Reducing Test Inputs Using Information Partitions[⋆]

Rupak Majumdar and Ru-Gang Xu

Department of Computer Science, University of California, Los Angeles
{rupak,rxu}@cs.ucla.edu

Abstract. Automatic symbolic techniques to generate test inputs, for example, through concolic execution, suffer from *path explosion*: the number of paths to be symbolically solved for grows exponentially with the number of inputs. In many applications though, the inputs can be partitioned into "non-interfering" blocks such that symbolically solving for each input block while keeping all other blocks fixed to concrete values can find the same set of assertion violations as symbolically solving for the entire input. This can greatly reduce the number of paths to be solved (in the best case, from exponentially many to linearly many in the number of inputs). We present an algorithm that combines test input generation by concolic execution with dynamic computation and maintenance of information flow between inputs. Our algorithm iteratively constructs a partition of the inputs, starting with the finest (all inputs separate) and merging blocks if a dependency is detected between variables in distinct input blocks during test generation. Instead of exploring all paths of the program, our algorithm separately explores paths for each block (while fixing variables in other blocks to random values). In the end, the algorithm outputs an input partition and a set of test inputs such that (a) inputs in different blocks do not have any dependencies between them, and (b) the set of tests provides equivalent coverage with respect to finding assertion violations as full concolic execution. We have implemented our algorithm in the Splat test generation tool. We demonstrate that our reduction is effective by generating tests for four examples in packet processing and operating system code.

1 Introduction

Automatic test case generation using symbolic execution and constraint solving has recently regained prominence as a comprehensive technique for generating all paths in a program with a bounded input [4,8,15]. In practice though, these techniques suffer from *path explosion*: the number of paths to be explored can increase exponentially with the number of inputs. This is particularly cumbersome for programs which manipulate data structures such as arrays. While compositional techniques [7,1] alleviate path explosion, full path coverage remains problematic

[⋆] This research was sponsored in part by the NSF grants CCF-0546170, CCF-0702743, and CNS-0720881, and a gift from Intel.

```
void test(int $a_1$, int $a_2$,
          ...,
          int $a_n$) {
1:      if ($a_1 = 0$) $a_1 := 1$;
2:      if ($a_2 = 0$) $a_2 := 1$;
        ...
$n$:      if ($a_n = 0$) $a_n := 1$;

$n+1$: if ($a_1 = 0$) error();
$n+2$: if ($a_2 = 0$) error();
        ...
$2n$:     if ($a_n = 0$) error();
}
```

```
void free(int A[], int count[]) {
  for (int i = 0; i < N; i++) {
    old_count[i] = count[i];
  }
  for (int i = 0; i < N; i++) {
    if (A[i] != 0)
       count[i]++;
  }
  for (int i = 0; i < N; i++) {
    if (A[i] != 0)
       assert(count[i]==old_count[i]+1);
  }
}
```

Fig. 1. Examples with many independent inputs: (a) Example `test` (b) Example `free`

even for medium-sized applications within a reasonable testing budget, say, one night.

In this paper, we develop a technique that exploits the *independence* between different parts of the program input to reduce the number of paths needed to be explored during test generation. As a simple example, consider the function `test` shown in Figure 1(a). While there are 2^n syntactic paths through the code, we quickly recognize that it is sufficient to check only $2 \cdot n$ paths: two each for each of the inputs a_1, a_2, ..., a_n being zero or non-zero. In particular, we conclude that `error()` is not reachable based only on these paths. The additional paths through the program do not add any more "interesting" behaviors, as the inputs are *non-interfering*: there is no data or control dependency between any two distinct inputs a_i, a_j for $i \neq j$. This indicates that by generating tests one independent input at a time (while holding the other inputs to fixed values), we can eliminate the combinational explosion of testing every arrangement of inputs, in this case, from an exponential number of paths (2^n for n inputs) to a linear number ($2 \cdot n$), while retaining all interesting program behaviors (e.g., behaviors that can cause assertion violations or behaviors that lead to an error). While the above example is artificial, there are many interesting examples where the input space can be partitioned into independent and non-interfering components, either through the application semantics (e.g., blocks in a file system, packet headers in network processors, permission-table entries in memory protection schemes) or due to security and privacy reasons (e.g., independent requests to a server).

We present an automatic test generation algorithm FlowTest that formalizes and exploits the independence among inputs. FlowTest is based on *concolic execution* [7,15] which explores executable paths of an input program using simultaneous concrete random simulation and symbolic execution. In contrast to the basic concolic execution algorithm, the main idea of FlowTest is to compute control and data dependencies among variables dynamically while performing concolic execution, and to use these dependencies to keep independent variables separated during test generation. FlowTest maintains a partitioning of the program inputs (where two variables in different blocks are assumed not to interfere), and generates tests by symbolically treating variables in each block in isolation while holding variables in other blocks to fixed values.

In case the partition does denote non-interfering sets of variables, and all program executions terminate, the test generation is relatively sound: any assertion violation detected by basic concolic execution is detected. To check for data or control dependencies between variables in separate blocks, FlowTest maintains a *flow map* during test generation which associates each variable with the set of input blocks in the current partition which can potentially influence the value of the variable. If there is some entry in the flow map which contains more than one input block, this indicates "information flow" between these input blocks. In this case, these input blocks are merged and test generation is repeated by tracking this larger block of inputs together.

The algorithm terminates when the input partitions do not change (and tests have been generated relative to this input partition). For example `test`, starting with the initial optimistic partition in which each variable is in a separate partition, FlowTest will deduce that this partition is non-interfering, and generate test cases that explore the $2n$ interesting paths. In contrast, concolic execution explores 2^n paths.

We have implemented FlowTest on top of the Splat directed testing implementation [18] to test and check information flow in C programs. The benefit of FlowTest is demonstrated on a memory allocator, a memory protection scheme, an intrusion detector module and a packet printer. FlowTest dramatically reduces the number of paths explored for all case studies without increasing much overhead per path due to flow-set generation and dependency checking. In all cases, FlowTest reduced the overall time for input generation and the number of paths generated. In two cases, FlowTest cut input generation in half. In one case, FlowTest terminated in less than ten minutes when the basic concolic execution algorithm failed to terminate even after a day.

Related Work. Test generation using concolic execution has been successfully applied to several large programs [8,15,4,18,3]. However, path explosion has been a fundamental barrier. Several optimizations have been proposed to prune redundant paths, such as function summarization [7,1] and the pruning of paths that have the same side-effects of some previously explored path through read-write sets (RWSets) [2]. FlowTest is an optimization orthogonal to both function summarization and RWSets.

Program slicing has been used to improve the effectiveness of testing and static analysis by removing irrelevant parts of the program [17,16,10,11]. One way to view FlowTest is as simultaneous path exploration by concolic execution and dynamic slicing across test runs: for each input block, the algorithm creates dynamic slices over every run, and merges input blocks that have common data or control dependencies. In contrast to running test generation on statically computed slices, our technique, by computing slices dynamically and on executable traces, can be more precise.

Our optimization based on control and data dependencies is similar to checking information flow [5,6,12]. For example, dynamic information flow checkers [5,12] are based on similar dependency analyzes.

2 Preliminary Definitions

We illustrate our algorithm on a simple imperative language with integer-valued variables while our implementation handles more general features such as pointers and function calls. We represent programs as *control flow graphs (CFG)* $P = (X, X_0, \mathcal{L}, \ell_0, op, E)$ consisting of (1) a set of variables X, with a subset $X_0 \subseteq X$ of *input* variables, (2) a set of control locations (or program counters) $\mathcal{L}$ which include a special start location $\ell_0 \in \mathcal{L}$, (3) a function op labeling each location $\ell \in \mathcal{L}$ with one of the following basic operations:

1. a termination statement halt,
2. an assignment $x := e$, where $x \in X$ and e is an arithmetic expression over X,
3. a conditional $\mathbf{if}(x)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$, where $x \in X$ and ℓ', ℓ'' are locations in $\mathcal{L}$,

and (4) a set of directed edges $E \subseteq \mathcal{L} \times \mathcal{L}$ defined as follows. The set of edges E is the smallest set such that (1) every node ℓ where $op(\ell)$ is an assignment statement has exactly one node ℓ' with $(\ell, \ell') \in E$, and (2) every node ℓ such that $op(\ell)$ is $\mathbf{if}(x)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$ has two edges (ℓ, ℓ') and (ℓ, ℓ'') in E. For a location $\ell \in \mathcal{L}$ where $op(\ell)$ is an assignment operation, we write $N(\ell)$ for its unique neighbor.

Thus, the locations of a CFG correspond to program locations with associated commands, and edges correspond to control flow from one operation to the next. We assume that there is exactly one node ℓ_{halt} in the CFG with $op(\ell_{\mathsf{halt}}) = \mathsf{halt}$. A *path* is a sequence of locations $\ell^1, \ell^2 \ldots \ell^n$ in the CFG. A location $\ell \in \mathcal{L}$ is reachable from $\ell' \in \mathcal{L}$ if there is a path $\ell' \ldots \ell$ in the CFG. We assume that every node in $\mathcal{L}$ is reachable from ℓ_0 and ℓ_{halt} is reachable from every node.

Semantics. The concrete semantics of the program is given using a *memory* that maps variables in X to values. For a memory M, we write $M[x \mapsto v]$ for the memory mapping x to v and every other variable $y \in X \setminus \{x\}$ to $M(y)$. For an expression e, we denote by $M(e)$ the value obtained by evaluating e where each variable x occurring in e is replaced by the value $M(x)$.

Execution starts from a memory M_0 containing initial values for input variables in X_0 and constant default values for variables in $X \setminus X_0$, at the entry location ℓ_0. Each operation updates the memory and the control location. Suppose the current location is ℓ and the current memory is M. If $op(\ell)$ is $x := e$, then the new location is $N(\ell)$ and the new memory is $M[x \mapsto M(e)]$. If $op(\ell)$ is $\mathbf{if}(x)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$ and $M(x) = 0$, then the new location is ℓ'' and the new memory is again M. On the other hand, if $M(x) \neq 0$ then the new location is ℓ' and the new memory remains M. If $op(\ell)$ is halt, the program terminates.

Execution of the program starting from a memory M_0 defines a path in the CFG in a natural way. A path is executable if it is the path corresponding to program execution from some initial memory M_0.

Symbolic and Concolic Execution. We shall also evaluate programs *symbolically*. Symbolic execution is performed using a *symbolic memory* μ, which maps variables in X to symbolic expressions over a set of symbolic constants, and a *path constraint* ξ, which collects predicates over symbolic constants along the execution path. Execution proceeds as in the concrete case, starting at ℓ_0 with an initial symbolic memory μ which maps each variable x in X_0 to a fresh symbolic constant α_x and each variable $y \in X \setminus X_0$ to some default constant value, and the path constraint *true*. For an assignment $x := e$, the symbolic memory μ is updated to $\mu[x \mapsto \mu(e)]$, where $\mu(e)$ denotes the symbolic expression obtained by evaluating e using μ and $\mu[x \mapsto v]$ denotes the symbolic memory that updates μ by setting x to v. The control location is updated to $N(\ell)$. For a conditional $\mathbf{if}(x)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$, there is a choice in updating the control location. If the new control location is chosen to be ℓ', the path constraint is updated to $\xi \wedge \mu(x) \neq 0$, and if the new control location is chosen to be ℓ'', the path constraint is updated to $\xi \wedge \mu(x) = 0$. In each case, the new symbolic memory is still μ. Symbolic execution terminates at halt.

For each execution path, every satisfying assignment to the path constraint ξ gives values to the input variables in X_0 that guarantee the concrete execution proceeds along this path. *Concolic execution* [8,15] is a variant on symbolic execution in which the program is run simultaneously with concrete and symbolic values.

Partitions. A *partition* $\Pi(X)$ of a set X is a set of pairwise disjoint subsets of X such that $X = \bigcup_{Y \in \Pi(X)} Y$. We call each subset in a partition a *block* of the partition. For a variable $x \in X$, we denote by $\Pi(X)[x]$ the block of $\Pi(X)$ that contains x.

Given a partition $\Pi(X)$ and a subset $Y \subseteq \Pi(X)$ of blocks in $\Pi(X)$, the partition $\texttt{Merge}(\Pi(X), Y)$ obtained by merging blocks in Y is defined as $(\Pi(X) \setminus Y) \cup \{\cup_{b \in Y} b\}$.

A partition $\Pi(X)$ *refines* a partition $\Pi'(X)$ if every block in $\Pi'(X)$ is a union of blocks in $\Pi(X)$. In this case we say $\Pi'(X)$ is *as coarse as* $\Pi(X)$. If $\Pi'(X)$ is as coarse as $\Pi(X)$ but $\Pi'(X) \neq \Pi(X)$, we say $\Pi'(X)$ is coarser than $\Pi(X)$. When the set X is clear from the context, we simply write Π.

Control and Data Dependence. For two locations $\ell, \ell' \in \mathcal{L}$ we say ℓ' *post-dominates* ℓ if every path from ℓ to ℓ_{halt} contains ℓ'. We say ℓ' is the *immediate* post-dominator of ℓ, written $\ell' = idom(\ell)$, if (1) $\ell' \neq \ell$, (2) ℓ' post-dominates ℓ, and (3) every ℓ'' that post-dominates ℓ is also a post-dominator of ℓ'. It is known that every location has a unique immediate post-dominator [13], and hence the function $idom(\ell)$ is well-defined for every $\ell \neq \ell_{\mathsf{halt}}$.

A node ℓ is *control dependent* on ℓ' if there exists some executable path $\ell_0 \ldots \ell' \ldots \ell$ such that $idom(\ell')$ does not appear between ℓ' and ℓ in the path.

Algorithm 1. FlowTest

```
   Input: Program $P$, initial partition $\Pi_0(X)$
1  local partitions $\Pi$ and $\Pi_{old}$ of $X$ ;
2  local flow map $flow$;
3  $\Pi := \Pi_0(X)$ ;
4  $\Pi_{old} := \emptyset$ ;
5  $flow(x) := \{\Pi[x]\}$ for $x \in X$;
6  while $\Pi_{old} \neq \Pi$ do
7      $\Pi_{old} := \Pi$;
8      for $I \in \Pi_{old}$ do
9          $input := \lambda x \in X_0.random()$;
10         $flow := \mathsf{Generate}(P, \Pi, I, flow, input, -1)$;
11     end
12     for each $x \in X$ do
13         $\Pi := \mathtt{Merge}(\Pi, flow(x))$;
14     end
15 end
```

For an expression e, we write $\mathsf{Use}(e)$ for the set of variables in X occurring in e. For variables $x, y \in X$, we say x is *data dependent* on y if there is some executable path to a location ℓ such that $op(\ell)$ is $x := e$ and $y \in \mathsf{Use}(e)$.

3 The FlowTest Algorithm

Overall Algorithm. Algorithm 1 shows the overall FlowTest algorithm. It takes as input a program P and an initial partition $\Pi_0(X)$ of the set X of variables, and applies test generation with iterative merging of partitions. It maintains a "current" partition Π of the inputs, which is updated based on control and data dependence information accrued by test generation. The "old" partition Π_{old} is used to check when a round of test generation does not change the partition. Initially, Π is the input partition $\Pi_0(X)$, and Π_{old} is the empty-set (lines 3, 4).

The main data structure to store dependency information is called a *flow map*, $flow : X \to 2^{\Pi}$, a function mapping each variable $x \in X$ to a set of blocks of the current partition Π. Intuitively, $flow(x)$ denotes the set of input blocks that are known to influence (through data or control dependence) the value of x. Initially, we set $flow(x) = \{\Pi[x]\}$ for each $x \in X$ (line 5).

The main loop (lines 6–14) implements test generation and iterative merging of partitions. The procedure Generate (described next) implements a standard path exploration algorithm using concolic execution, but additionally updates the flow map. Generate is called to generate tests for each block I of the current partition Π (lines 8–11). In each call, the variables in the block I are treated symbolically, and every other variable is given a fixed, random initial value. Generate returns an updated flow map which is used to merge blocks in Π to get an updated partition (lines 12–14). For every $x \in X$ such that $|flow(x)| > 1$, the blocks in $flow(x)$ are merged into one block to get a new coarser partition. The main loop is repeated with this coarser partition until there is no change.

Algorithm 2. Generate

```
   Input: Program $P$, partition $\Pi$, block $I \in \Pi$, flow map $flow$
   Input: input $input$, last explored branch $last$
1  $(\xi, flow)$ := Execute($P, \Pi, I, flow, input$);
2  $index$ := Length($\xi$) $- 1$;
3  while not empty($\xi$) $\wedge$ $index \neq last$ do
4      $p$ := pop($\xi$);
5      if $\xi \wedge \neg p$ is satisfiable then
6          $input$ := Solve($\xi, \neg p$);
7          $flow$ := Generate($P, \Pi, I, flow, input, index$);
8      $index$ := $index - 1$;
9  end
10 return $flow$;
```

Algorithm Generate**.** Algorithm 2 describes the path enumeration algorithm, and is similar to the path enumeration schemes in [8,15,4]. It takes as input the program P, a partition Π of the inputs of P, a block I of the partition, an input *input* mapping input variables to initial values, and an index *last* tracking the last visited branch. It performs test case generation using a depth first traversal of the program paths using concolic execution. In the concolic execution, only inputs from I are treated symbolically and the rest of the inputs are set to fixed concrete values (chosen randomly). The procedure returns an updated flow map.

The main loop of Algorithm Generate implements a recursive traversal of program paths. In each call to Generate, the function Execute (described next) is used to perform concolic execution along a single path and update the flow map. The returned path constraint ξ is used to generate a sequence of new inputs in the loop (lines 3–9). This is done by popping and negating the last constraint in the path constraint and generating a new input using procedure Solve. The new input causes the program to explore a new path: one that differs from the last one in the direction of the last conditional branch. The function Generate is recursively invoked to explore paths using this new input (line 7).

Notice that the pure concolic execution algorithm, from e.g., [8], is captured by the call Generate($P, \{X\}, X, \lambda x.\{X\}, \lambda x.0, -1$).

Algorithm Execute**.** Algorithm 3 describes the procedure for performing concolic execution and computing data and control dependencies along a single program path. It takes as input the program P, a concrete input i, and a partition Π. It returns a path constraint ξ and an updated flow map *flow*. Notice that the path constraint is maintained as a stack of predicates (instead of as a conjunction of predicates). This helps in simplifying the backtracking search in Generate.

Algorithm Execute, ignoring lines 10, 11, 16, 17, 20 and 21, is identical to the concolic execution algorithm [8,15]. It executes the program using both the concrete memory M and the symbolic memory μ. The extra lines update the flow map.

Algorithm 3. Execute

```
   Input: Program $P$, partition $\Pi$, block $I \in \Pi$, flow map $\mathit{flow}$, input $i$
   Result: Path constraint $\xi$ and updated flow map $\mathit{flow}$
 1 for $x \in X$ do
 2     $M(x) := i(x)$; if $x \in I$ then $\mu(x) := \alpha_x$;
 3 end
 4 $\xi :=$ emptyStack; $\ell := \ell_0$;
 5 $\mathsf{Ctrl}(\ell_0) := \emptyset$;
 6 while $op(\ell) \neq \mathsf{halt}$ do
 7     switch $op(\ell)$ do
 8         case $l := e$
 9             $M := M[l \mapsto M(e)]$; $\mu := \mu[l \mapsto \mu(e)]$; $\ell := N(\ell)$;
10             $\mathsf{Ctrl}(N(\ell)) := \mathsf{Ctrl}(\ell) \setminus \mathsf{RmCtrl}(\mathsf{Ctrl}(\ell), N(\ell))$;
11             $\mathit{flow}(l) := \mathit{flow}(l) \cup \bigcup_{x \in \mathsf{Use}(e)} \mathit{flow}(x) \cup \bigcup_{\langle \ell', x' \rangle \in \mathsf{Ctrl}(N(\ell))} \mathit{flow}(x')$;
12         end
13         case if$(x)$then $\ell'$ else $\ell''$
14             if $M(x) = 0$ then
15                 $\xi :=$ push$(\mu(x) = 0, \xi)$; $\ell := \ell''$;
16                 $\mathsf{Ctrl}(\ell'') := (\mathsf{Ctrl}(\ell) \cup \{\langle \ell, x \rangle\}) \setminus \mathsf{RmCtrl}(\mathsf{Ctrl}(\ell) \cup \{\langle \ell, x \rangle\}, \ell'')$;
17                 $\mathit{flow}(x) := \mathit{flow}(x) \cup \bigcup_{\langle \hat{\ell}, y \rangle \in \mathsf{Ctrl}(\ell'')} \mathit{flow}(y)$;
18             else
19                 $\xi :=$ push$(\mu(x) \neq 0, \xi)$; $\ell := \ell'$;
20                 $\mathsf{Ctrl}(\ell') := (\mathsf{Ctrl}(\ell) \cup \{\langle \ell, x \rangle\}) \setminus \mathsf{RmCtrl}(\mathsf{Ctrl}(\ell) \cup \{\langle \ell, x \rangle\}, \ell')$;
21                 $\mathit{flow}(x) := \mathit{flow}(x) \cup \bigcup_{\langle \hat{\ell}, y \rangle \in \mathsf{Ctrl}(\ell')} \mathit{flow}(y)$;
22
23         end
24     end
25 end
26 return$(\xi, \mathit{flow})$;
```

We now describe the working of Algorithm Execute. The concrete memory is initialized with the concrete input i, and the symbolic memory is initialized with a fresh symbolic constant α_x for each $x \in I$ (lines 1–3). The path constraint is initialized to the empty stack and the initial control location is ℓ_0 (line 4).

The main loop of Execute (lines 6–25) performs concrete and symbolic evaluation of the program while updating the flow map. The loop executes while the program has not terminated (or, in practice, until some resource bound such as the number of steps has been exhausted).

We first ignore the update of the flow map and describe how the concrete and symbolic memories and the path constraint are updated in each iteration.

If the current location is ℓ and the current operation is an assignment $l := e$ (lines 8–12), the concrete memory updates the value of l to $M(e)$ and the symbolic memory updates it to $\mu(e)$ (line 9). Finally, the control location is updated to be $N(\ell)$.

If the current location is ℓ and the current operation is **if**(x) **then** ℓ' **else** ℓ'', the updates are performed as follows (lines 13–23). The concrete memory M

and symbolic memory μ remain unchanged. If $M(x) \neq 0$, then the constraint $\mu(x) \neq 0$ is pushed on to the path constraint stack ξ, and the new control location is ℓ' (line 19). If $M(x) = 0$, then the constraint $\mu(x) = 0$ is pushed on to the path constraint stack ξ, and the new control location is ℓ'' (line 15).

We now describe how the control dependencies and the flow maps are updated. We use a helper data structure Ctrl mapping each location to a set of pairs of locations and expressions. This data structure is used to maintain the set of conditionals on which a location is control dependent along the current execution. At the initial location ℓ_0, we set $\mathsf{Ctrl}(\ell_0) = \emptyset$ (line 5). Each statement updates the set Ctrl. We use the following definition. Let $L \subseteq \mathcal{L} \times X$ be a set of pairs of locations and variables. Let $\ell \in \mathcal{L}$. We define $\mathsf{RmCtrl}(L, \ell) = \{\langle \ell', x \rangle \in L \mid \ell \text{ is the immediate post-dominator of } \ell'\}$. Intuitively, these are the set of conditionals that on which ℓ is no longer control dependent.

Suppose the current location is ℓ and $op(\ell)$ is the assignment $l := e$. The set $\mathsf{Ctrl}(N(\ell))$ consists of the set $\mathsf{Ctrl}(\ell)$ minus the set of all locations which are immediate post-dominated by $N(\ell)$ (line 10). The flow map for the variable l is updated as follows (see line 11). There are three components in the update. The first component is $\mathit{flow}(l)$, the flow computed so far. The second component $\bigcup_{x \in \mathsf{Use}(e)} \mathit{flow}(x)$ captures data dependencies on l due to the assignment $l := e$: for each variable x appearing in e, it adds every input block known to influence x (the set $\mathit{flow}(x)$) to $\mathit{flow}(l)$. The third component captures dependencies from controlling conditionals. The controlling conditionals for $N(\ell)$ and their conditional variables are stored in $\mathsf{Ctrl}(N(\ell))$. For every $\langle \ell', x' \rangle \in \mathsf{Ctrl}(N(\ell))$, we add the set $\mathit{flow}(x')$ of inputs known to influence x' to $\mathit{flow}(l)$.

Now suppose the current location is ℓ and $op(\ell)$ is $\mathbf{if}(x)$ $\mathbf{then}$ ℓ' $\mathbf{else}$ ℓ''. In this case, depending on the evaluation of the conditional x, the execution goes to ℓ' or ℓ'' and the corresponding data structure $\mathsf{Ctrl}(\ell')$ or $\mathsf{Ctrl}(\ell'')$ is updated to reflect dependence on the conditional x (lines 16, 20). The pair $\langle \ell, x \rangle$ is added to the set of controllers to indicate that the conditional may control execution to ℓ' and ℓ'', and as before, the set of conditionals post-dominated by ℓ' (respectively, ℓ'') are removed. Finally, for each $\langle \hat{\ell}, y \rangle$ in $\mathsf{Ctrl}(\ell')$ (respectively, $\mathsf{Ctrl}(\ell'')$), the set of input blocks $\mathit{flow}(y)$ is added to the flow map for x.

The updated flow map is returned at the end of the loop.

Algorithm solve. Finally, procedure solve takes as input a stack of constraints ξ and a predicate p, and returns a satisfying assignment of the formula

$$\bigwedge_{\phi \in \xi} \phi \wedge p$$

using a decision procedure for the constraint language. In the following, we assume that the decision procedure is *complete*: it always finds a satisfying assignment if the formula is satisfiable.

Relative Soundness. As we have seen, FlowTest can end up exploring many fewer paths than pure concolic execution (i.e., the call Generate $(P, \{X\}, X, \lambda x.\{X\}, \lambda x.0, -1)$). However, under the assumption that all program executions terminate, we can show that FlowTest is relatively sound: for every location ℓ, if concolic execution finds a feasible path to ℓ, then so does FlowTest. In particular, if all program executions terminate, then FlowTest does not miss any assertion violation detected by concolic execution.

We say a location $\hat{\ell}$ is *reachable* in $\mathsf{FlowTest}(P, \Pi_0)$ or Generate $(P, \{X\}, X, \lambda x.\{X\}, \lambda x.0, -1)$ if the execution reaches line 7 of Execute with $\ell = \hat{\ell}$. Clearly, a location reachable in either algorithm is reachable in the CFG by an executable path.

Theorem 1. *Let* $P = (X, X_0, \mathcal{L}, \ell_0, op, E)$ *be a program and* Π_0 *an initial partition of the inputs of* P*. Assume* P *terminates on all inputs. If* $\mathsf{FlowTest}(P, \Pi_0)$ *terminates then every location* $\ell \in \mathcal{L}$ *reachable in* Generate $(P, \{X\}, X, \lambda x.\{X\}, \lambda x.0, -1)$ *is also reachable in* $\mathsf{FlowTest}(P, \Pi_0)$*.*

We sketch a proof of the theorem. We prove the theorem by contradiction. Suppose that there is a location ℓ that is reachable in concolic execution but not in FlowTest. Fix a path $\pi = \ell_0 \to \ell_1 \to \ldots \to \ell_k$ executed by concolic execution such that ℓ_k is not reachable in FlowTest but each location $\ell_0, \ldots, \ell_{k-1}$ is reachable by FlowTest. (If there are several such paths, choose one arbitrarily.) Notice that since π is executed by concolic execution, the path constraints resulting from executing π is satisfiable. Also, $op(\ell_{k-1})$ must be a conditional, since if it were an assignment and ℓ_{k-1} is reachable in FlowTest, ℓ_k would also be reachable in FlowTest.

Since every program execution terminates, we can construct a *path slice* [9] of π, i.e., a subsequence π' of operations of π, with the following properties: (1) every initial memory M_0 that can execute π can also execute π', and (2) every initial memory M_0 that can execute π' is such that there is a (possibly different) program path π'' from ℓ_0 to ℓ_k such that M_0 can execute π''. Such a slice can be computed using the algorithm from [9]. Since the path constraint for π is satisfiable, so is the path constraint for π'. Let $\mathsf{V}(\pi')$ be the set of variables appearing in π', i.e., $\mathsf{V}(\pi')$ is the smallest set such that for every operation $l := e$ in π' we have $\{l\} \cup \mathsf{Use}(e) \subseteq \mathsf{V}(\pi')$ and for every operation $\textbf{if}(x)\ \textbf{then}\ \ell'\ \textbf{else}\ \ell''$ in π', we have $x \in \mathsf{V}(\pi')$.

We show that each variable in $\mathsf{V}(\pi')$ is eventually merged into the same input block. We note that every conditional operation in π' controls the next operation in π' and every assignment operation $l := e$ in π' either uses l in a subsequent assignment or in a conditional. Now, since every location along π' is reachable, we can show (by induction on the length of π') that all variables in $\mathsf{V}(\pi')$ are merged into the same input block. Call this block I.

Consider the call to Generate made by FlowTest with the input block I. This call is made by FlowTest in the iteration after I is created. Since π' is a path slice of π, and π is executable, we know that the sequence of operations in π' can be executed by suitably setting values of variables in I, no matter how the rest of the variables are set. Moreover, since the path constraint for π is satisfiable, so is the path constraint for π'. Since every execution terminates, the backtracking search implemented in Generate is complete (modulo completentess of the decision procedure), so the call to Generate using input block I must eventually hit ℓ_k. This is a contradiction, since this shows ℓ_k is reachable in FlowTest.

While the theorem makes the strong assumption that all program paths terminate, in practice this is ensured by setting a limit on the length of paths simulated in concolic execution. In fact, if the program has an infinite execution, then the concolic execution algorithm does not terminate: it either finds inputs that show non-termination (and gets stuck in Execute), or finds an infinite sequence of longer and longer program execution paths.

4 Example

We illustrate the working of FlowTest on a simplified version of a virtual memory page-free routine in an OS kernel (the actual code was tested in our experiments). Figure 1(b) shows the `free` function which takes as input two arrays of integers `A` and `count` each of size N, a fixed constant. For this example, let us set $N = 2$. For readability, we use C-like syntax and an array notation as a shorthand for declaring and using several variables (our simple language does not have arrays, but our implementations deals with arrays).

Notice that the loop in lines 4–7 of the function `free` in has 2^N paths, because the conditional on line 5 could be true or false for $0 \leq i < N$. So, even for small N, concolic execution becomes infeasible. For example, our implementation of concolic execution in the tool Splat [18] already takes one day on an Intel Core 2 Duo 2.33 Ghz machine when $N = 20$.

Now we show how FlowTest can reduce the cost of testing this function. Our intuition is that the behavior of one "page" in the system is independent of all other pages, so we should test the code one page at a time. Concretely, in the code, there is no control or data dependency between two separate indices of the arrays. For example, the value $A[0]$ is not control or data dependent on $A[1]$, and $count[0]$ is not control or data dependent on $count[1]$. However, there is dependence between $A[i]$ and $count[i]$ (through the conditional in lines 5–6).

Initially, we start with the optimistic partition

$$\{\ \{A[0]\},\ \{A[1]\},\ \{count[0]\},\ \{count[1]\}\}$$

in which each variable is in its own partition. Consider the run of Generate when $A[0]$ is treated symbolically, but all other variables are fixed to constant values. The concolic execution generates two executions: one in which $A[0]$ is 0 and a second in which $A[0]$ is not zero. For the run in which $A[0]$ is not zero, moreover, the flow map is updated with the entry:

$$\mathit{flow}(count[0]) = \{\{A[0]\}, \{count[0]\}\}$$

since the assignment to $count[0]$ is control dependent on the predicate $A[0] \neq 0$. Thus, the blocks $\{A[0]\}$ and $\{count[0]\}$ are merged. In a similar way, the blocks $\{A[1]\}$ and $\{count[1]\}$ are merged.

In the next iteration of the main loop of FlowTest, the function Generate generates tests individually for the blocks $\{A[0], count[0]\}$ and $\{A[1], count[1]\}$. This time, there is no additional merging of input blocks. Hence, the algorithm terminates. The assertion holds on all paths of the resulting test set. The relative soundness result implies that the assertions hold for every execution of `free`.

5 Evaluation

5.1 Implementation

We have implemented a subset of the FlowTest algorithm to generate inputs for C programs on top of the Splat concolic execution tool [18]. In our tool, we take as input a program and a manual partition of the inputs, and implement the test generation and dependence checking part of the algorithm (lines 8–11 in Algorithm 1). However, instead of merging input blocks and iterating, our tool stops if two different input blocks must be merged (and assumes that the merging is performed manually). Our experiments (described below) confirm the following hypotheses:

– The performance impact of the added checks for dependencies is small, and more than compensated by the reduction in the number of paths explored.
– A tester can find non-interfering input partitions by superficially examining the source code.

Our implementation is divided in four main components: control flow generation, tracking flow, symbolic execution, and test generation. We use CIL [14] to instrument C code and to statically build the post-dominators (assuming every loop terminates) using the standard backwards dataflow algorithm [13]. We use the concolic execution and test generation loop of Splat to additionally track information flow between input partitions. Our implementation handle function calls and dynamic memory allocation in the standard way [8,15]. In our experiments, we did not put a priori bounds on the lengths of executions, but ran each function to completion.

Functions. Statement labels are made to be unique across all functions, therefore, entering and exiting a function during an execution is similar to in-lining the function at each call site.

Pointers and Memory. Given a branch label ℓ, we track all writes from that branch label to the label that post-dominates ℓ. Because our algorithm executes the code, address resolution is done during runtime. This implies we do not require a static alias analysis or suffer from the imprecision associated with static alias analysis, especially in the presence of address arithmetic or complex data structures.

We merge all dynamically allocated memory by allocation site when computing data dependencies, but distinguish each memory location individually when performing concolic execution. This can merge different potentially independent locations into the same block, but does not introduce imprecision in concolic execution. For statically- and stack-allocated variables, each byte is distinguished individually. For multiple writes to the same address from the branch label ℓ to the immediate post-dominator of ℓ, we take the union of all blocks that flow into that address. This limits the memory tracked to be (stack size $\times$ static allocation points) per branch node and is conservative because writes to one allocation point flow into all dynamic memory allocated from that point. Experientially, we found that this trade off did not lead to false alarms and was sufficient to check that blocks did not interfere with each other – mostly because each field in a data structure is distinguished.

Performance. Experimentally, we found that the extra information stored and checked along each path results in doubling the time taken to generate and explore a path. However, this increase in the per path time was more than compensated by the reduction in the number of paths explored. The static analysis cost (to compute post-dominators) is negligible ($<$ 1 second for each of our case studies). While we did not experiment with automatic merging of input blocks, we expect there would not be a significant performance hit because finer grain blocks would have few paths and merging would occur quickly.

5.2 Case Studies

We have compared our algorithm with Splat on the following case studies. `pmap` is the code managing creation and freeing of a new processes in a virtual memory system for an educational OS kernel (1.6 KLOC), `mondrian` is the insertion module in a memory protection scheme (2.6 KLOC), `snort` is a module in an intrusion detection system (1.7 KLOC), and `tcpdump` consists of eight printers in a packet sniffer (12 KLOC).

In all programs selected, it was possible to come up with a good partition of the input with only a cursory examination of the source code (even though we did not write the code ourselves). In `pmap`, each header representing a physical page is a separate block. In `mondrian`, the protection bits could be partitioned from all other inputs. In both of the packet processing programs `snort` and `tcpdump`, various parts of the packet header could be partitioned. For `tcpdump`, we looked at 10 different packet types that could be printed by `tcpdump`. The partition for all those packet types were destination address, source address and everything else.

The implementation automatically generated inputs to check if there was interference between the manually chosen partitions for each program. Table 1 shows the results. All experiments were performed on a 2.33 GHz Intel Core 2 Duo with 2 GB of RAM. The two columns compare partitioning with concolic execution. The **Input Size** is the size of the symbolic input buffer used for both implementations. **Blocks** is the number of input blocks found manually and used for FlowTest. **Paths** is the number of unique paths found. **Coverage** is the number of branches covered. For all experiments, we could not achieve 100% coverage because we were only testing a specific part of the program. While these partitions may not be the best partition, FlowTest finished in half the time

Table 1. Experimental Results: Comparing FlowTest and Splat. **Input** is the size of the symbolic input buffer in bytes. **Blocks** is the number of blocks into which the input was (manually) partitioned. **Coverage** is branch coverage for both FlowTest and Splat. **Paths** is the number of unique paths explored. **Time** is the time taken up to a maximum of one hour.

				FlowTest		Splat	
Program	**Input**	**Blocks**	**Coverage**	**Paths**	**Time**	**Paths**	**Time**
`pmap`	1024	512	50/120 = 42%	1536	8m29s	6238	>1hr
`mondrian`	12	2	57/192 = 30%	733	23m20s	2916	36m35s
`snort`	128	2	44/52 = 85%	45	4m44s	73	4m55s
`tcpdump`	128	3	404/2264 = 18%	1247	34m1s	4974	53m46s

of Splat in two cases. For `pmap`, FlowTest finished in less than 10 minutes. In contrast, Splat did not finish in one day (Table 1 shows the progress made by Splat after one hour (6238 paths).

We did not compare our algorithm with combinations of other optimizations such as function summaries [7,1] or RWSets [2], as these techniques are orthogonal and can be combined to give better performance.

Limitations. In several examples, we found artificial dependencies between inputs caused by error handling code. For example, the `snort` module first checks if a packet is an UDP packet, and returns if not:

```
if(!PacketIsUDP(p)) return;
```

This introduces control dependencies between every field checked in `PacketIsUDP` and the rest of the code. However, these fields can be separated into their own blocks if the `PacketIsUDP(p)` predicate holds. A second limitation is our requiring that inputs are *partitioned*. In many programs, there is a small set of inputs that have dependencies with all other inputs, but the rest of the inputs are independent. Thus, it makes sense to divide the inputs into subsets whose intersections may not be empty.

Acknowledgments. We thank the anonymous referees for their detailed suggestions.

References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
2. Boonstoppel, P., Cadar, C., Engler, D.: RWset: Attacking path explosion in constraint-based test generation. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 351–366. Springer, Heidelberg (2008)
3. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI (2008)

4. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: Exe: automatically generating inputs of death. In: CCS (2006)
5. Clause, J., Li, W., Orso, A.: Dytan: A generic dynamic taint analysis framework. In: ISSTA (2007)
6. Denning, D.E.: A lattice model of secure information flow. Commun. ACM 19(5), 236–243 (1976)
7. Godefroid, P.: Compositional dynamic test generation. In: POPL. ACM, New York (2007)
8. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI (2005)
9. Jhala, R., Majumdar, R.: Path slicing. In: PLDI 2005, ACM, New York (2005)
10. Korel, B., Laski, J.: Dynamic program slicing. Information Processing Letters 29, 155–163 (1988)
11. Korel, B., Yalamanchili, S.: Forward computation of dynamic program slices. In: ISSTA (1994)
12. Masri, W., Podgurski, A., Leon, D.: Detecting and debugging insecure information flows. In: ISSRE (2004)
13. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
14. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, p. 213. Springer, Heidelberg (2002)
15. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: FSE (2005)
16. Tip, F.: A survey of program slicing techniques. Journel of Programming Languages 3, 121–189 (1995)
17. Weiser, M.: Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D Thesis (1979)
18. Xu, R., Godefroid, P., Majumdar, R.: Testing for buffer overflows with length abstraction. In: ISSTA (2008)