a good balance of precision and scalability, without sacrificing automation, at least for a large class of programs written in imperative languages such as C/C++, Java, or C#.

In this paper, we apply *automated differential program verification* [7], [8] (implemented in SymDiff [9]) towards the problem of *logical*[1] reasoning about program approximations. In previous work, we have shown that structural similarity of closely-related programs can be exploited to perform automated verification of relative safety for assertions [7]. In this work, we leverage two independent ideas in SymDiff to harness the power of SMT solvers towards differential verification. First, we use the concept of *mutual summaries* for specifying relational (two-program) properties related to approximation [8]. Second, we use a novel product program construction for *differential assertion checking* that permits procedural programs, and allows leveraging off-the-shelf program verifiers and invariant inference engines [7]. We describe how the same product construction can be used to check mutual summary specifications as well. Finally, we have incorporated additional relational specification inference capabilities in SymDiff, which is crucial for proving the examples in this paper (§ VI-A).

We have applied SymDiff towards two approximate computing case studies. First, we apply SymDiff towards formalizing *control flow equivalence*, a precise specification for ensuring that an approximation does not impact control flow of a program. Approximations that impact control flow often lead to undesirable behavior, such as non-termination or crashes. Unlike type-based approaches [2], we illustrate that SymDiff can provide verification of this property with desired precision and little overhead. We show examples where the need to track dynamic segments of arrays are crucial to enable precise reasoning about impact on control flow. [2]

Second, we illustrate the modeling, specification, and proof of several *acceptability conditions* for approximate transformations studied by Carbin et al. [4]. They developed a domain-specific language for specifying approximations and

[1]We distinguish from approaches that provide provide probabilistic guarantees regarding approximations [3].

[2] Tool and examples present at http://tinyurl.com/approx-symdiff-2015

```
var arr:[int]int;
var n:int; var x:int; var y:int;
procedure ReplaceChar() {
  call Helper(0);
}
procedure Helper(i:int) {
  var tmp:int;
  if (i < n && arr[i] != -1) {
    tmp := arr[i];
    havoc tmp;
    arr[i] := tmp == x ? y : tmp;
    call Helper(i+1);
  }
}
```

Fig. 1: Replacing a character in a string.

```
function RelaxedEq(x:int, y:int) returns (bool) {
  (x <= 10 && x == y) || (x > 10 && y >= 10)
}

procedure swish(max_r:int, N:int)
    returns (num_r:int) {
  old_max_r := max_r; havoc max_r;
  assume RelaxedEq(old_max_r, max_r);
  num_r := 0;
  while (num_r < max_r && num_r < N)
    num_r := num_r + 1;
  return;
}
```

Fig. 2: Swish++ example with dynamic knobs approximation.

acceptability conditions, and performed the verification of several examples using interactive theorem prover Coq. These examples cover approximations due to truncating loops, unreliable memory, and relative memory safety [7]. Overall, their proofs for three examples required around 955 lines of Coq proof script — this makes it difficult to scale the effort to larger programs or hundreds of such programs. In contrast, our verification in SymDiff requires the user specify 8 simple atomic predicates.

Contributions. In summary, we make the following contributions:

1) We formalize the notion of *control flow equivalence* as a differential verification problem.
2) We apply automated differential verification in SymDiff for checking control flow equivalence and correctness of approximate transformations.
3) We have enhanced the inference of relational specifications in SymDiff, which improved the automation on the examples in this paper.

Organization. The rest of the paper is organized as follows. We give an overview of motivating problems in § II. Next, we briefly summarize relevant background on SymDiff; this includes combining the specification mechanism [8] and product program construction [7], originally developed separately (§ III). We then describe the examples from our two case studies: checking impact of approximation on control flow (§ IV), and checking various approximation acceptability conditions (§ V). We describe the modifications made to SymDiff (§ VI-A) and the experimental results (§ VI-B). Finally, we discuss the related work (§ VII) and conclusions (§ VIII).

II. Motivating Examples

A. Control Flow Equivalence

Approximating statements that impact control flow often leads to serious problems in guaranteeing program termination, unacceptably high corruptions in output data, and program crashes. Preservation of control flow has been identified as a natural and useful relaxed specification for approximations [2]. One can obtain a conservative estimate of the set of statements that do not affect control flow by performing type-based analysis [2], dataflow analysis [10], or slicing [11]. Although mostly automatic (type-based analyses typically require user-provided type information), these approaches are conservative and cannot exploit detailed program semantics.

Consider the program in Fig. 1 that replaces a given character x with y in a character array arr. The procedure Helper iterates over indices of the array until the bound n or the termination character (-1 in this case) is reached. Let us consider the approximation of the variable tmp indicated by the underlined statement. Since tmp flows into arr which controls the conditional, most mentioned conservative analysis would mark the approximation as unsafe. However, observe that the indices that store the value in tmp never participate in the conditional. Therefore, any analysis that cannot track dynamic segments of an array will result in a false alarm. Similarly, lack of path-sensitivity can also result in such imprecisions. Our approach leverages differential program verification to check for control flow equivalence, which allows for precise analysis, as described in detail in § IV.

B. Acceptability of Approximate Programs

Fig. 2 gives the example from an open-source search engine *Swish++*, which was taken from a recent approximate computing work by Carbin et al. [4]. The authors developed a special-purpose language to specify the transformations and reason about the relaxed specifications. They used the general purpose Coq theorem prover to discharge proof obligations; each proof required roughly 330 lines of proof scripts according to the authors. By using differential program verification that leverages existing program verifiers and SMT solvers, we show that we can obtain the proof almost completely automatically (see § V).

The program swish takes as input (a) a threshold for the maximum number of results to display max_r, and (b) the total number of search results N. It returns the number num_r denoting the actual number of results to display, which has to be bounded by max_r and N.

a) Approximation: The underlined statements denote the approximation that non-deterministically changes the threshold to a possibly smaller number, without suppressing the top few (10 in this case) results. This approximation (referred to as *dynamic knobs*) allows the search engine to trade-off the number of search results to display under heavy server load. The approximation is justified as users are typically interested in the top few results, and care more about the performance of displaying the search results. The predicate `RelaxedEq` denotes the relationship between the original and the approximate value — the important part is that approximate value has to be at least 10 when the original value exceeds 10.

b) Relaxed Specification: The relaxed specification (akin to *acceptability property* [4], [12]) can be expressed as a *mutual summary* [8] over the original and approximate versions of `swish` (prefixed with `v1.` and `v2.` respectively) as follows:

$$\mathbf{old}(\texttt{v1.max\_r} = \texttt{v2.max\_r} \wedge \texttt{v1.N} = \texttt{v2.N}) \Rightarrow \texttt{v1.RelaxedEq}(\texttt{v1.num\_r}, \texttt{v2.num\_r}),$$

where the construct $\mathbf{old}(\mathit{Expr})$ evaluates the given expression at procedure entry. In this work, we automated the process of proving this and similar examples using SymDiff, which is described in detail in § V.

III. Differential Program Verification

In this section, we cover recent works on differential program verification [8], [7] to verify (relational) properties over two programs, as implemented in the SymDiff tool [9]. We first describe *mutual summaries* [8] as a specification mechanism for relational properties (§ III-B). Then, we introduce a method for modularly checking mutual summary specifications based on a product program transformation [7] (§ III-C). Although the transformation was proposed for *differential assertion checking*, we show that the construction can be used to check more general mutual summary specifications. These mechanisms are well-suited for reasoning about programs with multiple (recursive) procedures. More importantly, the technique allows for leveraging any off-the-shelf invariant inference engine to infer intermediate specifications required to prove the desired specification (§ III-D). We start by formalizing the language in the next section.

A. Simple Programming Language

Fig. 3 defines the syntax of a simple programming language, which is a subset of the Boogie language [13]. The language supports integers **int**, arrays [**int**]**int**, and booleans **bool**. A program consists of a set of global variables and a set of one or more procedures. A procedure has zero or more input parameters and output variables. The **requires** and **ensures** clauses specify preconditions and postconditions/summaries, respectively; the **modifies** clause specifies the globals that may be modified in a procedure. A procedure body contains local variable declarations and a sequence of statements *Stmt*. Loops are assumed to be already automatically extracted into deterministic tail-recursive procedures [7].

Most statements, including assignments (scalar and array), conditionals, and sequential composition, are standard. Statement **havoc** x sets variable x to an arbitrary value, while the **call** statement denotes a procedure invocation. Informally, the **assert** *Expr* (resp., **assume** *Expr*) *fails* (resp., *blocks*) execution when *Expr* evaluates to **false** in a state; otherwise,

$$\begin{array}{rcl}
\mathit{Type} & ::= & \mathbf{int} \mid [\mathbf{int}]\mathbf{int} \mid \mathbf{bool} \\
\mathit{Program} & ::= & (\mathbf{var}\ \mathit{Id} : \mathit{Type};)^{*}\ \mathit{Procedure}^{+} \\
\mathit{Procedure} & ::= & \mathbf{procedure}\ \mathit{Id}((\mathit{Id} : \mathit{Type},)^{*})\ \mathit{Returns}^{?} \\
 & & \quad \mathit{Spec}^{*}\ \{\mathit{Body}\} \\
\mathit{Spec} & ::= & \mathbf{requires}\ \mathit{Expr}; \mid \mathbf{ensures}\ \mathit{Expr}; \\
 & \mid & \mathbf{modifies}\ (\mathit{Id},)^{*}; \\
\mathit{Returns} & ::= & \mathbf{returns}\ ((\mathit{Id} : \mathit{Type},)^{*}) \\
\mathit{Body} & ::= & (\mathbf{var}\ \mathit{Id} : \mathit{Type};)^{*}\ \mathit{Stmt} \\
\mathit{Stmt} & ::= & \mathit{Id} := \mathit{Expr} \mid \mathit{Id}[\mathit{Expr}] := \mathit{Expr} \\
 & \mid & \mathbf{if}\ (\mathit{Expr})\ \mathit{Stmt}\ \mathbf{else}\ \mathit{Stmt} \\
 & \mid & \mathit{Stmt};\ \mathit{Stmt} \mid \mathbf{havoc}\ \mathit{Id} \\
 & \mid & \mathbf{call}\ (\mathit{Id},)^{*} := \mathit{Id}((\mathit{Id},)^{*}) \mid \mathbf{return} \\
 & \mid & \mathbf{assume}\ \mathit{Expr} \mid \mathbf{assert}\ \mathit{Expr}
\end{array}$$

Fig. 3: Simple programming language. *Id* and *Expr* have the usual meaning.

it acts as a skip. The expression language of *Expr* is left unspecified, but includes standard integer-valued arithmetic expressions (e.g., $x+y$) and Boolean-valued expressions (e.g., $x \leq y$). In addition, the construct $\mathbf{old}(\mathit{Expr})$ can be used to evaluate an expression at entry to a procedure.

We informally sketch the semantics for the language here; more formal details can be found in earlier works [13]. A state σ of a program at a program location is a type-consistent valuation of variables in scope at the location, or the error state $Error$. Let Σ be the set of all states for a program. For any procedure p, we assume a transition relation $\mathcal{T}_p \subseteq \Sigma \times \Sigma$ that characterizes the input-output relation of the procedure p. In other words, two states $(\sigma, \sigma') \in \mathcal{T}_p$ if there is an execution of the procedure p starting at σ and ending in σ'. The transition relations can be defined inductively on the structure of the program, which is fairly standard for our simple language. For any state σ and an expression e, $\langle e \rangle_\sigma$ evaluates e in the state σ.

B. Specification: Mutual Summaries

```
var g:int; // global                var g:int; // global
procedure F(x:int)                  procedure F(x:int)
modifies g;                         modifies g;
{                                   {
  if (x < 100) {                      if (x < 100) {
    g := g + x;                         g := g + 2*x;
    call F(x+1);                        call F(x+1);
  }                                   }
}                                   }
procedure Main()                    procedure Main()
modifies g;                         modifies g;
{ call F(0); }                      { call F(0); }
```

Fig. 4: Two versions of a program with a change in `F`.

Consider a program P and procedure p belonging to P. A *summary* specification S_p is a Boolean-valued expression

```
procedure MS_v1.F_v2.F(v1.x:int, v2.x:int)
modifies v1.g, v2.g;
requires v1.x >= 0; // intermediate spec (manual)
{
...
}

procedure MS_v1.Main_v2.Main()
modifies v1.g, v2.g;
requires v1.g == v2.g; // equal initial state
ensures  v1.g <= v2.g; // mutual postcond (manual)
{
...
}
```

Fig. 5: Signature of the composed program for example in Fig. 4. Details of the construction are in Appendix A.

over the input (parameters and globals) and output (returns and globals) variables of p that specifies a constraint on the transition relation $\mathcal{T}_p$ of the procedure. More formally, a well-formed summary expression S_p induces a relation $\lfloor S_p \rfloor = \{(\sigma, \sigma') \mid \langle S_p \rangle_{\sigma,\sigma'} = \mathbf{true}\}$. A procedure p *satisfies a summary* S_p if $\mathcal{T}_p \subseteq \lfloor S_p \rfloor$ — all terminating executions of p satisfy S_p.

Now consider two procedures p and q. An expression $M_{p,q}$ is a well-formed *mutual summary* specification if it is an expression over the input and output variables of p and q. Such a specification represents the relation $\lfloor M_{p,q} \rfloor = \{(\sigma_p, \sigma'_p, \sigma_q, \sigma'_q) \mid (\sigma_p, \sigma'_p) \in \mathcal{T}_p,\ (\sigma_q, \sigma'_q) \in \mathcal{T}_q,\ \langle M_{p,q} \rangle_{\sigma_p,\sigma'_p,\sigma_q,\sigma'_q} = \mathbf{true}\}$. A procedure pair (p, q) *satisfies a mutual summary* $M_{p,q}$ if $\mathcal{T}_p \times \mathcal{T}_q \subseteq \lfloor M_{p,q} \rfloor$.

Consider the two program versions in Fig. 4, with a change in procedure F, and the following mutual summary for Main:

$$\mathbf{old}(\texttt{v1.g} = \texttt{v2.g}) \Rightarrow \texttt{v1.g} \leq \texttt{v2.g}.$$

The summary relates the pre- and post-states of the two versions (prefixed with v1. and v2. respectively) of the program. It is not difficult to see that the procedure pair $(\texttt{v1.Main}, \texttt{v2.Main})$ satisfies this mutual summary specification, since the argument x to F is always non-negative in executions starting from Main. In the next section, we describe how to specify and modularly verify such mutual summary specifications for a pair of programs.

C. Modular Checking of Mutual Summaries

We describe the modular checking of mutual summaries (using the construction from previous work [7]) with the aid of the running example in Fig. 4. We first sketch the product program construction for the running example, and later describe how to add specifications to the product program.

1) Product Programs: For a program P, let us overload P to also represent the set of procedures in P. Consider two programs P_1, P_2, and a mapping relation $\beta \subseteq P_1 \times P_2$ that maps procedures from two versions. A default value of β is a one-to-one mapping between identically named procedures from the two programs, but this can be changed by the user. For our running example $P_1 = \{\texttt{v1.F}, \texttt{v1.Main}\}$ and $P_2 = \{\texttt{v2.F}, \texttt{v2.Main}\}$, and we consider the default mapping $\beta = \{(\texttt{v1.Main}, \texttt{v2.Main}), (\texttt{v1.F}, \texttt{v2.F})\}$.

Given such P_1, P_2 and β, we construct a product program $P_{1\times2}$ with the following properties:

- The set of globals in $P_{1\times2}$ is the disjoint union of globals in P_1 and P_2. The globals are prefixed with v1. and v2. respectively to avoid name clashes.
- The set of procedures in $P_{1\times2}$ consists of the disjoint union of procedures from P_1 and P_2 (signature suitably prefixed) along with a set of *product* procedures. For each $(p, q) \in \beta$, there is a procedure MS_p_q whose input and output parameters are concatenations of the parameter lists from p and q.

For the running example, $P_{1\times2}$ consists of globals $\{\texttt{v1.g}, \texttt{v2.g}\}$ and procedures $\{\texttt{v1.F}, \texttt{v1.Main}, \texttt{v2.F}, \texttt{v2.Main}, \texttt{MS\_v1.F\_v2.F}, \texttt{MS\_v1.Main\_v2.Main}\}$. Fig. 5 shows the signature (parameters and specifications) of the MS_v1.F_v2.F procedure. Details of the construction of the MS_v1.F_v2.F procedure are described in Appendix A.

Let $\sigma_1 \oplus \sigma_2$ denote a composed state consisting of states from the two programs with disjoint signatures. The following theorem relates MS_p_q with the procedures p and q.

Theorem 1 ([7]): For two procedures $p \in P_1$ and $p_2 \in P_2$, $(\sigma_1, \sigma'_1) \in \mathcal{T}_p$ and $(\sigma_2, \sigma'_2) \in \mathcal{T}_p$ if and only if $(\sigma_1 \oplus \sigma_2, \sigma'_1 \oplus \sigma'_2) \in \mathcal{T}_{\texttt{MS\_}p\texttt{\_}q}$.

2) Adding Specifications: Theorem 1 allows us to write summary specifications on the product program to capture mutual summary specifications over P_1 and P_2. Since the signature (inputs and outputs) of a product procedure MS_p_q is the disjoint union of the signatures of p and q, a well-formed summary expression $S_{\texttt{MS\_}p\texttt{\_}q}$ is a well-formed mutual summary expression for the procedure pair (p, q). Hence, verifying summary specifications on the product program allows us to verify mutual summary specifications over the two programs.

Theorem 2: Consider a product procedure $\texttt{MS\_}p\texttt{\_}q \in P_{1\times2}$ and a summary specification $S_{\texttt{MS\_}p\texttt{\_}q}$. Let $M_{p,q}$ be a mutual summary specification such that $\lfloor M_{p,q} \rfloor = \lfloor S_{\texttt{MS\_}p\texttt{\_}q} \rfloor$. If $\mathcal{T}_{\texttt{MS\_}p\texttt{\_}q} \subseteq \lfloor S_{\texttt{MS\_}p\texttt{\_}q} \rfloor$, then $\mathcal{T}_p \times \mathcal{T}_q \subseteq \lfloor M_{p,q} \rfloor$.

Recall the mutual summary specification for the pair of Main procedures. This can be expressed as a summary for MS_v1.Main_v2.Main procedure as a postcondition (**ensures** clause):

$$\mathbf{ensures}(\mathbf{old}(\texttt{v1.g} = \texttt{v2.g}) \Rightarrow \texttt{v1.g} \leq \texttt{v2.g}).$$

Fig. 5 shows the above specification; however, we have broken up the specification into a **requires** (a precondition constraining the state at entry) and **ensures** clause to simplify the specification. SymDiff automatically inserts the equalities in the **requires** for entry procedures, and the user only has to specify the **ensures** clause in this case.

The program $P_{1\times2}$ along with the desired specification can be handed off to any off-the-shelf (single) program verifier such as Boogie to attempt the verification. The verifier can leverage advances in SMT solvers to perform reliable and predictable verification. If verification succeeds, then we can establish the mutual summary for the pair of procedures.

D. Invariant Inference

By default, SymDiff performs automatic inference of simple relative specifications by searching for preconditions and postconditions of the form $\texttt{v1.x} \bowtie \texttt{v2.x}$, where $\bowtie \in \{=, \leq, \geq, <, >, \Rightarrow\}$. It leverages the implementation of the Houdini [14] (monomial predicate abstraction) inference technique available in Boogie [15]. This allows SymDiff to communicate relational (mutual) specifications to the invariant inference engine. For example, this allows us to automatically infer several intermediate specifications, including inferring that $\texttt{v1.g} \leq \texttt{v2.g}$ is both a precondition and postcondition for MS_v1.F_v2.F. Any remaining intermediate specification (e.g. the **requires** for MS_v1.F_v2.F) have to be manually specified by the user.

In Section VI-A, we describe improvements we implemented in the SymDiff invariant inference engine to automate the examples described in this paper.

IV. Control Flow Equivalence

A. Modeling Control Flow

We describe a precise and automated (although not push-button) approach to ensure that an approximation does not impact control flow by leveraging differential program verification. We achieve this by performing an automatic program instrumentation (described below) and then leveraging the differential verifier as described earlier in § III.

Let us define a *basic block* to be the maximal sequence of statements that do not contain any conditional statements. We also assume that each such basic block has a unique identifier associated with it. To track the sequence of basic blocks visited along any execution, we augment the state of a program by introducing an integer-valued global variable cflow. Then, we instrument every basic block of the program with a statement of the form $\texttt{cflow} := \texttt{trackCF}(\texttt{cflow}, \texttt{blockID})$, where trackCF is an uninterpreted function defined as $\texttt{trackCF}(\texttt{int}, \texttt{int})$ returns int, and blockID is the unique integer identifier of the current basic block.

Let v1.p and v2.p be the two versions of a procedure p in the original and the approximate program. Consider the following mutual summary for the product procedure MS_v1.p_v2.p (assuming the inputs including cflow start out equal):

$$\textbf{ensures}\ \texttt{v1.cflow} == \texttt{v2.cflow}.$$

If the product program satisfies this mutual specification, then the injected approximations do not change the control flow of the program (*control flow equivalence*). More formally, if MS_v1.p_v2.p satisfies this specification, then the following holds:

> For any pair of executions $(\sigma_1, \sigma_2) \in \mathcal{T}_{v1.p}$ and $(\sigma_1, \sigma_3) \in \mathcal{T}_{v2.p}$ starting at the same input state σ_1, the sequences of basic blocks in the two executions are identical.

Hence, we translate the problem of determining if a set of approximations impacts control flow to the problem of verifying a mutual summary on the product program. Note that the formalism currently does not detect non-termination introduced in the approximation; we plan to leverage the *relative termination* specifications in future work [8].

```
var array:[int]int;
var n:int;

procedure SelectionSort() {
  var c:int, position:int, temp:int;
  position := 0;
  temp := 0;
  c := 0;

  while (c < (n - 1)) {
    call position := Find(c);
    if (position != c) {
      temp := array[position];
      array[position] := array[c];
      havoc temp;
      array[c] := temp;
    }
    c := c + 1;
  }
}

procedure Find(c:int) returns (position:int) {
  var d:int;
  position := c;
  d := c + 1;
  while (d < n) {
    if (array[position] > array[d]) {
      position := d;
    }
    d := d + 1;
  }
}
```

Fig. 6: Selection sort example.

Recall the *ReplaceChar* example from Fig. 1 where we wish to verify that the approximation preserves control flow equivalence. The main challenge for verification is to capture the fact that control flow depends on only a fragment of the array, which are identical in the two programs. We capture this by defining a quantified atomic predicate $\mathit{ArrayEqAfter}(\texttt{v1.arr}, \texttt{v2.arr}, \texttt{v1.i}) \doteq \forall j : \textbf{int} :: j \geq \texttt{v1.i} \Rightarrow \texttt{v1.arr}[j] == \texttt{v2.arr}[j]$. This predicate is manually specified to SymDiff to construct specifications by combining this predicate with other automatically generated predicates.

B. Selection Sort Example

The same reasoning principle (tracking array fragments to verify control flow equivalence) also arises in other array based programs. Fig. 6 gives the source code of selection sort. The algorithm sorts an array of length n by pushing the maximum element of the $[0 \ldots c-1]$ subarray to the position c after every iteration. Once an element has been pushed to the end, it does not play a part in determining future control flow behavior. Therefore, the underlined fault does not influence the control flow of the algorithm. As before, taint analysis flags the whole array object as tainted since tracking array indices precisely is typically infeasible using static analysis. The predicate $\mathit{ArrayEqAfter}(\texttt{v1.arr}, \texttt{v2.arr}, \texttt{v1.i})$ is also

```
function A(i:int, j:int) returns (int);
const e:int; axiom e >= 0;

function RelaxedEq(x:int, y:int) returns (bool) {
  x <= y + e && y <= x + e
}

procedure lu(j:int, N:int, max0:int)
    returns (max:int, p:int) {
  i := j+1; max := max0;
  while (i < N) {
    a := A(i, j);
    old_a := a; havoc a; assume RelaxedEq(old_a,a);
    if (a > max) { max := a; p := i; }
    i := i + 1;
  }
  return;
}
```

Fig. 7: LU decomposition example.

essential for proving control flow equivalence for this sorting example (§ VI-B).

In addition to selection sort, we also verified control flow equivalence for a version of bubble sort containing a similar approximation. Unlike selection sort, the approximation requires introducing an additional instruction to havoc the rightmost index modified by the inner loop.

V. Acceptability of Approximate Programs

We motivated proving relaxed specification for approximate transformation in § II-B. This was one of the examples studied by the work of Carbin et al. [4]. In this section, we illustrate the application of differential verification for the remaining two examples from this work. Later we describe our experience automating verification of these examples with SymDiff.

A. Approximate Memory and Data Type

Fig. 7 gives a portion of the *LU Decomposition* algorithm implemented in SciMark2 benchmark suite [16]. The algorithm computes the index of the pivot row p for a column j, where the pivot row contains the maximum value among all rows in the column. It returns the index p of the pivot in addition to the value of the maximum element in column j.

a) Approximation: The underlined statements model the introduction of an error value e if the matrix is stored in approximate memory [17]. As before, the predicate RelaxedEq denotes the relationship between the original and approximate value read from the memory; in this case, they are bounded by a non-negative constant e.

b) Relaxed Specification: Similar to Swish++, the relaxed specification for the pair of lu procedures is specified by the postcondition on MS_v1.lu_v2.lu:

$$\textbf{ensures}\ \texttt{RelaxedEq}(\texttt{v1.max}, \texttt{v2.max}).$$

The Coq proof comprised of 315 lines of proof script [4].

```
1   var FF, RS:[int]int;
2   var K:int;
3   function exp(int) returns (int);
4
5   procedure water(len_FF:int,
6   len_RS:int, N:int, gCUT2:int) {
7     K := 1;
8     havoc RS; // approximation
9
10    while (K < N) {
11      assert (K < len_FF);
12      assert (K < len_RS);
13      if (RS[K] < gCUT2) {
14        // assert (K < len_FF);
15        FF[K] := exp(RS[K]);
16      }
17      K := K + 1;
18    }
19  }
```

Fig. 8: Water example.

B. Statistical Automatic Parallelization

Fig. 8 gives an example from a parallelization of the Water computation [18]. In the loop, the result of RS[K] is compared with a cutoff gCUT2, and then another array FF is updated at index K. The bounds of the two arrays are provided in the len_RS and len_FF variables.

c) Approximation: To maximize performance, the parallelization eliminates locks, which can result in race conditions for the array RS. This is modeled by **havoc**-ing the entire array RS.

d) Relaxed Specification: The assertions model memory safety, and ensure that the program accesses the two arrays within bounds. The relaxed specification has to ensure *relative memory safety* — that the assertions in the approximate version do not fail more often than the original version. The Coq proof comprised of 310 lines of proof script [4].

We exploit the formalization of *differential assertion checking* [7], which automatically replaces an assertion **assert** ϕ with an update to a global variable $\texttt{OK} := \texttt{OK} \land \phi$, and inserts a mutual postcondition $\texttt{v1.OK} \Rightarrow \texttt{v2.OK}$ when starting from equal states. Hence, we did not have to make any changes to define the relative specification.

Although it is desirable to check the assertion in line 14 relatively, the approximate version is not relatively correct with this assertion (also mentioned by Carbin et al. [4]). We instead prove the weaker assertion in line 11 that essentially expresses that len_FF is not correlated with the value in array RS.

VI. Evaluation

A. Implementation

Recall that SymDiff automatically generates relational (two-program) predicates of the form $\texttt{v1.x} \bowtie \texttt{v2.x}$, where $\bowtie \in \{=, \leq, \geq, <, >, \Rightarrow\}$, and searches for conjunctive (mutual) pre- and post-conditions over these predicates. We augmented SymDiff in two main directions to improve the inference of intermediate specifications beyond this existing scheme.

TABLE I: Experimental results. The last four rows check control flow equivalence. #Preds is the number of atomic predicates automatically generated by SymDiff; #Manual is the number of manually provided predicates; #Min-disj is the minimum number of disjunctions required; Time is the runtime in seconds with the minimum disjunction bound. Experiments were performed on a 2.3 GHz 64-bit Quad Core Intel i7-3610QM processor with 8GB DDR3 RAM, running Microsoft Windows 8.1 Professional Edition.

Benchmark	#Preds	#Manual	#Min-disj	Time(s)
Swish++	14	4	1	5.7
LU Decomposition	32	4	0	6.7
Water	27	0	0	6.7
ReplaceChar	10	1	0	7.2
Selection Sort	66	4	6	306.7
Bubble Sort	38	4	3	48.8
Array Operations	41	1	0	6.7

First, we implemented a mechanism for users to express additional relational and non-relational predicates over pairs of procedures in a separate file. We provide an option to only consider these predicates or augment the auto-generated predicate set with these additional predicates. Second, for each product procedure `MS_v1.f_v2.f`, SymDiff now explores arbitrary Boolean combination (full predicate abstraction) over predicates for constructing mutual pre- and postconditions. The atomic predicates include all the relational (mutual) predicates over inputs (for preconditions) and both inputs and outputs (for two-state postconditions), in addition to any predicates manually specified by the user. The inference is performed by leveraging the new predicate abstraction implementation in Boogie [19].[3] We also attempted using *Duality* interpolation based inference [20]. However, Duality currently diverges for even the simplest of examples (e.g., Swish++) due to its inability to infer relational (cross-program) relationships.

B. Experiments

Table I lists our benchmarks and presents the results of verifying them using SymDiff. It includes all examples described in the previous sections and *Array Operations* example performing mapping over all elements of an array. (See Appendix B for our exercise of checking control flow equivalence on C benchmarks using bounded verification and comparing against taint analysis.) First, note that only *Water* can be verified completely automatically (even without our additions to SymDiff). Second, most of the examples require the support for disjunctions in the invariants.

The need for manual predicates can be broken down into roughly three categories: (i) non-relational predicate such as $\texttt{v2.num\_r} \geq 10$ (*Swish*), (ii) non-trivial relational predicates that require arithmetic such as $\texttt{v1.max} \leq \texttt{v2.max} + \texttt{e}$ (*LU*), and (iii) specialized predicates such as $ArrayEqAfter(\texttt{v1.arr}, \texttt{v2.arr}, \texttt{v1.i})$ (for all the control equivalence examples). Alternately, for *Swish* and *LU*, reusing the property $\texttt{RelaxedEq}(\texttt{v1.x}, \texttt{v2.x})$ as an atomic predicate (instantiated on variables in scope) suffices as well (no additional predicates or disjunction needed). This points to the potential benefit of performing property-directed predicate discovery to improve automation on these examples. Finally, we usually specify the same syntactic predicate for both precondition and postcondition for tail-recursive procedures extracted from loops — handling loops directly will halve the number of manual predicates.

Disjunctions in the sorting examples result from the need to construct complex two-state mutual summaries for product procedures that typically have the form $\mathbf{old}(\phi_1 \wedge \ldots \phi_k) \Rightarrow \psi$. For *Selection Sort*, there are around 15 predicates for such procedures thereby causing large runtime overhead in inferring the strongest inductive invariant. Manually specifying the subset of relevant predicates reduces the runtime significantly to the order of 10s of seconds for this example. Again, this shows the potential for more goal-directed inference of weaker invariants (e.g., CEGAR [21], interpolants [6]) for scaling to larger examples.

VII. Related Work

A number of complementary approaches have been recently proposed to reason about approximations. These approaches can be roughly categorized (with overlaps) into (i) language based, (ii) static analysis, and (iii) dynamic approaches. Language based approaches propose language constructs and annotations to make approximations explicit in a program [2], [4]. EnerJ [2] introduces approximate types and ensures that such values do not impact precise computations, including conditional statements. Our work can be used to improve the precision of the type-based analysis, as demonstrated in § IV. Carbin et al. [4] describe language constructs for introducing approximations and relaxed specifications (based on *relational Hoare logic* [22]), and prove correctness of transformations using Coq [23]. We show that mutual summaries and SMT-based verification can significantly improve the automation for most transformations covered by this approach.

Rely [3] performs static quantitative reliability analysis to provide probabilistic guarantees on the impact of approximations on overall behavior of a program. We believe that SymDiff can be augmented with this framework to create an automated framework for improving precision using relative invariants. ExPAX [24] is a framework that generates a set of safe-to-approximate operations based on a dataflow taint analysis. It develops an algorithm to compute the approximation level for each operation in the set so that energy consumption is minimized and reliability constraints are satisfied.

Among dynamic approaches, fault injection at the source or intermediate representation level has been used to profile the sensitivity of output quality to approximations. Fault injectors such as KULFI [25], LLFI [26], and PDSFIS [27] approximate instructions at runtime. Though these techniques achieve high levels of accuracy, they provide no formal coverage guarantees, unlike our approach. Offline dynamic analysis techniques provide information on dataflow and correlation

[3]We are grateful to Akash Lal for his implementation and help with this feature in Boogie.

difference (e.g., [28], [29]). The former may be imprecise as it is based on static dataflow analysis, while the latter again does not provide formal guarantees. Although there are optimizations for selective instruction perturbation, such as statistical methods [30], the reasoning is only for a subset of all the possible executions of the program.

Finally, our work is closely related to previous works on translation validation [31], [32] that validate equivalence-preserving intraprocedural compiler transformations, using lock-step symbolic execution and SMT solvers. However, mutual summaries and the product construction allows for richer relaxed specifications other than equivalence, interprocedural reasoning [8], and leveraging off-the-shelf program verifiers and inference engines.

VIII. Conclusions

In this paper, we have described the application of automated differential verification for providing formal guarantees of approximations. The structural similarity between original and approximate programs are leveraged to automate most intermediate relative specifications. We are currently working on automating predicate generation, more expressive inference engines such as interpolants [6] and indexed predicate abstraction [33] to infer remaining specifications. We would also like to leverage the concept of relative termination [8] to improve on the partial correctness guarantees of mutual summaries.

References

[1] L. Kugler, "Is "good enough" computing good enough?" *Commun. ACM*, vol. 58, no. 5, pp. 12–14, Apr. 2015.

[2] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 164–174.

[3] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2013, pp. 33–52.

[4] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 169–180.

[5] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard: Version 2.0," in *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[6] K. L. McMillan, "An interpolating theorem prover," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004, pp. 16–30.

[7] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, "Differential assertion checking," in *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 345–355.

[8] C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebelo, "Towards modularly comparing programs using automated theorem provers," in *International Conference on Automated Deduction (CADE)*. Springer, 2013, pp. 282–299.

[9] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "SymDiff: A language-agnostic semantic diff tool for imperative programs," in *International Conference on Computer Aided Verification (CAV)*, 2012, pp. 712–717.

[10] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1995, pp. 49–61.

[11] S. Horwitz, T. W. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.

[12] M. Rinard, "Acceptability-oriented computing," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003, pp. 221–239.

[13] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects (FMCO)*, 2006, pp. 364–387.

[14] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *International Symposium of Formal Methods Europe (FME)*, 2001, pp. 500–517.

[15] S. K. Lahiri, S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies, "Intra-module inference," in *International Conference on Computer Aided Verification (CAV)*, 2009, pp. 493–508.

[16] "SciMark 2.0," http://math.nist.gov/scimark2.

[17] J. Nelson, A. Sampson, and L. Ceze, "Dense approximate storage in phase-change memory," in *Ideas and Perspectives session at ASPLOS*, 2001.

[18] W. Blume and R. Eigenmann, "Performance analysis of parallelizing compilers on the perfect benchmarks programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 6, pp. 643–656, Nov. 1992.

[19] A. V. Thakur, A. Lal, J. Lim, and T. W. Reps, "Posthat and all that: Automating abstract interpretation," *Electr. Notes Theor. Comput. Sci.*, vol. 311, pp. 15–32, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2015.02.003

[20] K. L. McMillan, "Lazy annotation revisited," in *International Conference on Computer Aided Verification (CAV)*, 2014, pp. 243–259.

[21] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.

[22] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004, pp. 14–25.

[23] "The Coq proof assistant," http://coq.inria.fr.

[24] J. Park, K. Ni, X. Zhang, H. Esmaeilzadeh, and M. Naik, "Expectation-oriented framework for automating approximate programming," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[25] V. C. Sharma, A. Haran, Z. Rakamarić, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013, pp. 41–50.

[26] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[27] A. Jin, J. Jiang, J. Hu, and J. Lou, "A pin-based dynamic software fault injection system," in *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*. IEEE, 2008, pp. 2160–2167.

[28] M. F. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Dynamic analysis of approximate program quality," University of Washington, Tech. Rep. UW-CSE-14-03-01.

[29] M. F. Ringenburg, A. Sampson, L. Ceze, and D. Grossman, "Profiling and autotuning for energy-aware approximate programming," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[30] P. Roy, R. Ray, C. Wang, and W.-F. Wong, "ASAC: Automatic sensitivity analysis for approximate computing," in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2014, pp. 95–104.

[31] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1998, pp. 151–166.

[32] G. C. Necula, "Translation validation for an optimizing compiler," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000, pp. 83–94.

[33] S. K. Lahiri and R. E. Bryant, "Predicate abstraction with indexed predicates," *ACM Trans. Comput. Log.*, vol. 9, no. 1, 2007.

[34] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[35] H. R. Myler and A. R. Weeks, *The pocket handbook of image processing algorithms in C*. Prentice Hall Press, 2009.

[36] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić, "A reachability predicate for analyzing low-level software," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007, pp. 19–33.

```
 1  procedure MS_v1.F_v2.F(v1.x:int, v2.x:int)
 2  modifies v1.g, v2.g;
 3  requires v1.x >= 0; // intermediate contract (manual)
 4  {
 5    // initialize call witness variables
 6    v1.b_1, v2.b_1 := false, false;
 7    // v1
 8    if (v1.x < 100) {
 9      v1.g := v1.g + v1.x;
10      v1.i_1, v1.gi_1 := v1.x + 1, v1.g; // store inputs
11      call v1.F(v1.x + 1);
12      v1.b_1 := true;      // record call
13      v1.go_1 := v1.g;     // store outputs
14    }
15    // v2
16    if (v2.x < 100) {
17      v2.g := v2.g + 2*v2.x;
18      v2.i_1, v2.gi_1 := v2.x + 1, v2.g; // store inputs
19      call v2.F(v2.x + 1);
20      v2.b_1 := true;      // record call
21      v2.go_1 := v2.g;     // store outputs
22    }
23    // constrain calls
24    if (v1.b_1 && v2.b_1) { // for pair of calls
25      v1.st_g, v2.st_g := v1.g, v2.g; // store globals
26      v1.g, v2.g := v1.gi_1, v2.gi_1;
27      call MS_v1.F_v2.F(v1.i_1, v2.i_1);
28      assume (v1.g == v1.go_1);  // constrain outputs
29      assume (v2.g == v2.go_2);  // constrain outputs
30      v1.g, v2.g := v1.st_g, v2.st_g; // restore globals
31    }
32    return;
33  }
34
35  procedure MS_v1.Main_v2.Main()
36  modifies v1.g, v2.g;
37  requires v1.g == v2.g; // globals start equal (automatic)
38  ensures  v1.g <= v2.g; // mutual postcondition (manual)
39  {
40  ...
41  }
```

Fig. 9: Composed program for example in Fig. 4. Underlined statements correspond to constituent procedures.

Appendix A
Product program details

We briefly sketch the important components of this construction:

- Line 6 initializes a list of *call witness* variables, one per call-site within v1.F and v2.F respectively.
- Lines 8–14 inline the body of v1.F, whose statements are underlined. Each procedure call (e.g., line 11) is instrumented so that the inputs and outputs are recorded in local variables and the witness variable for the call-site is set.
- Lines 16–22 do the same for v2.F.
- Lines 24–31 are the most interesting part. First, we test using the witness variables if a pair of callees (v1.F, v2.F) has been executed. If so, we call the joint procedure for the callee-pair MS_v1.F_v2.f with the stored arguments and globals. The recursive call to MS_v1.F_v2.f in line 27 results from the recursive calls to F in the two versions. The assume statements after the call constrain the earlier output values of the two callees. Finally, the globals are restored back to the state before the recursive call to MS_v1.F_v2.f.

Appendix B
Control Flow Equivalence: C Benchmarks

We have also performed preliminary experiments to determine the feasibility of using *bounded* differential analysis to infer the set of approximations that do not impact control flow. The main objective is to compare our differential analysis with a more traditional *taint* analysis [34] on realistic benchmarks. The taint analysis for Boogie programs (also implemented in SymDiff) checks if a statement lies in the slice of any of the conditionals using interprocedural dataflow analysis. To develop an automated differential analysis (Diff-Inline), we inline procedure calls (up to a small bound, say 10) before checking the mutual summary specifications — this leads to an unsound analysis in the presence of loops and recursion. In essence, the taint analysis and Diff-Inline provide respectively a lower and upper bound on the number of statements that can be safely approximated without impacting control flow.

TABLE II: Experimental results for control flow equivalence. LOC is the number of lines of code; #P is the number of procedures; #Locs. is the number of program locations that could potentially be approximated; #Inline is the chosen inlining bound; #Approx. is the reported number of locations that can be approximated (i.e., those that do not affect control flow when approximated); Time is cumulative runtime in minutes.

Benchmark	LOC	#P	#Locs.	Diff-Inline			Taint	
				#Inline	#Approx.	Time	#Approx.	Time
Insertion Sort	24	2	13	10	1	1.3	1	1.1
Bubble Sort	25	2	13	10	1	1.6	1	0.8
Selection Sort	30	2	15	10	2	1.8	1	1.9
Brightness Correction	21	1	8	10	4	1.4	4	0.4
Arithmetic Mean Filter	27	1	13	10	5	1.3	5	2.5
Centroid Computation	55	3	30	10	14	8.3	14	3.3
Matrix Multiplication	38	3	17	16	7	5.4	7	2.9
Linked List Operations	76	5	40	6	7	55.4	2	0.8
Array Operations	78	7	35	10	12	115.4	3	2.5

```
var array: [int]int;
const n:int;

procedure max_of(x:int, y:int) returns (r:int) {
  if(x > y) {
    r := 1;
    return;
  }
  if(x == y) {
    r := 0;
    return;
  }
  r := -1;
  return;
}

procedure modify_each_element(value:int) {
  var i, tmp : int;
  i, tmp := 0, 0;
  while (i < n) {
    call tmp := max_of(array[i], value);
    havoc tmp; // approximation: return value of
               // max_of stored in unreliable memory
    array[i] := tmp;
    i := i + 1;
  }
}
```

Fig. 10: Array operations example. Taint analysis is imprecise on this example since it cannot precisely track array segments.

We have chosen a set of 9 realistic C programs including sorting, image processing [35], data structure implementations, and operations on matrices. Table II summarizes our benchmarks. We first translate them into Boogie programs using the HAVOC verifier [36]. In our experiments, an approximation is modeled as just a **havoc** statement (of the appropriate variable) introduced at every program location of interest (i.e., variable assignments). We then establish control flow equivalence for every such approximation in turn, and we report total cumulative runtimes.

Table II presents the results of our experiments. Overall, benchmarks from the domain of image processing are most amenable to approximations that do not affect control flow, since most computations are local to a pixel neighborhood. As expected, the inlining-based approach scales poorly compared to the modular taint analysis. However, it is encouraging to see that the differential analysis improves the precision on three benchmarks. Among these, we have studied the *Selection Sort* example in detail (§ IV-B), and have applied the sound differential verification on *Selection Sort* and *Array Operations* (§ VI-B). The Boogie source code of *Array Operations* is shown in Fig. 10. This exercise illustrates the robustness of our differential approach to verify more complex examples, albeit with a little more user effort.