Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation

Patrice Godefroid[1], Shuvendu K. Lahiri[1], and Cindy Rubio-González[2]

[1] Microsoft Research, Redmond, WA, USA
[2] University of Wisconsin, Madison, WI, USA

Abstract. Compositional dynamic test generation can achieve significant scalability by memoizing symbolic execution sub-paths as test summaries. In this paper, we formulate the problem of statically validating symbolic test summaries against code changes. Summaries that can be proved still valid using a static analysis of a new program version do not need to be retested or recomputed dynamically. In the presence of small code changes, incrementality can considerably speed up regression testing since static checking is much cheaper than dynamic checking and testing. We provide several checks ranging from simple syntactic ones to ones that use a theorem prover. We present preliminary experimental results comparing these approaches on three large Windows applications.

1 Introduction

Whitebox fuzzing [15] is a promising new form of security testing based on dynamic test generation [5, 14]. Dynamic test generation consists of running a program while simultaneously executing the program symbolically in order to gather constraints on inputs from conditional statements encountered along the execution. Those constraints are then systematically negated and solved with a constraint solver, generating new test inputs to exercise different execution paths of the program. Over the last couple of years, whitebox fuzzing has extended the scope of dynamic test generation from unit testing to whole-program security testing, thanks to new techniques for handling very long execution traces (with billions of instructions). In the process, whitebox fuzzers have found many new security vulnerabilities (buffer overflows) in Windows [15] and Linux [21] applications, including codecs, image viewers and media players. Notably, our whitebox fuzzer SAGE found roughly one third of *all* the bugs discovered by file fuzzing during the development of Microsoft's Windows 7 [12]. Since 2008, SAGE has been continually running on average 100+ machines automatically "fuzzing" hundreds of applications in a dedicated security testing lab. This represents the largest computational usage ever for any Satisfiability Modulo Theories (SMT) solver [27], according to the authors of the Z3 SMT solver [8].

Despite these successes, several challenges remain, such as increasing code coverage and bug finding, while reducing computational costs. A key promising idea is *compositionality*: the search process can be made compositional by memoizing symbolic execution sub-paths as test summaries which are re-usable during the search, resulting in a search algorithm that can be exponentially faster than a non-compositional

one [11]. By construction, symbolic test summaries are "must" summaries guaranteeing the existence of some program executions and hence useful for proving *existential* reachability properties (such as the existence of an input leading to the execution of a specific program branch or bug). They dualize traditional "may" summaries used in static program analysis for proving *universal* properties (such as the absence of specific types of bugs for all program paths). We are currently building a general infrastructure to generate, store and re-use symbolic test summaries for large parts of the Windows operating system.

In this context, an important problem is the maintenance of test summaries as the code under test slowly evolves. Recomputing test summaries dynamically from scratch for every program version sounds wasteful, as new versions are frequent and much of the code has typically not changed. Instead, *whenever possible*, it could be *much cheaper* to *statically* check whether previously-computed symbolic test summaries are still valid for the new version of the code. The formalization and study of this problem is the motivation for this paper.

We introduce the *must-summary checking problem*:

> *Given a set S of symbolic test summaries for a program Prog and a new version $Prog'$ of Prog, which summaries in S are still valid must summaries for $Prog'$?*

We also consider the more general problem of checking whether an arbitrary set S of summaries are valid must summaries for an arbitrary program *Prog*.

We present three algorithms with different precision to statically check which old test summaries are still valid for a new program version. First, we present an algorithm (in Section 3) based on a simple impact analysis of code changes on the static control-flow and call graphs of the program; this algorithm can identify local code paths that have not changed and for which old summaries are therefore still valid. Second, we present (in Section 4) a more precise predicate-sensitive refined algorithm using verification-condition generation and automated theorem proving. Third, we present an algorithm (in Section 5) for checking the validity of a symbolic test summary against a program regardless of program changes, by checking whether the pre/postconditions captured in the old summary still hold on the new program. We discuss the strengths and weaknesses of each solution, and present preliminary experimental results with sample test summaries generated for three large Windows applications. These experiments confirm that hundreds of summaries can be validated statically in minutes, while validating those dynamically can require hours or days.

2 Background and Problem Definition

2.1 Background: Compositional Symbolic Execution

We assume we are given a sequential program *Prog* with input parameters I. Dynamic test generation [14] consists of running the program *Prog* both concretely and symbolically, in order to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution. For each execution path w, i.e., a sequence of statements executed by the program, a *path constraint* ϕ_w is constructed that characterizes the input values for which the program executes along w. Each variable appearing

```
#define N 100
void P(int s[N]) { // N inputs
  int i, cnt = 0;
  for (i = 0; i < N; i++)
    cnt = cnt + is_positive(s[i]);
  if (cnt == 3) error(); // (*)
  return;
}
```

```
int is_positive(int x) {
  if (x > 0) return 1;
  return 0;
}
int g(int x, int y) {
  if ((x > 0)&&(hash(y) > 10))
    return 1;
  return 0;
}
```

Fig. 1. Example.

in ϕ_w is thus a program input. Each constraint is expressed in some theory T decided by a constraint solver (for instance, including linear arithmetic, bit-vector operations, etc.). A constraint solver is an automated theorem prover which also returns a satisfying assignment for all variables appearing in constraints it can prove satisfiable. All program paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths w for which ϕ_w is satisfiable are *feasible* and are the only ones that can be executed by the actual program provided the solutions to ϕ_w characterize exactly the inputs that drive the program through w. Assuming that the constraint solver used to check the satisfiability of all formulas ϕ_w is sound and complete, this use of symbolic execution for programs with finitely many paths amounts to program verification.

Systematically testing and symbolically executing *all* feasible program paths does not scale to large programs. Indeed, the number of feasible paths can be exponential in the program size, or even infinite in the presence of loops with unbounded number of iterations. This *path explosion* [11] can be alleviated by performing symbolic execution *compositionally* [2, 11].

Let us assume the program *Prog* consists of a set of functions. In the rest of this section, we use the generic term of *function* to denote any part of the program *Prog* whose observed behaviors are summarized; any program fragments can be treated as "functions" as will be discussed later. To simplify the presentation, we assume the functions in *Prog* do not perform recursive calls, and that all the executions of *Prog* terminate. These assumptions do not prevent *Prog* from having infinitely many executions paths if it contains a loop whose number of iterations depends on some unbounded input.

In compositional symbolic execution, a function summary ϕ_f for a function f is defined as a logic formula over constraints expressed in theory T. ϕ_f can be derived by successive iterations and defined as a *disjunction* of formulas ϕ_{w_f} of the form $\phi_{w_f} = pre_{w_f} \wedge post_{w_f}$, where w_f denotes an intraprocedural path inside f, pre_{w_f} is a conjunction of constraints on the inputs of f, and $post_{w_f}$ is a conjunction of constraints on the outputs of f. An input to a function f is any value that can be read by f, while an output of f is any value written by f. ϕ_{w_f} can be computed automatically from the path constraint for the intraprocedural path w_f [2, 11].

For instance, given the function `is_positive` in Figure 1, a summary ϕ_f for this function can be

$$\phi_f = (x > 0 \wedge ret = 1) \vee (x \leq 0 \wedge ret = 0)$$

where *ret* denotes the value returned by the function.

Symbolic variables are associated with function inputs (like x in the example) and function outputs (like *ret* in the example), in addition to whole-program inputs. In order to generate a new test to cover a new branch b in some function, *all* the previously known summaries can be used to generate a formula ϕ_P representing symbolically all the paths known so far during the search. By construction [11], symbolic variables corresponding to function inputs and outputs are all bound in ϕ_P, and the remaining free variables correspond exclusively to whole-program inputs (since only those can be controlled for test generation).

For instance, for the program `P` in Figure 1, a formula ϕ_P to generate a test covering the then branch (*) given the above summary ϕ_f for function `is_positive` can be

$$(ret_0 + ret_1 + \ldots + ret_{N-1} = 3) \wedge \bigwedge_{0 \le i < N} ((s[i] > 0 \wedge ret_i = 1) \vee (s[i] \le 0 \wedge ret_i = 0))$$

where ret_i denotes the return value of the ith call to function `is_positive`. Even though program `P` has 2^{N+1} feasible whole-program paths, compositional test generation can cover "symbolically" all those paths in at most 4 test inputs: 2 tests to cover both branches in function `is_positive` plus 2 tests to cover both branches of the conditional statement (*). Compositionality avoids an exponential number of tests and calls to the constraint solver, at the cost of using more complex formulas with more disjunctions.

2.2 Problem Definition: Must Summary Checking

In practice, symbolic execution of large programs is bound to be imprecise due to complex program statements (pointer manipulations, floating-point operations, etc.) and calls to operating-system and library functions that are hard to reason about symbolically with good enough precision at a reasonable cost. Whenever precise symbolic execution is not possible during dynamic test generation, concrete values can be used to simplify constraints and carry on with a simplified, partial symbolic execution [14]. The resulting path constraints are then *under-approximate*, and summaries become *must* summaries.

For example, consider the function `g` in Figure 1 and assume the function `hash(y)` is a complex or unknown function for which no constraint is generated. Assume we observe at runtime that when `g` is invoked with $y = 45$, the value of `hash(45)` is 987. The summary for this execution of function `g` can then be

$$(x > 0 \wedge y = 45 \wedge ret = 1)$$

Here, symbolic variable y is constrained to be equal to the concrete value 45 observed along the run because the expression $hash(y)$ cannot be symbolically represented. This summary is a must summary since all value pairs (x, y) that satisfy its precondition define executions of `g` that satisfy the postcondition $ret = 1$. However, this set is a subset of all value pairs that satisfy this postcondition assuming there exists some other value of y different from 45 such that $hash(y) > 10$. For test generation purposes, we safely under-approximate this perfect but unknown input set with the smaller precondition $x > 0 \wedge y = 45$. A must summary can thus be viewed as an abstract witness of some

execution. Must summaries are useful for bug finding and test generation, and dualize may summaries for proving correctness, i.e., the absence of bugs.

We denote a must summary by a quadruple $\langle lp, P, lq, Q \rangle$ where lp and lq are arbitrary program locations, P is a summary precondition holding in lp, and Q is a summary postcondition holding in lq. lp and lq can be anywhere in the program: for instance, they can be the entry and exit points of a function (as in the previous examples) or block, or two program points where consecutive symbolic constraints are injected in the path constraint during symbolic execution, possibly in different functions. In what follows, we call a summary *intraprocedural* if its locations (lp, lq) are in a same function f and the function f did not return between lp to lq when the summary was generated (i.e., no instruction from a function calling f higher in the call stack was executed from lp to lq when the summary was generated). We will only consider intraprocedural summaries in the remainder of this paper, unless otherwise specified.

Formally, must summaries are defined as follows.

Definition 1. *A must summary $\langle lp, P, lq, Q \rangle$ for a program Prog implies that, for every program state satisfying P at lp in Prog, there exists an execution that visits lq and satisfies Q at lq.*

A must summary is called *valid* for a program *Prog* if it satisfies Definition 1. We define the *must-summary checking problem* as follows.

Definition 2. *(Must-summary checking)* Given a valid must summary $\langle lp, P, lq, Q \rangle$ for a program *Prog* and a new version *Prog′* of *Prog*, is $\langle lp, P, lq, Q \rangle$ still valid for *Prog′*?

We also consider later in Section 5 the more general problem of checking whether an arbitrary must summary is *valid* for an arbitrary program *Prog*. These problems are different from the *must summary inference/generation* problem discussed in prior work [2, 11, 16].

We present three different algorithms for statically checking which old must summaries are still valid for a new program version. These algorithms can be used in isolation or in a pipeline, one after another, in successive "phases" of analysis.

3 Phase 1: Static Change Impact Analysis

The first "Phase 1" algorithm is based on a simple impact analysis of code changes in the static control-flow and call graphs of the program.

A sufficient condition to prove that an old must summary $\langle lp, P, lq, Q \rangle$ generated as described in Section 2.1 is still valid in a new program version is that *all* the instructions that were executed in the original program path taken between lp and lq when the summary was generated remain unchanged in the new program. Recording all unique instructions executed between each pair (lp, lq) would be expensive for large programs as many instructions (possibly in other functions) can be executed.

Instead, we can *over-approximate* this set by statically finding all program instructions that *may* be executed on *all* paths from lp to lq: this solution requires no additional storage of runtime-executed instructions but is less precise. If no instruction in this larger set has changed between the old and new programs, any summary for (lp, lq) can

then be safely reused for the new program version; otherwise, we have to conservatively declare the summary as potentially invalid since a modified instruction might be on the original path taken from *lp* to *lq* when the summary was generated.

To determine whether a specific instruction in the old program is unchanged in the new program, we rely on an existing lightweight syntactic "diff"-like tool which can (conservatively) identify instructions that have been modified, deleted or added between two program versions by comparing their abstract syntax trees.

Precisely, an instruction i of a program $Prog$ is defined as *modified* in another program version $Prog'$ if i is changed or deleted in $Prog'$ or if its ordered set of immediate successor instructions changed between $Prog$ and $Prog'$. For instance, swapping the then and else branches of a conditional jump instruction "modifies" the instruction. However, the definition is local as it does not involve non-immediate successors.

Program instructions that are not modified can be mapped across program versions. Conversely, if an instruction cannot be mapped across program versions, it is considered as "deleted" and therefore *modified*. Similarly, a program function is defined as *modified* if it contains either a modified instruction, or a call to a modified function, or a call to an unknown function (e.g., a function outside the program or through a function pointer which we conservatively assume may have been modified). Note that this definition is transitive, unlike the definition of modified instruction.

Given those definitions, we can soundly infer valid summaries using the following rule.

> An intraprocedural summary from *lp* to *lq* inside a same function f is *valid* if, in the control-flow graph for f, no instruction between *lp* and *lq* is modified or is a call to a modified function.

The correctness of this rule is immediate for intraprocedural summaries (as defined in Section 2.2) since, if the condition stated in the rule holds, we know that all instructions between *lp* and *lq* are unchanged across program versions.

Implementing this rule requires building the control-flow graph of every function containing an old intraprocedural summary and the call graph for the entire program in order to transitively determine which functions are modified. Note that the precision of the rule above could be improved by considering interprocedural control-flow graphs (merging together multiple intraprocedural control-flow graphs), at the cost of building larger graphs.

4 Phase 2: Predicate-Sensitive Change Impact Analysis

Consider the summary $\langle lp,\ x > 0 \wedge y < 10,\ lq,\ w = 0 \rangle$ for the code fragment shown on the left of Figure 2. Assume the instructions marked with "MODIFIED" have been modified in the new version. Since some instructions on some paths from *lp* to *lq* have been modified, the Phase 1 analysis will invalidate the summary. However, notice that the set of executions that start from a state satisfying $x > 0 \wedge y < 10$ at *lp* and reach *lq* has not changed.

In this section, we present a second change impact analysis "Phase 2" that exploits the predicates P and Q in a summary $\langle lp,\ P,\ lq,\ Q \rangle$ to perform a more refined analysis. The basic idea is simple: instead of considering all the paths between *lp* and *lq*,

```
    ...
lp: if (x > 0) {
      if (y == 10)
        w++; // MODIFIED
      else
        w = 0;
    } else {
        w = 1; // MODIFIED
    }
lq: ...
```

```
    ...
lp: if (x < 0) {
      if (y < 0)
        r = 1;
      else {
        r = 0; //MODIFIED to r = 4;
       }
   }
lq: ...
```

Fig. 2. Motivating examples for Phase 2 (left) and Phase 3 (right).

we only consider those that also satisfy P in *lp* and Q in *lq*. We now describe how to perform such a predicate-sensitive change impact analysis using static verification-condition generation and theorem proving. We start with a program transformation for checking that all executions satisfying P in *lp* that reach *lq* and satisfy Q in *lq* are not modified from *lp* to *lq*.

Given an intraprocedural summary $\langle lp, P, lq, Q\rangle$ for a function f, we modify the body of f in the *old* code as follows. Let *Entry* denote the location at the beginning of f, i.e., just before the first instruction executed in f. We use an auxiliary Boolean variable modified, and insert the following code at the labels *Entry*, *lp*, *lq* and at all labels ℓ corresponding to a *modified* instruction or a call to a modified function (just before the instruction at that location).

$$
\begin{aligned}
\mathit{Entry}:&\ \mathsf{goto}\ \mathit{lp};\\
\mathit{lp}:&\ \mathsf{assume}\ P;\ \mathsf{modified} := \mathsf{false};\\
\mathit{lq}:&\ \mathsf{assert}\ (Q \implies \neg\mathsf{modified});\\
\ell:&\ \ \mathsf{modified} := \mathsf{true};
\end{aligned}
$$

The assume statement assume P at *lp* is a blocking instruction [4], which acts as a no-op if control reaches the statement in a state satisfying the predicate P, and blocks the execution otherwise. The assertion at *lq* checks that if an execution reaches *lq* where it satisfies Q via *lp* where it satisfied P, it does not execute any *modified* instruction between *lp* and *lq*.

Theorem 1. *Given an intraprocedural must summary $\langle lp, P, lq, Q\rangle$ valid for a function f in an old program Prog, if the assertion at lq holds in the instrumented old program for all possible inputs for f, then $\langle lp, P, lq, Q\rangle$ is a valid must summary for the new program Prog′.*

Proof. The assertion at *lq* ensures that *all* executions in the old program *Prog* that (1) reach *lq* and satisfy Q in *lq* and (2) satisfy P at *lp* do not execute any instruction that is marked as *modified* between *lp* and *lq*. This set of executions is possibly over-approximated by considering *all* possible inputs for f, i.e., ignoring specific calling contexts for f and *lp* in *Prog*. Since all the instructions executed from *lp* to *lq* during those executions are preserved in the new program *Prog′*, all those executions W from

lp to *lq* are still possible in the new program. Moreover, since $\langle lp, P, lq, Q\rangle$ is a must summary for the old program *Prog*, we know that for every state s satisfying P in *lp*, there exists an execution w from s that reaches *lq* and satisfies Q in *lq* in *Prog*. This execution w is included in the set W preserved from *Prog* to *Prog′*. Therefore, by Definition 1, $\langle lp, P, lq, Q\rangle$ is a valid must summary for *Prog′*. □

The reader might wonder the reason for performing the above instrumentation on the old program *Prog* instead of on the new program *Prog′*. Consider the case of a state that satisfies P at *lp* from which there is an execution that reaches *lq* in *Prog*, but from which no execution reaches *lq* in *Prog′*. In this case, the must summary $\langle lp, P, lq, Q\rangle$ is invalid for *Prog′*. Yet applying the above program transformation to *Prog′* would not necessarily trigger an assertion violation at *lq* since *lq* may no longer be reachable in *Prog′*.

To validate must summaries statically, one can use any static assertion checking tool to check that the assertion in the instrumented program does not fail for all possible function inputs. In this work, we use Boogie [3], a verification condition (VC) based program verifier to check the absence of assertion failures. VC-based program verifiers create a logic formula from a program with assertions with the following guarantee: if the logic formula is valid, then the assertion does not fail in any execution. The validity of the logic formula is checked using a theorem prover, typically a SMT solver. For loop-free and call-free programs, the logic formula is generated by computing variants of *weakest liberal preconditions* (*wlp*) [9]. Procedure calls can be handled by assigning non-deterministic values to the return variable and all the globals that can be potentially modified during the execution of the callee. Similarly, loops can be handled by assigning non-deterministic values to all the variables that can be modified during the execution of the loop. Although procedure postconditions and loop invariants can be used to recover the loss of precision due to the use of non-determinism for over-approximating side effects of function calls and loop iterations, we use the default postcondition and loop invariant true for our analysis to keep the analysis automated and simple.

5 Phase 3: Must Summary Validity Checking

Consider the code fragment shown on the right of Figure 2 where the instruction marked "MODIFIED" is modified in the new code. Consider the summary $\langle lp,\ x < 0,\ lq,\ r \geq 0\rangle$. Since the *modified* instruction is along a path between *lp* and *lq*, even when restricted under the condition P at *lp*, neither Phase 1 nor Phase 2 will validate the summary. However, note that the change does not affect the validity of the must summary: all executions satisfying $x < 0$ at *lp* still reach *lq* and satisfy $r \geq 0$ in the new code, which means the must summary is still valid. In this section, we describe a third algorithm dubbed "Phase 3" for statically checking the validity of a must summary $\langle lp, P, lq, Q\rangle$ against some code, *independently of code changes*.

In the rest of this section, we assume that the programs under consideration are (i) *terminating*, i.e., every execution eventually terminates, and (ii) *complete*, i.e., every state has a successor state.

Given an intraprocedural summary $\langle lp, P, lq, Q\rangle$ for a function f, we perform the following instrumentation on the *new* code. We denote by *Entry* the location of the

first instruction in f, while *Exit* denotes any exit instruction in f. We use an auxiliary Boolean variable reach_lq, and insert the following code at the labels *Entry*, *lp*, *lq* and *Exit*.

$$
\begin{aligned}
\mathit{Entry}: &\ \mathsf{reach\_lq} := \mathsf{false};\ \mathsf{goto}\ \mathit{lp};\\
\mathit{lp}: &\ \mathsf{assume}\ P;\\
\mathit{lq}: &\ \mathsf{assert}\ (Q);\ \mathsf{reach\_lq} := \mathsf{true};\\
\mathit{Exit}: &\ \mathsf{assert}\ (\mathsf{reach\_lq});
\end{aligned}
$$

The variable reach_lq is set when *lq* is visited in an execution, and initialized to false at the *Entry* node. The assume P blocks the executions that do not satisfy P at *lp*. The assertion at *lq* checks that if an execution reaches *lq* via *lp*, it satisfies Q. Finally, the assertion at *Exit* checks that *all* executions from *lp* have to go through *lq*.

Theorem 2. *Given an intraprocedural must summary $\langle lp, P, lq, Q\rangle$ for a function f, if the assertions hold in the instrumented program for all possible inputs of f, then $\langle lp, P, lq, Q\rangle$ is a valid must summary for the program.*

Proof. The assertion at *lq* ensures that *every* execution that reaches *lq* from a state satisfying P at *lp*, satisfies Q. This set of executions is possibly over-approximated by considering *all* possible inputs for f, i.e., ignoring specific calling contexts for f and *lp*. Since we consider programs that are terminating and complete, the assertion at *Exit* is checked for every execution (except those blocked by assume P in *lp* which do not satisfy P), and ensures that every execution that satisfies P at *lp* visits *lq*. The goto *lp* ensures that *lp* is reached from *Entry*, otherwise the two assertions could vacuously hold if *lp* was not reachable or through restricted calling contexts smaller than P. □

The assertions in the instrumented function can be checked using any off-the-shelf assertion checker as described in Section 4. Our implementation uses VC generation and a theorem prover to validate the summaries. Since loops and procedure calls are treated conservatively by assigning non-deterministic values to modified variables, the static validation is also approximate and may sometimes fail to validate valid must summaries.

Note that Phase 3 is *not* an instance of the Phase 2 algorithm when every statement is marked as "modified": Phase 3 checks the *new* program while Phase 2 checks the *old* program (see also the remark after Theorem 1).

Moreover, the precision of Phase 3 is incomparable to the precision of Phase 2 (which refines Phase 1). Both Phase 1 and Phase 2 validate a must summary for the new program *assuming* it was a must summary for the old program, whereas Phase 3 provides an absolute guarantee on the new program. At the start of this section, we presented an example of a valid must summary that can be validated by Phase 3 but not by Phase 2. Conversely, Phase 3 may fail to validate a summary due to the presence of complex code between *lp* and *lq* and imprecision in static assertion checking, while Phase 1 or Phase 2 may be able to prove that the summary is still valid by detecting that the complex code has not been modified.

6 Dealing with Partial Summaries

In practice, tracking *all* inputs and outputs of large program fragments can be problematic in the presence of large or complex heap-allocated data structures or when dealing with library or operating-system calls with possibly unknown side effects. In those cases, the constraints P and Q can be approximate, i.e., only *partially defined*: P constraints only *some* inputs, while Q can capture only *some* outputs (side effects). The must summary is then called *partial*, and may be wrong in some other unknown calling context. Constraints containing partial must summaries may generate test cases that will not cover the expected program paths and branches. Such *divergences* [14] can be detected at runtime by comparing the expected program path with the actual program path being taken. In practice, divergences are often observed in dynamic test generation, and partial summaries can still be useful to limit path explosion, even at the cost of some divergences.

Consider the partial summary $\langle lp,\ x > 0,\ lq,\ ret = 1\rangle$ for the function

```
    int k(int x) {
lp:  if ((x > 0) && (vGlobal > 10)) return 1;
     return 0;
lq: }
```

where the input value stored in the global variable `vGlobal` is not captured in the summary, perhaps because it does not depend on a whole-program input. If the value of `vGlobal` is constant, the constraint `(vGlobal > 10)` is always true and can safely be skipped. Otherwise, the partial summary is imprecise: it may be wrong in some calling contexts.

The validity of partial must summaries could be defined in a weaker manner to reflect the fact that they capture only partial preconditions, for instance as follows:

Definition 3. *A* partial *must summary $\langle lp, P, lq, Q\rangle$ is valid for a program Prog if there exists a predicate R on program variables, such that (i) R does not imply* false*, (ii) the support*[1] *of R is disjoint from the support of P, and (iii) $\langle lp, P \wedge R, lq, Q\rangle$ is a must summary for Prog.*

Since R is not false, the conditions (ii) and (iii) cannot be vacuously satisfied. Moreover, since the supports of P and R are disjoint, R does not constrain the variables in P yet requires that the partial must summary tracks a subset of the inputs (namely those appearing in P) precisely.

In practice, it can be hard and expensive to determine whether a must summary is partial or not. Fortunately, any partial must summary can be soundly validated using the stronger Definition 1, which is equivalent to setting R to true in Definition 3. Phases 1, 2 and 3 are thus all sound for validating partial must summaries.

Validating partial summaries with Definition 3 or full summaries for non-deterministic programs with Definition 1 could be done more precisely with an assertion checker that can reason about alternating existential and universal quantifiers, which is non-standard. It would be interesting to develop such an assertion checker in future work.

[1] The support of an expression refers to the variables in the expression.

7 Recomputing Invalidated Summaries

All the summaries declared valid by Phase 1, 2 or 3 are mapped to the new code and can be reused. In contrast, all invalid summaries need to be recomputed, for instance using a breadth-first strategy in the graph formed by superposing path constraints.

Consider the graph G whose nodes are all the program locations lp and lq mentioned in the old set of test summaries, and where there is an edge from lp to lq for each summary. Note that, by construction [11], every node lq of a summary matches the node lp of the next summary in the whole-program path constraint, unless lq is the last conditional statement in the path constraint or lp is the first one, which we denote by r for "root". By construction, G is a directed acyclic graph.

Consider any invalid summary $\langle lp, P, lq, Q\rangle$ that is closest to the root r of G. Let $\mathcal{P}$ denote the set of paths from r to lp. By construction with a breadth-first strategy, all summaries along all the paths in $\mathcal{P}$ are still valid for the new program version. To recompute the summary $\langle lp, P, lq, Q\rangle$ for the new program, we call the constraint solver with the formula

$$P \wedge \bigvee_{\phi_i \in \mathcal{P}} \phi_i$$

in order to generate a test to exercise condition P at the program location lp (see Section 2.1). Then, we run this test against the new program version and generate a new summary from lp to wherever it leads to (possibly a new lq and Q). This process can be repeated to recompute all invalidated summaries in a breadth-first manner in G.

8 Experimental Results

We now present preliminary results for validating intraprocedural must summaries generated by our tool SAGE [15] for several benchmarks, with a focus on understanding the relative effectiveness of the different approaches.

8.1 Implementation

We have developed a prototype implementation for analyzing x86 binaries, using two existing tools: the Vulcan [10] library to statically analyze Windows binaries, and the Boogie [3] program verifier. We briefly describe the implementation of the different phases in this section.

Our tool takes as input the old program (DLLs), the set of summaries generated by SAGE for the old program, and the new version of the program. We use Vulcan to find differences between the two versions of the program, and propagate them interprocedurally. In this work, we focus on the validation of must summaries that are intraprocedural (SAGE classifies summaries as intraprocedural or not at generation time). Intraprocedural summaries that cannot be validated by Phase 1 are further examined by the more precise Phases 2 and 3. For each of those, we conservatively translate the x86 assembly code of the function containing the summary to a function in the Boogie input language, and use the Boogie verifier (which uses the Z3 SMT solver) to validate the

		Functions with Changes								Summaries
Benchmark	Functions	M	% M	IM	% IM	U	% U	IU	% IU	(Intraprocedural)
ANI	6978	703	10%	3130	45%	2340	34%	5174	74%	286
GIF	13897	712	5%	4370	31%	3814	27%	8827	64%	288
JPEG	20357	623	3%	6150	30%	7463	37%	12184	60%	517

Fig. 3. Benchmark characteristics.

summaries using the Phase 2 or Phase 3 checks. Finally, our tool maps the *lp* and *lq* locations of every validated summary from the old program to the new program.

Unfortunately, Boogie currently does not generate a VC if the function under analysis has an *irreducible* control-flow graph [1], although the theory handles it [3]. A function has an irreducible control-flow graph if there is an unstructured loop with multiple entry points into the loop. Such an unstructured loop can arise from two sources: (i) x86 binaries often contain unstructured `goto` statements, and (ii) we add a `goto` *lp* statement in Phases 2 and 3 that might jump inside a loop. Such irreducible graphs appear in roughly 20% of the summaries considered in this section. To circumvent this implementation issue, we report experimental results in those cases where such loops are unrolled a constant number of times (four times). Although we have manually checked that many of these examples will be provable if we had support for irreducible graphs, we can treat those results to indicate the potential of Phase 2 or Phase 3: if their effectiveness is poor after unrolling, it can only be worse without unrolling.

8.2 Benchmarks

Table 3 describes the benchmarks used for our experiments. We consider three image parsers embedded in Windows: ANI, GIF and JPEG. For each of these, we ran SAGE to generate a sample of summaries. The number of DLLs with summaries for the three benchmarks were 3 for ANI, 4 for GIF, and 8 for JPEG. Then, we arbitrarily picked a newer version of each of these DLLs; these were between one and three years newer than the original DLLs. The column "Functions" in Table 3 denotes the total number of functions present in the original DLLs. The columns marked "M", "IM", "U" and "IU" denote the number of functions that are "Modified", "Indirectly Modified" (i.e., calling a modified function), "Unknown" (i.e., calling a function in an unknown DLL or through a function pointer) and "Indirectly Unknown", respectively. The table also contains the percentage of such functions over the total number of functions. Finally, the "Summaries" column denotes the number of summaries classified as intraprocedural. For all three benchmarks, most summaries generated by SAGE are intraprocedural.

Although these benchmarks have a relatively small fraction of modified functions (between 3% – 10%), the fraction of functions that can transitively call into these functions can be fairly large (between 30% – 45%). The impact of unknown functions is even more significant, with most functions being marked U or IU. Note that any call to a M, IM, U or IU function would be marked as *modified* in Phase 1 of our validation algorithm (Section 3). Although we picked two versions of each benchmark separated by more than a year, we expect the most likely usage of our tool to be for program versions separated only by a few weeks.

Benchmark	# Summ	Phase 1			Phase 2			Phase 3			All		
		#	%	time	#	%	time	#	%	time	#	%	time
ANI	286	167	58%	8m (3m)	244	85%	37m	86	30%	42m	256	90%	87m
GIF	288	198	69%	12m (4m)	264	92%	23m	90	31%	35m	274	95%	70m
JPEG	517	317	61%	18m (6m)	487	94%	31m	173	33%	37m	501	97%	86m

Fig. 4. Different phases on all the intraprocedural summaries.

8.3 Results

The three tables (Fig. 4, Fig. 5 and Fig. 6) report the relative effectiveness of the different phases on the previous benchmarks. Each table contains the number of intraprocedural summaries for each benchmark ("# Summ"), the validation done by each of the phases, and the overall validation. For each phase (and overall), we report the number of summaries validated ("#"), the percentage of the total number of summaries validated ("%") and the time (in minutes) taken for the validation. The time reported for Phase 1 includes the time taken for generating the *modified* instructions interprocedurally, and mapping the old summaries to the new code; the fraction of time spent solely on validating the summaries is shown in parenthesis. The failure to prove a summary valid in Phase 2 or Phase 3 could be the result of a counterexample, timeout (100 seconds per summary), or some internal analysis errors in Boogie.

Figure 4 reports the effect of passing *all* the intraprocedural summaries independently to all the three phases. First, note that the total number of summaries validated is quite significant, between 90% and 97%. Phase 1 can validate between 58%–69% of the summaries, Phase 2 between 85%–94% and Phase 3 between 30%–33%. Since Phase 1 is simpler, it can validate the summaries the fastest among the three approaches. The results also indicate that Phase 2 has the potential to validate significantly more summaries than Phase 1 or Phase 3. After a preliminary analysis of the counterexamples for Phase 3, its imprecision seems often due to the partiality of must summaries (see Section 6): many must summaries do not capture enough constraints on states to enable their validation using Phase 3.

To understand the overlap between the summaries validated by each phase, we report the results of the three phases in a "pipeline" fashion, where the summaries validated by an earlier phase are not considered in the later stages. In all the configurations, Phase 1 was allowed to go first because it generates information required for running Phase 2 and Phase 3, and because it is the most scalable as it does not involve a program verifier. The invalid summaries from Phase 1 are passed either to Phase 2 first (Figure 5) or to Phase 3 first (Figure 6).

The results indicate that the configuration of running Phase 1, followed by Phase 2 and then Phase 3 is the fastest. The overall runtime in Figure 5 is roughly half than the overall runtime in Figure 4. Note that the number of additional summaries validated by Phase 3 beyond Phases 1 and 2 is only 1%–4%.

On average from Figure 5, it takes about (43 min divided by 256 summaries) 10 secs to statically validate one summary for ANI, 6 secs for GIF and 5 secs for JPEG. In contrast, the average time needed by SAGE to dynamically re-compute a summary

Benchmark	# Summ	Phase 1			Phase 2			Phase 3			All		
		#	%	time	#	%	time	#	%	time	#	%	time
ANI	286	167	58%	8m	77	27%	29m	12	4%	6m	256	90%	43m
GIF	288	198	69%	12m	73	25%	15m	3	1%	1m	274	95%	28m
JPEG	517	317	61%	18m	179	35%	18m	5	1%	5m	501	97%	41m

Fig. 5. Pipeline with Phase 1, Phase 2 and Phase 3.

Benchmark	# Summ	Phase 1			Phase 3			Phase 2			All		
		#	%	time	#	%	time	#	%	time	#	%	time
ANI	286	167	58%	8m	30	10%	12m	59	21%	27m	256	90%	47m
GIF	288	198	69%	12m	25	9%	7m	51	18%	12m	274	95%	31m
JPEG	517	317	61%	18m	52	10%	14m	132	26%	14m	501	97%	46m

Fig. 6. Pipeline with Phase 1, Phase 3, Phase 2.

from scratch is about 10 secs for ANI, 70 secs for GIF and 100 secs for JPEG. Statically validating summaries is thus up to 20 times faster for these benchmarks.

9 Related Work

Compositional *may static* program analysis has been amply discussed in the literature [25]. A compositional analysis always involves some form of summarization. Incremental program analysis is also an old idea [7, 24] that nicely complements compositionality. Any incremental analysis involves the use of some kind of "derivation graph" capturing inference interdependencies between summaries during their computation, such as which lower-level summary was used to infer which higher-level summary. While compositional interprocedural analysis has now become mainstream in industrial-strength static analysis tools (e.g., [19]) which otherwise would not scale to large programs, incremental algorithms are much less widely used in practice. Indeed, those algorithms are more complicated and often not really needed as well-engineered compositional static analysis tools can process millions of lines of code in only hours on standard modern computers.

The purpose of our general line of research is to replicate the success of compositional static program analysis to the testing space. In our context, the summaries we memoize (cache) are symbolic test must summaries [2, 11] which are general input-dependent pre/postconditions of a-priori arbitrary code fragments, and which are represented as logic formulas that are used by an SMT solver to carry out the interprocedural part of the analysis. Because test summaries need to be precise (compared to those produced by standard static analysis) and are generated during an expensive dynamic symbolic execution of large whole programs, incrementality is more appealing for cost-reduction in our context.

The algorithms presented in Sections 3 and 4 have the general flavor of incremental algorithms [24], while the graph formed by superposing path constraints and used to

recompute invalidated summaries in Section 7 corresponds to the "derivation graph" used in traditional incremental compositional static-analysis algorithms. However, the details of our algorithms are new due to the specific nature of the type of summaries we consider.

The closest related work in the testing space are probably techniques for *regression test selection* (e.g., see [17]) which typically analyze test coverage data and code changes to determine which tests in a given test suite need to be re-executed to cover newly modified code. The techniques we use in Phase 1 of our algorithm are similar, except we do not record coverage data for each pair *lp* and *lq* as discussed at the beginning of Section 3. There is a rich literature on techniques for static and dynamic *change impact analysis* (see [26] for a summary). Our Phase 1 can be seen as a simple instance of these techniques, aimed at validating a given must summary. Although more sophisticated static-analysis techniques (based on dataflow analysis) have been proposed for change impact analysis, we are not aware of any attempt to use verification-condition generation and automated theorem proving techniques like those used in Phase 2 and Phase 3 for precise checking of the impact of a change. The work on *differential symbolic execution* (DSE) [22] is the closest to our Phase 3 algorithm. Unlike DSE, we do not summarize paths in the new program to compare those with summaries of the old program; instead, we want to avoid recomputing new summaries by reusing old ones as much as possible. Whenever an old summary $\langle lp, P, lq, Q\rangle$ becomes invalid and needs to be recomputed, a data-flow-based impact analysis like the one discussed in [23] could refine the procedure described in Section 7 by identifying which specific program paths from *lp* to *lq* need to be re-executed symbolically. In our experiments, every summary covers one or very few paths (of the old program), and this optimization is not likely to help much.

Must abstractions are program abstractions geared towards finding errors, which dualize may abstractions geared towards proving correctness [13]. Reasoning about must abstractions using logic constraint solvers has been proposed before [6, 13, 16, 18, 20], and are related to Phase 3 in our work.

10 Conclusions

In this work, we formulated the problem of statically validating must summaries to make compositional dynamic test generation more incremental. We described three approaches for validating must summaries, that differ in their strengths and weaknesses. We outlined the subtleties involved in using an off-the-shelf verification-condition-based checker for validating must summaries, and the impact of partial predicates on precision. We presented a preliminary evaluation of these approaches on a set of representative intraprocedural summaries generated from real-world applications, and demonstrated the effectiveness of static must summary checking. We plan to evaluate our tool on a larger set of summaries and benchmarks, investigate how to validate interprocedural summaries, and improve the precision of the path-sensitive analysis.

Acknowledgements. We thank the anonymous reviewers for their constructive comments. The work of Cindy Rubio-González was done mostly while visiting Microsoft Research. A preliminary version of this work appeared under the title "Incremental Compositional Dynamic Test Generation" as MSR Technical Report MSR-TR-2010-11, February 2010.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *TACAS'2008*, volume 4963 of *LNCS*, pages 367–381, 2008.
3. M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO '05*, LNCS 4111, pages 364–387, 2005.
4. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05*, pages 82–87, 2005.
5. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.
6. S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A Powerful Approach to Weakest Preconditions. In *PLDI'2009*, 2009.
7. C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV*, pages 449–461, 2005.
8. L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *TACAS '08*, LNCS 4963, pages 337–340, 2008.
9. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
10. A. Edwards, A. Srivastava, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, MSR-TR-2001-50, Microsoft Research, 2001.
11. P. Godefroid. Compositional Dynamic Test Generation. In *POPL'2007*, pages 47–54, 2007.
12. P. Godefroid. Software Model Checking Improving Security of a Billion Computers. In *SPIN'2009*, page 1, 2009.
13. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based Model Checking using Modal Transition Systems. In *CONCUR'2001*, volume 2154 of *LNCS*, pages 426–440, 2001.
14. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'2005*, pages 213–223, 2005.
15. P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS'2008*, pages 151–166, 2008.
16. P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *POPL'2010*, 2010.
17. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, Apr. 2001.
18. A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A Software Model Checker for Verification and refutation. In *CAV'2006*, volume 4144 of *LNCS*, pages 170–174, 2006.
19. S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. In *PLDI'02*, pages 69–82, 2002.
20. J. Hoenicke, K. R. M. Leino, A. Podelski, M. Schaf, and T. Wies. It's doomed; we can prove it. In *FM'2009*, 2009.

21. D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proc. of the 18th Usenix Security Symposium*, 2009.
22. S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.
23. S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *PLDI'2011*, pages 504–515, 2011.
24. G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Algorithms. In *POPL'93*, pages 502–510, 1993.
25. T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL'95*, pages 49–61, 1995.
26. R. A. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *ICST*, pages 429–438, 2010.
27. Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.