Automatic Partial Loop Summarization in Dynamic Test Generation

Patrice Godefroid
Microsoft Research
Redmond, WA, USA
pg@microsoft.com

Daniel Luchaup*
University of Wisconsin Madison
Madison, WI, USA
luchaup@cs.wisc.edu

ABSTRACT

Whitebox fuzzing extends dynamic test generation based on symbolic execution and constraint solving from unit testing to whole-application security testing. Unfortunately, input-dependent loops may cause an explosion in the number of constraints to be solved and in the number of execution paths to be explored. In practice, whitebox fuzzers arbitrarily bound the number of constraints and paths due to input-dependent loops, hence at the risk of missing code and bugs.

In this work, we investigate the use of simple loop-guard pattern-matching rules to automatically guess an input constraint defining the number of iterations of input-dependent loops during dynamic symbolic execution. We discover the loop structure of the program on the fly, detect *induction variables*, which are variables modified by a constant value during loop iterations, and infer simple partial loop invariants relating the value of such variables. Whenever a guess is confirmed later during the current dynamic symbolic execution, we then inject new constraints representing pre and post loop conditions, effectively summarizing the symbolic execution of that loop. These pre and post conditions are derived from the partial loop invariants synthesized dynamically using pattern-matching rules on the loop guards and induction variables, without requiring any static analysis, theorem proving, or input-format specification. This technique has been implemented in the whitebox fuzzer SAGE, scales to large programs with many nested loops, and we present results of experiments with a Windows 7 image parser.

1. INTRODUCTION

Dynamic test generation [11, 6] consists of running a program while simultaneously executing the program symbolically in order to gather constrains 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 few years, whitebox fuzzing [12] 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 [12] and Linux [14] 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 [10]. 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 SMT solver, according to the authors of the Z3 SMT solver [7].

*The work of this author was done mostly while visiting Microsoft.


```
void main(int x) { // x is an input
  int c = 0, p = 0;
  while (1) {
    if (x<=0) break;
    if (c == 50) abort1(); /* error1 */
    c = c + 1;
    p = p + c;
    x = x - 1;
  }
  if (c == 30) abort2();  /* error2 */
}
```

Figure 1: A simple program with an input-dependent loop.

Unfortunately, the number of paths to be explored can be astronomical. For instance, the presence of a single loop whose number of iterations depends on some unbounded input makes the number of feasible program paths infinite. Such a pathological case is illustrated in the small program example shown in Figure 1, where the number of iterations of the `while` loop depends on the input value x_0 stored in variable `x` at the beginning of the program execution.

Suppose we start testing this program with an initial input value $x_0 = 10$ for `x`. The first path constraint generated will be $(x_0 > 0) \wedge (x_0 - 1 > 0) \wedge \cdots \wedge (x_0 - 9 > 0) \wedge (x_0 - 10 <= 0)$. Indeed, the path constraint is defined as a conjunction of tests on inputs that the program executes along the execution. In other words, symbolic execution in dynamic test generation [11, 6] only tracks direct data dependencies on program inputs, not indirect dependencies. Negating each of the constraints in the path constraint one by one will generate new tests (10 in this example), that will exercise new whole-program paths. This process can be repeated, in principle possibly forever if x_0 can be any unbounded integer. In practice, tools like SAGE include counters to bound the number of constraints that can be generated from a particular program branch. This heuristics effectively prunes the search space in an unsound manner, i.e., may fail to exercise code and miss bugs.

In this paper, we investigate an alternative approach based on automatic (partial) loop-invariant generation which, when applicable, can (partially) summarize a loop body during a single dynamic symbolic execution. This allows reasoning about *many* loop unfoldings in one shot, by treating all of these as members of a single symbolic equivalence class of executions. Intuitively, in the example of Figure 1, the key to generate inputs to exercise the two aborts, hence to prove their reachability, is to relate the value of program variable `c` with the input x_0. In this case, this relationship is indirect and not captured by symbolic execution. However, it is captured by the loop invariant $c + x = x_0$ holding in lines 3 to 5, where c and x denote the current value of variables `c` and `x`, respectively. When the loop terminates and executes line 10, we have $x = 0$ and the loop invariant can be simplified to obtain the (partial) loop postcondition $c = x_0$. In both cases, thanks to the loop invariant, we obtain a *symbolic* expression for c which relates its value to the input value x_0.

Given this loop invariant, it is then possible to *summarize* [9] the loop body with a logic formula of the form $pre_{loop} \wedge post_{loop}$, where pre_{loop} is a loop precondition defining a set of executions covered by the summary, and $post_{loop}$ is a loop postcondition capturing (perhaps only partially) side-effects occurring during those executions. For our example, a loop summary can be defined by the precondition $pre_{loop} = (x_0 > 0)$, characterizing all program executions where the loop body is executed at least once, and the (partial) postcondition $post_{loop} = (x = 0) \wedge (c = x_0)$, which captures only partially side-effects resulting from such executions, namely the effects on variables x and c but not p. Such a summary thus *generalizes* the specific current execution (where $x_0 = 10$) to a set of executions (where $x_0 > 0$), and is akin to loop-acceleration techniques used in infinite-state model checking.

We present in this paper an algorithm for automatically generating such loop summaries *dynamically*, during (a single) dynamic symbolic execution. This algorithm detects the loop structure of the current program execution on the fly, as well as input-dependent loops. It also tracks *induction variables*, which are variables modified by a constant value during each loop iteration. Our algorithm includes a (partial) loop-invariant generator that uses pattern-matching rules on the loop guards to guess the number of loop iterations, and can infer loop invariants relating values of induction variables, a restricted but common class of loop invariants. This algorithm does *not* require any user annotations, input-format specifications, theorem proving, or static analysis. It is also applicable to any program, including nested loops, loops with multiple guards, and arbitrary control-flow graphs with unstructured loops and gotos.

When the loop is about to start executing its last iteration during the current dynamic symbolic execution, our algorithm updates the current path constraint and symbolic execution state with new constraints representing pre and post loop conditions that (partially) summarizes and generalizes the symbolic execution of that loop. Program variables appearing in post-conditions are associated with symbolic values relating them to input values. This way, subsequent tests on those program variables, either inside the body of the loop (as in line 5) or after the loop terminates (as in line 10 in Figure 1), are added to the path constraint.

In our example, when dynamic symbolic execution with $x_0 = 10$ starts its last (10th) iteration, the loop is summarized by $pre_{loop} = (x_0 > 0)$ and $post_{loop} = (x = 1) \wedge (c + x = x_0)$, the path constraint is updated to become $(x_0 > 0)$, and variable `c` is associated with the new symbolic value $x_0 - 1$ (since $c = x_0 - x$ and $x = 1$). Then, when the conditional statement on line 5 is executed, the constraint $(x_0 - 1) \neq 50$ is added to the path constraint. After negating this last constraint, a solution to the new path constraint $(x_0 > 0) \wedge (x_0 - 1 = 50)$ is $x_0 = 51$, which leads to a new test hitting the abort1 statement on line 5. Later, when dynamic symbolic execution exits the loop and reaches line 10, the path constraint is now $(x_0 > 0) \wedge ((x_0 - 1) \neq 50)$, and variable `c` is now associated with the symbolic value $(x_0 - 1) + 1$ (because of the symbolic execution of line 6 in the 10th iteration), that is, x_0 after simplification. Then, when the conditional statement on line 10 is executed, the constraint $x_0 \neq 30$ is added to the path constraint. After negating this last constraint, a solution to the new path constraint $(x_0 > 0) \wedge ((x_0 - 1) \neq 50) \wedge (x_0 = 30)$ is $x_0 = 30$, which leads to a new test hitting the abort2 statement on line 10.

To sum up, for the example of Figure 1, a single dynamic symbolic execution augmented with automatic partial loop summarization can generate only two new tests that will lead directly to the two abort statements, plus one to negate $(x_0 > 0)$, and the systematic search will then stop after a total of 4 tests (namely, with x_0 equal to 10, 51, 30 and 0), instead of running forever.

2. BACKGROUND

2.1 Dynamic Test Generation

Dynamic test generation (see [11] for further details) consists of running the program P under test both concretely, executing the actual program, and symbolically, calculating constraints on values stored in program variables and expressed in terms of input parameters. Side-by-side concrete and symbolic executions are performed using a concrete store M and a symbolic store S, which are mappings from *memory addresses* (where program variables are stored) to concrete and symbolic values respectively. A *symbolic value* is any expression e in some theory $\mathcal{T}$ where all free variables are exclusively input parameters. For any memory address m, $M[m]$ denotes the *concrete value* at m in M, while $S[m]$ denotes the *symbolic value* at m in S. For notational convenience, we assume that $S[m]$ is always defined and is simply $M[m]$ by default if no symbolic expression in terms of inputs is associated with m in S. The notation $+$ for mappings denotes updating; for example, $M' = M + [m \mapsto e]$ is the same map as M, except that $M'[m] = e$.

The program P manipulates the memory (concrete and symbolic stores) through *statements* that are abstractions of the machine instructions actually executed. We assume a statement can be an *assignment* of the form $m \leftarrow e$ (where m is an address and e is an expression), a *conditional statement* of the form `if` e `then goto` ℓ where e denotes a boolean expression and ℓ denotes the location of the unique[1] next command to be executed when e holds, or `stop` corresponding to a program error or normal termination.

Given an input vector I assigning a concrete value I_i to the i-th input parameter, the evaluation of a program defines a unique finite[2] *program execution*. For a finite sequence w of statements (i.e., a control path w), a *path constraint* ϕ_w is a quantifier-free first-order formula over theory $\mathcal{T}$ that characterizes the input assignments for which the program executes along w. The path constraint is *sound and complete* when this characterization is exact, i.e., when every input assignment satisfying ϕ_w defines a program execution following w (soundness) *and* when every input assignment following path w is a satisfying assignment, or *model*, of ϕ_w (completeness).

Path constraints are generated by symbolically executing the program and collecting input constraints at conditional statements, as

[1] We assume program executions are sequential and deterministic.

[2] We assume program executions terminate. In practice, a timeout prevents non-terminating program executions and issues a runtime error.

illustrated in Figure 4 where the lines prefixed with * should be ignored for now. Initially, the path constraint is set to `true`. We assume that every program execution starts in the same initial concrete store, except for input values I_i which may differ. For every input I_i, we define initially $S[m] = x_i$ if m is an address storing input I_i (denoted $m \in I$) and where x_i denotes the symbolic variable corresponding to input I_i. By construction, all symbolic variable appearing in ϕ_w are variables x_i corresponding to program inputs I_i.

Systematic dynamic test generation [11] consists of systematically exploring all (or in practice many) feasible control-flow paths of the program under test by using path constraints and a constraint solver. After executing a whole-program control-flow path w, if a conditional statement of the form `if` e `then goto` ℓ is reached, any satisfying assignment of the formula $\phi_w \wedge c$ (respectively $\phi_w \wedge \neg c$) where $c =$ evaluate_symbolic(e), defines program inputs that will lead the program to execute the `then` (resp. `else`) branch of the conditional statement (assuming path constraints are sound and complete).

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* can be alleviated by performing symbolic execution *compositionally* [9], using symbolic execution *summaries*. For instance, a function summary ϕ_f for a function f is defined as a logic formula over constraints expressed in theory $\mathcal{T}$. 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 [9]. A summary thus represents symbolically a *set* of (intraprocedural) paths, which can be included in a path constraint in order to generate tests to cover new branches after the function f returns. By memoizing symbolic execution sub-paths as symbolic test summaries that are re-usable during the search, a systematic search can become exponentially faster than a non-compositional one. See [9] for further details. In this paper, we show how to generate automatically such symbolic summaries for input-dependent loops.

2.2 Loops and Induction Variables

The *control-flow graph* (CFG) of a function is a directed graph whose nodes represent program locations and whose edges represent possible transitions of the control flow between locations, i.e., there is an edge from n_1 to n_2 if the statement located at n_2 can be executed immediately after the statement located at n_1 in some program execution. The *start node* of the graph is the entry point in the function. An *exit* node is associated with all return statements. A node n_1 *dominates* another node n_2 if n_1 belongs to every path from the *start node* to n_2. A *loop* is a strongly connected component in the control flow graph. The *header* of a loop is a node of the loop which dominates all other nodes in the loop. Not all loops have a header. Loops with a header are called *reducible loops* or *normal loops*, and their header is unique. The nesting relation between reducible loops in a CFG forms a *loop tree*. Each node in a loop tree represents a loop with its header and its body (the list of nodes inside the loop), and its successors correspond to other nested loops. By convention, the *root* (a node without a parent) of the loop tree represents the entry point in the function with the start

```
1       void main(int x, y, z){ // x,y,z are inputs
2         int cy = 0, y1=0, done = 0;
3         while (1) {
4   GX:     if (x<=0) {
5             done = 1;
6             break;
7           }
8           y1=y;
9           while (1) {
10  GY:       if (y1<=0) break;
11            y1--;
12            cy = cy+1;
13          }
14  GZ:     if (z<=0) break;
15          x--;
16          z--;
17        }
18        if (cy == y*101)
19          error();
20      }
```

Figure 2: Example with nested loops.

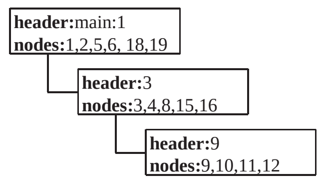

Figure 3: Loop tree for *main* in Figure 2.

node as its header. As an example, Figure 3 shows the loop tree for the *main* function in Figure 2. For instance, the statement at line 11 belongs to two loops, with headers at lines 3 and 9 respectively; the latter loop is called the *innermost loop* for line 11.

A program trace is a sequence of program locations as they are executed by a particular run of a program. For two locations in a trace, we write $l_m \simeq l_n$ if they represent the same program location, and $l_m \diamond l_n$ if they execute during the same function invocation. Given the static CFG for each function in a program, we define the following dynamic concepts relative to a program trace. A loop L is *entered* by l_j if $l_j \in L \wedge l_{j-1} \notin L$. (If a loop has a header, it is its entry point.) A loop L is *exited* by l_k if $l_k \in L \wedge l_{k+1} \notin L$. A loop exit l_k for L *matches* a loop entry l_j for L if $k = min\{i | j < i \wedge l_j \diamond l_i\}$ (the loop entry and exit behave as matching open and closed parenthesis). A *loop activation* is the sequence of locations in a trace between a loop entry and the matching loop exit.[3] An *iteration* for a loop activation with entry point l_j is a sequence of instructions $l_i, l_{i+1}, ..., l_m$ contained in that loop activation such that $l_i \simeq l_{m+1} \simeq l_j \wedge l_i \diamond l_{m+1} \diamond l_j$ and $\forall l_x \in \{l_{i+1}, , ..., l_m\} \neg(l_x \simeq l_j \wedge l_x \diamond l_j)$. The *Execution Count* (*EC*) for a loop activation is the number of loop iterations contained in that activation. A loop L is *active* at a program location l part of a trace if l is included in an activation of L.

For a particular activation of a loop L, we define:

DEFINITION 1. *An* **induction variable (IV)** *is a variable that changes by a nonzero constant amount during each iteration of that loop activation.*

DEFINITION 2. *A condition is* **linear** *if it is of the form* $(LHS \lhd RHS)$ *where* $\lhd \in \{<, \leq, >, \geq, \neq, =\}$*. A conditional statement is* **IV-dependent** *if its condition is linear and if a dummy variable assigned the value* $LHS - RHS$ *at that statement location is an IV inside* L.

[3] Since we assume every program execution terminates, every loop terminates.

```
1    evaluateSymbolic (P, I) {
2      Initialize memory M;
3      S = [m → x_i | m ∈ I];
4      pc= Initial program counter;
5      s=statement_at(pc);
6  *   LoopRecord L;
7      while s ∉ {stop} {
8  *     L = getCurrentLoop(pc);
9  *     if (L≠ NULL ∧ pc == L.header) {
10 *       L.iteration  = L.iteration + 1;
11 *       if (L.iteration  == 1) {
12 *         create L.IVT and L.GT tables;
13 *       } else {
14 *         updateIVT(L.iteration , L.IVT);
15 *         guess_postconditions(L.iteration ,L.IVT,L.GT);
16 *     } }
17       switch(s) {
18         case m ← e:
19 *         if (L≠ NULL ∧m ∉ L.MOD) {
20 *           L.MOD[m].V = M[m];
21 *           L.MOD[m].V^S = S[m];
22 *           if (L.iteration==1) {
23 *             L.IVT[m].V = M[m];
24 *             L.IVT[m].V^S = S[m];
25 *         } }
26           S = S + [m → evaluate_symbolic(e)];
27           M = M + [m → evaluate_concrete(e)];
28           pc=pc+1;
29         case if e then goto pc':
30           b=evaluate_concrete(e);
31           c=evaluate_symbolic(e);
32 *         if (L ≠ NULL) updateGT(pc,e,L.iteration ,L.GT);
33           if(b) {
34             path_constraint = path_constraint ∧ c;
35             pc=pc';
36           } else {
37             path_constraint = path_constraint ∧ ¬ c;
38             pc=pc+1;
39       }   }
40       s=statement_at(pc);
41   } }
```

Figure 4: Path constraint generation during dynamic symbolic execution. Lines prefixed by * are new.

```
updateIVT(iteration , IVT) {
  if(iteration == 2) {
    for v ∈ IVT {
      IVT[v].dV = M[v] − IVT[v].V; //1^st change in value
      IVT[v].dV^S = S[v] − IVT[v].V^S;
      IVT[v].V = M[v]; //used to compute future dV
    }
  } else {                  //purge failed IV candidates
    for v ∈ IVT {
      dV = M[v] − IVT[v].V; //current change in value
      if (dV ≠ IVT[v].dV) //changed by the same amount?
        remove v from IVT; // v is not an IV
      else
        IVT[v].V = M[v]; //used to compute future dV
    }
  }
}
```

Figure 5: *updateIVT* ensures that only valid IV candidates are in IVT.

when?		IVT changes			
Iteration	line	Var	V	dV	$V^{\mathcal{S}}$
1	3	-	-	-	-
1	6	c	0	-	0
1		c	0	-	0
	7	p	0	-	0
1		c	0	-	0
		p	0	-	0
	8	x	10	-	x_0
2	3	c	1	1	0
		p	1	1	0
		x	9	-1	x_0
3	3	c	2	1	0
		~~p~~	3	1	0
		x	8	-1	x_0
4	3	c	3	1	0
		x	7	-1	x_0
....					
10	3	c	9	1	0
		x	1	-1	x_0

Figure 6: Finding IVs in the example of Figure 1.

DEFINITION 3. *A* **guard** *of a loop L is a conditional statement that has one target inside the loop and the other outside the loop. A guard for an activation of L is* **IV-dependent** *if its condition is IV-dependent for that activation of L.*

3. SUMMARIZING INDIVIDUAL LOOPS

We now describe how dynamic symbolic execution can be extended to automatically generate partial loop summaries that are usable for test generation. Figure 4 shows a modified procedure evaluateSymbolic with new lines prefixed with a *.

To simplify the presentation, we start by discussing in this section how to summarize loops in programs that contain a single reducible loop L with a known header $L.header$ in a single non-recursive function (as in the example of Figure 1). We also assume for now that, for any given statement, we can tell whether its location is in L or not. Also assume that all basic data types are integers of the same size. These simplifying assumptions will be lifted in the next sections.

These assumptions however do not tell us if the loop has IVs or IV-dependent guards. We infer this information dynamically. We do so using two tables, one for "IV candidates" (IVT) and one for "IV-dependent guard candidates" (GT). These tables (described later) record both concrete and symbolic values of variables or conditions that might be induction variables or IV-dependent guards. Concrete values are used to discard non-IV candidates from the tables, and symbolic values are used later during loop summarization.

For every location, we start by finding the loop it belongs to (line 8 of evaluateSymbolic in Figure 4), if any. If this is the header of the loop (line 9), then a new loop iteration starts. If this is the beginning of the first iteration (line 11), then we create the two tables (line 12), otherwise we update the IV Table (line 14) and check if we can guess loop invariants (line 15), as discussed later.

3.1 IV candidates Table (IVT)

In order to find IVs, we track the variables modified inside the loop and eliminate those that are unchanged or that change by different amounts in two consecutive iterations. We maintain this information in an Induction Variable candidates Table (IVT) with one entry per active candidate variable containing:

- $V =$ concrete value when the control reaches the header.
- $dV = V_{(2)} - V_{(1)}$ the change in concrete value between the first two iterations.
- $V^{\mathcal{S}} =$ symbolic value at the loop entry.
- $dV^{\mathcal{S}} = V^{\mathcal{S}}_{(2)} - V^{\mathcal{S}}_{(1)} =$ symbolic change in value between the first two iterations.

When the loop header is encountered for the first time, an empty IVT is created. Each variable modified in the first iteration gets an entry in the IVT (indexed by the address of the variable), where we record its starting concrete and symbolic values (lines 22-24 of *evaluateSymbolic* in Figure 4). We record those initial values before the first time the variable is written inside the loop because we need the initial value which the variable had at the loop entry. The use of the MOD table (lines 19 to 21) will be explained later, when we discuss nested loops.

After the first iteration, we only update or remove entries from the IVT table, and keep just those candidates that change by the same constant amount between every two iterations. Procedure *updateIVT* shown in Figure 5 performs these updates. When we encounter the loop header the second time, we compute and store in the IVT the difference $dV = V_{(2)} - V_{(1)}$ between the current value and the saved one for each variable in the table (line 4 of updateIVT). We then save the symbolic value of the difference $dV^{\mathcal{S}} = V^{\mathcal{S}}_{(2)} - V^{\mathcal{S}}_{(1)}$, for potential summarization later (in Figure 10), and we update the current value V. Each time we reach the header at subsequent iterations, for each variable in IVT we compare the initially saved dV with the current dV (line 10 in Figure 5), and remove entries where they do not match since IVs must change by the same amount at each iteration. We update the IVT once per iteration, rather then after each memory operation, because for each induction variable we need the cumulative effects of the entire iteration, rather than local changes.

Figure 6 shows how the IVT is updated during dynamic symbolic execution of the example in Figure 1 with $x_0 = 10$. ($dV^{\mathcal{S}}$ is always equal to dV for this example and not shown in the figure.) When the control reaches the loop header for the first time (iteration 1 in line 3 of Figure 1), an empty IVT is created. When we perform the write at line 6 and variable *c* changes from 0 to 1, we add an entry for *c*, and record its starting value 0. We also add one entry for each of *p* and *x* at lines 7 and 8 respectively. Variable *p* changes by 1 between the first two iterations and by 2 between the 2^{nd} and 3^{rd} iteration, so it is then removed from the IVT, since it cannot be an IV.

3.2 Guard candidates Table (GT)

Similarly, we maintain a Guard candidates Table (GT) in order to detect IV-dependent conditions for possible guards of a loop. When the loop header is encountered for the first time, an empty GT is created (line 12 of Figure 4). Procedure updateGT shown in Figure 7 is called to maintain and use this table whenever a conditional statement is executed during dynamic symbolic execution (see line 32 of Figure 4). We track guard candidates that are conditional statements with a symbolical condition of the form $(LHS \lhd RHS)$. Lines 2 and 3 of updateGT filter out candidates that do not satisfy this requirement. For each guard candidate, we store:

- $B =$ concrete boolean value.
- $D = (LHS - RHS)$ the distance between the two operands. Note that $B = (LHS \lhd RHS) \equiv ((LHS - RHS) \lhd 0) \equiv (D \lhd 0)$, where $\equiv$ denotes logical equivalence.
- $D^{\mathcal{S}} =$ the first symbolic value for D. This is used to compute $EC^{\mathcal{S}}$ (see below).
- $dD = D - old(D)$ the change in concrete value for the distance.
- $EC =$ the expected execution count. This is the number of loop iterations that the condition stays unchanged (evaluates to B), assuming that $(LHS - RHS)$ is indeed an IV.
- $EC^{\mathcal{S}} =$ the symbolic value for EC.
- $hit =$ the number of times the candidate was encountered. An IV-dependent guard should be encountered once at each iteration.
- $loc =$ location in the path constraint for precondition.

```
1  updateGT(pc, cond, iteration, GT) {
2   if(  cond is not symbolic
3       ∨ cond is not (LHS ◁ RHS) with ◁ ∈ {<, ≤, >, ≥, ≠, =}
4       ∨ (iteration > 1 ∧ pc ∉ GT)
5       ∨ both targets of statement_at(pc) are inside loop)
6      return;
7   B = evaluate_concrete(cond)? true: false;
8   D = evaluate_concrete(LHS − RHS);
9   if (iteration==1) {
10    GT[pc]=new entry, with GT[pc].hit==0
11    GT[pc].B = B;
12    GT[pc].D = D;
13    GT[pc].D^S = evaluate_symbolic(LHS − RHS);
14    GT[pc].loc = current location in path_constraint;
15  }
16  else if (iteration==2) {
17    GT[pc].dD = D − GT[pc].D;
18    D^S = evaluate_symbolic(LHS − RHS);
19    dD^S = D^S − GT[pc].D^S;
20    switch ('◁') {
21      case ≤: {
22        if (D > 0) {
23          if (dD < 0) {
24            insert constraint D^S > 0 ∧ dD^S < 0 at
25            location GT[pc].loc in path_constraint;
26            GT[pc].EC = (GT[pc].D − GT[pc].dD − 1) / (−GT[pc].dD);
27            GT[pc].EC^S = (GT[pc].D^S − dD^S − 1) / (−dD^S);
28          } else ...
29        } else ...
30      }
31      ...;
32    }
33  }
34  GT[pc].hit=GT[pc].hit+1;
35  if(GT[pc].hit≠iteration) //candidates should execute
36    remove pc from GT;       //once every iteration
37  if(GT[pc].B ≠ B∧ GT[pc].done ∧ iteration==GT[pc].EC + 1)
38    guess_preconditions(pc,GT);
39  if(GT[pc].B ≠ B∨ GT[pc].dD ≠ D − GT[pc].D)
40    remove pc from GT;
41  else
42    GT[pc].D = D;
43 }
```

Figure 7: *updateGT* ensures that only IV-dependent guard candidates are in GT.

During the first loop iteration, we add guard candidates as we execute them, indexed by their program location. For each new candidate, we check whether its condition is symbolic and matches a pattern we handle (lines 2 and 3); if so, we compute and record its initial value B (lines 7,11), $D = (LHS - RHS)$ (lines 8,12), and $D^{\mathcal{S}}$ (line 13), and we initialize and increment (line 34) its hit count which becomes 1 for new candidates. At line 14 we save the current location in the path constraint. If this candidate is indeed an IV-dependent guard, and we use it for summarization, then we use this location to remove over-restrictive constraints and insert new ones. If the candidate is not an IV-dependent guard, and we remove it from the GT table, then we use this location to remove the constraint inserted at line 24 of $updateGT$. During the second loop iteration, we compute and save the difference $dD = D - old(D)$ (line 17) in order to check if D changes by the same amount during every other iteration. We also compute and save EC and $EC^{\mathcal{S}}$. For instance, if $\lhd$ is $\leq$ (lines 21-27) then $EC = (D - dD - 1)/-dD$. However, this only holds if $D > 0 \wedge dD < 0$ (not for $D = 0$, for instance), so we insert in the path constraint a constraint under which this holds (line 24). At subsequent iterations, we remove GT entries which cannot be IV-dependent guards (lines 36, 39-40). Lines 37-38 are used for loop summarization and are explained later.

Figure 8 shows the changes of the GT for the example of Figure 1 (for simplicity we don't show $D^{\mathcal{S}}$ and loc fields). When

when?		GT changes					
Iteration	line	B	D	dD	EC	$EC^{\mathcal{S}}$	hit
1	3	-	-	-	-	-	-
1	4	F	10	-	-	-	1
2	4	F	9	-1	10	x_0	2
...							
10	4	F	1	-1	10	x_0	10
11	4	T	0	-1	10	x_0	11

Figure 8: Finding guard candidates in Figure 1.

```
guess_preconditions(pc, GT) {
 set ◁ to <;
 for l ∈ GT in insertion order {
  remove the constraints associated to l which were inserted
  in path_constraint after GT[l].loc during the current
  function activation, *except* those inserted by updateGT.
  if (l ≠ pc) {
    Constraint = GT[pc].EC^S ◁ GT[l].EC^S;
  } else {
    Constraint = GT[pc].EC^S > 0;
    set ◁ to ≤;
  }
  insert constraint Constr at location GT[pc].loc
  in path_constraint;
}}
```

Figure 9: *guess_preconditions:* we are about to exit the loop due to the conditional jump at pc. The preconditions must ensure that we execute at least one iteration of the loop, and that the condition at pc is the first one to exit the loop.

the loop is activated (iteration 1 line 3), an empty GT is created. When the control reaches line 4 for the first time, we add an entry identified by that program location (the current value of the pc) with the condition $x \leq 0$, and store the values $B = \texttt{False}$, $D = 10$ and $D^{\mathcal{S}} = x_0$. We also record in loc the current location in the path constraint. The second time we visit line 4, we compute $D = 9, dD = 9 - 10 = -1, EC = (10 - (-1) - 1)/-1 = 10$ and $EC^{\mathcal{S}} = (x_0 - (-1) - 1)/-1 = x_0$ since $\lhd$ is $\leq$ and $D > 0 \wedge dD < 0$. We also insert the condition $x_0 > 0$ in the path constraint at loc. At subsequent iterations, we only check that it is still a valid candidate and update D and hit.

3.3 Loop Summarization

Recall from Section 2.2 that for a given program trace, a *loop activation* is the sequence of instructions between a loop entry and the matching loop exit. Since a loop exit is inside the loop and the location of the next statement is outside the loop, a loop exit is always a loop guard. At a loop exit, the current path constraint and symbolic store represent only the current execution with a specific number of loop iterations. We now describe how to *generalize* the symbolic execution of that loop from that specific number of iterations to a *set* of possible loop executions.

The key idea is to generate a *loop summary* which characterizes (perhaps only partially) such a set of executions. The generation of a loop summary is performed in two steps: (1) *generate preconditions*, and (2) *generate postconditions*.

Before we describe these two steps, we point out the following property, which we can prove and which will be useful shortly.

LEMMA 1. *(GT Completeness) Given a loop L, if every loop guard is symbolic and IV-dependent, then*

1. *every guard is executed exactly once during each iteration.*
2. *the relative execution order of the guards is fixed for all iterations.*
3. *all guards are in the GT table at the beginning of the second iteration (if there is such an iteration).*

```
symbolic_update(v, V0^S dV^S, EC^S) {
  V^S = V0^S + dV^S * (EC^S - 1);
  S = S + [v → V^S];
}
guess_postconditions(iteration, IVT, GT) {
 for l ∈ GT in insertion order {
   if GT[l].EC == iteration { // last iteration predicted
     for v ∈ IVT {
       symbolic_update(v, IVT[v].V^S, IVT[v].dV^S, GT[l].EC^S);
     }
     GT[l].done = true;
   }
   break;
 }
}
```

Figure 10: *guess_postconditions:* If a valid guard candidate predicts the start of the last full iteration, update the symbolic memory for the IV candidates.

Loop Precondition. Consider an input-dependent loop L with a symbolic IV-dependent guard G at location pc and which executes a fixed $GT[pc].EC$ number of iterations on a given dynamic symbolic execution. Since by construction we have $GT[pc].EC =$ evaluate_concrete($GT[pc].EC^{\mathcal{S}}$), the value of the symbolic expression $GT[pc].EC^{\mathcal{S}}$ determines the number of iterations of that loop. Since this expression is symbolic, i.e., input dependent, we can control this number of iterations via test inputs. For instance, the (input) constraint $GT[pc].EC^{\mathcal{S}} > 0$ characterizes the set of program executions where the body of loop L is executed *at least* once.

When a loop L has more than one symbolic IV-dependent guards, the guard with the smallest execution count is the one which will "expire first" and hence controls the number of loop iterations. Let us denote this guard by G. Moreover, thanks to Lemma 1, we know that there is a fixed total ordering among all such guards. Let us write $G_1 < G_2$ if guard G_1 precedes guard G_2 in this ordering.

When the loop exits, *all* the constraints that were inserted for all loop guards of L in the current path constraint characterize the inputs for which L iterates exactly $GT[G].EC$ times. In order to generalize this specific execution, we remove all such constraints and replace them by the new constraint

$$GT[G].EC^{\mathcal{S}} > 0 \wedge$$
$$\forall G' \neq G \begin{cases} GT[G].EC^{\mathcal{S}} < GT[G'].EC^{\mathcal{S}} & \text{if } G' < G \\ GT[G].EC^{\mathcal{S}} \leq GT[G'].EC^{\mathcal{S}} & \text{if } G < G' \end{cases}$$

injected at the location of the first removed constraint in the path constraint. This new constraint forces the loop to iterates at least once, and forces all other symbolic IV-dependent guards G' to expire after guard G. In other words, this loop precondition defines a set of program executions satisfying such properties. Note that other program executions outside this set will be eventually explored/covered by dynamic test generation when those new individual constraints are flipped later during the systematic search.

The new constraint is computed by procedure $guess\_preconditions$ shown in Figure 9, which also updates the path constraint. This procedure is called in line 38 in updateGT (Figure 7) when the loop is exited by guard G (detected in line 37). Updating the path constraint at this point (instead of, say, the last visit of the loop header $L.header$) ensures that all the constraints due to loop guards are in the path constraint and hence that all of those can be removed.

Loop Postcondition. Given the loop precondition defined above, we want to capture all the side-effects through induction variables whose value depends indirectly on inputs. When all side-effects inside the loop are exclusively through IVs, the symbolic values of IVs define the set of all possible states reachable by program executions satisfying the loop precondition.

Each time procedure $evaluateSymbolic$ reaches the loop header $L.header$ in line 15 of Figure 4, procedure $guess\_postconditions$ (shown in Figure 10) is called and searches the GT table for a guard G whose EC indicates that this is the start of the last complete iteration. If this is the case, remember $GT[G].EC^{\mathcal{S}}$ is the symbolic expression that defines the number of loop iterations for any program execution characterized by the loop precondition we defined above. Then, for each IV v in IVT (line 8 of $guess\_postconditions$), we update the symbolic store S to set $S[v]$ equal to the symbolic value $V^{\mathcal{S}} + dV^{\mathcal{S}} * (GT[G].EC^{\mathcal{S}} - 1)$, where $V^{\mathcal{S}}$ is symbolic value of v at the entry of the loop, and $dV^{\mathcal{S}}$ is the symbolic delta value between any two loop iterations. Indeed, this expression defines/predicts the value of v at the beginning of the last full loop iteration parametrized by $GT[G].EC^{\mathcal{S}}$, which itself defines the number of loop iterations when G is the first expiring symbolic IV-dependent loop guard.

Note that we perform the symbolic memory update when the execution reaches the loop header at the beginning of the last complete loop iteration rather than when the loop exits. This way, we are able to generate constraints on IVs if there are tested inside the body of the loop during the last loop iteration until the loop exits (like the assert in line 5 of Figure 1, while removing all constraints due to guards in L is easier after they are all included in the path constraint, i.e., when the loop exits.

Considering again the simple example of Figure 1 with an initial value of $x_0 = 10$, there is only one loop guard (located at line 4) and its $EC^{\mathcal{S}}$ is x_0 (see the GT in Figure 8). At the beginning of the 10^{th} loop iteration, line 15 of $evaluateSymbolic$ in Figure 4 calls $guess\_postconditions$ (Figure 10), which detects that this is the start of the last complete iteration of the loop. There are then two IVs in the IVT (see Figure 6), namely c and x. $guess\_postconditions$ then updates their symbolic values as $V^{\mathcal{S}} + dV^{\mathcal{S}} * (GT[G].EC^{\mathcal{S}} - 1)$, i.e., the symbolic value of c becomes $0 + 1 * (x_0 - 1)$, that is $x_0 - 1$, while the symbolic value of x becomes $x_0 + (-1) * (x_0 - 1)$, that is 1. Now, c is symbolic, and so is the condition at line 5 for which a constraint $((x_0 - 1) \neq 50)$ is added to the path constraint. When the loop guard at line 4 is reached for the 11^{th} time, procedure $updateGT$ (Figure 7) in line 37 detects that the boolean condition B of the guard indeed changes, that this guard was used to update the symbolic memory earlier (via the $done$ boolean flag), and the iteration is one greater than the predicted EC. After all these sanity checks, $guess\_preconditions$ is called to update the path constraint. All the constraints $(x_0 > 0) \wedge (x_0 > 1) \wedge ... \wedge (x_0 > 9) \wedge (x_0 - 10 \leq 0)$ accumulated inside the loop are removed and replaced by just $GT[G].EC^{\mathcal{S}} > 0$, that is $x_0 > 0$, at the location of the first constraint in the path constraint. Later, when the constraint $((x_0 - 1) \neq 50)$ is negated, a solution to the new path constraint $(x_0 > 0) \wedge ((x_0 - 1) = 50)$, namely $x_0 = 51$, leads to a test hitting the abort1 statement. The case for the assert2 statement is similar and as discussed in Section 1.

The following theorem defines the correctness (and hence the guiding design principles) of the algorithms presented in this section.

THEOREM 2. *(Correctness) Consider a program P with a single loop L. Assume that path constraint generation performed during dynamic symbolic execution is sound and complete, that all variables modified inside the loop are induction variables, and that every loop guard is symbolic and IV-dependent. Then, if the loop is executed at least three times during a loop activation, loop summarization is performed, and the resulting generalized path constraint is sound and complete.*

In practice, loops may have symbolic guards that are not IV-dependent, or have side-effects through non-IV variables. In those cases, our algorithm for loop summarization can still be used as a *"best effort"* approach to limit path explosion but without soundness and completeness guarantees. However, our algorithm may still be sound and complete in some of those cases. For instance, consider again the simple example of Figure 1. This example contains a loop which has side-effects via variable p, which is not an induction variable. Therefore, the loop summarization performed by our algorithm for this example is not sound and complete *for all possible contexts* in which the loop *may* be executed. However, for the *specific context* provided by function *main* in Figure 1, our loop summarization is actually sound and complete, since variable p is never read after the loop terminates and therefore does not influence the control flow afterwords. Why would a loop have side-effects that are not read afterwords? In practice, this can happen when populating data structures inside a loop that are to be used later by another program or module that is not the focus of the current testing session. For instance, an image processor parses input files, populates data structures, and then pass pointers to such data structures to a graphics card; if we are interested in finding security-critical buffer-overflow bugs in the image parser itself, populating some of the data structures may look like write-only operations during symbolic execution of the image parser itself.

4. SUMMARIZING NESTED LOOPS

In this section, we lift the simplifying assumptions made in Section 3 regarding the program structure: we now allow an arbitrary number of possibly recursive functions with an arbitrary number of possibly nested loops. But we still assume for now that we know the control flow graph, including its loop structure.

In the presence of recursion, a loop can have multiple nested activations at the same time. Each activation requires its own candidate tables which are no longer needed after it exits. In a manner similar to function activation records, we keep track of loop activations using *loop records* which are maintained using a *loop stack*. The record for the innermost loop activation is always on top of the stack. The *loop context* is the dynamic state of the loop stack.

Procedure *getCurrentLoop* uses the CFG to determine the static loop membership of the current program counter pc. When a loop is activated, *getCurrentLoop* creates a corresponding loop record and pushes it on the loop stack thus entering a new *loop context*. For this purpose, a function call is considered entering a new loop. This is consistent with the loop tree structure where the root is the function entry point, not a real loop. Procedure *getCurrentLoop* always returns the record on top of the loop stack, which contains the IVT and GT tables for the current loop. When a function terminates or one or more loops terminate, their records are removed from the stack.

Consider a variable v which is modified inside a function call or a loop nested inside some loop L, but which is not written by any statement immediately contained in L. In this case v does not get added to L's IVT at lines 23- 24 of *evaluateSymbolic* (Figure 4), and yet it may be an induction variable for L. To account for such cases, we use the MOD table to propagate to parent loops information about variables that are modified inside children loops. This is why each time a new variable is modified in a given context, we save its initial concrete and symbolic values at lines 21-20 of *evaluateSymbolic*. When a record is removed from the context stack, we propagate the information in MOD to the record of the enclosing context, and if the parent loop is executing its first iteration,

(a)	(b)	(c)	(b)	(d)	...
...	...	...	...	...	...
$L_a[1]$	$L_a[1]$	$L_a[1]$	$L_a[1]$	$L_a[1]$	...
	$L_b[3]$	$L_b[3]$	$L_b[3]$	$L_b[3]$	
		$L_c[9]$		$L_d[9]$	

Figure 11: Loop Contexts for Figure 2. Each function call or loop activation creates a new context. $L_x[H]$ denotes a loop record on top of the stack for context (x) and for a loop activation with header at program line H.

we then also update its IVT. However, we do not change the entries for variables already changed in the parent context. This way, all modified variables become IV candidates for a loop, even those modified only in a sub-loop or sub-function. Except for the maintenance of the loop stack and the propagation of the MOD table, summarization in the general case is performed in a similar way to the single loop case. It is always performed one loop at a time, independently of the summarization of other loops.

In practice, we limit the depth to which we propagate MOD information in order to reduce the memory overhead. Also note that is possible that a guard exits more than one loop. For instance, if L' is nested inside L, a conditional statement inside L' may have a target outside L. However, we believe it is unlikely that such a conditional statement is an IV-dependent guard for the outer loop, and in the spirit of our "best effort" approach, our algorithm does not handle such corner cases. We can nevertheless prove the following.

THEOREM 3. *(General Correctness) Consider a program P containing only reducible loops. Consider a loop L whose guards are all symbolic, IV-dependent and exits only L (therefore, no guard of L is contained inside a loop nested in L). Further assume that path constraint generation performed during dynamic symbolic execution is sound and complete, and that all variables modified inside L are induction variables. Then, if L is executed at least three times during an activation, loop summarization is performed, and the resulting generalized path constraint is sound and complete.*

Consider the example in Figure 2 and the initial input values $x_0 = 10, y_0 = 20, z_0 = 30$. Figure 11 shows the evolution of the loop stack. When *main* is called, a new loop record is pushed on the loop stack, and the program enters context (a). When the outer loop is activated we create and push on the loop stack loop record L_b, and enter context (b). The first activation of the inner loop is executed in context (c) with L_c on top of the stack. At the end of this activation we pop L_c and return to context (b). The second activation of the inner loop is performed in context (d) with L_d on top of the stack, etc.

During the first activation of the inner loop, variables $y1$ and cy are recorded in $L_c.MOD$ and $L_c.IVT$. At the end of this activation, the summarization of L_c replaces the constraints due to guard GY in the path constraint by the loop precondition $y_0 > 0$. At that point of the symbolic execution, we have $S[y1] = 0$ and $S[cy] = y_0$. When we pop L_c from the context stack, we propagate $L_b.MOD[cy] = L_c.MOD[cy]$, and since L_b is still during its first iteration, we also copy $L_b.IVT[cy] = L_c.IVT[cy]$. Note that, due to the assignment at line 8, $y1$ is already contained in L_b's tables and no updates for $y1$ are needed. At the end of the second activation of the inner loop, we summarize L_d and we then have $S[cy] = y_0 + y_0$. Each time we exit the inner loop in Figure 2, we return to context (b). In this context, we remove $y1$ from $L_b.IVT$ because it has the same value at the loop header. During the second iteration of L_b, we compute $IVT[x].D^{\mathcal{S}} = -1, IVT[cy].D^{\mathcal{S}} = y_0, IVT[z].D^{\mathcal{S}} = -1, GT[GX].EC^{\mathcal{S}} = x_0$ and $GT[GZ].EC^{\mathcal{S}} = z_0$. During this iteration, we also insert the constraints $x_0 > 0$ and $z_0 > 0$ to the path constraint, in this case as a result of executing line 24 of *updateGT* when we compute a symbolic execution count. Because $x_0 = 10$ and $z_0 = 30$, we have $x_0 < z_0$ and we exit the loop through GX after $x_0 = 10$ iterations. At the beginning of the last iteration, we perform the following symbolic updates using $L_b.GT[GX].EC^{\mathcal{S}} = x_0$ as the execution count: $S[x] = x_0 - (x_0 - 1), S[cy] = y_0 * (x_0 - 1), S[z] = z_0 - (x_0 - 1)$. After we summarize the inner loop one more time, we get $S[cy] = y_0 * (x_0 - 1) + y_0$. After the last two decrements inside L_b (in lines 15 and 16), we reach line 18 with the symbolic state $S[x] = x_0 - (x_0 - 1) - 1 = 0, S[cy] = y_0 * x_0, S[z] = z_0 - (x_0 - 1) - 1 = z_0 - x_0$ and $S[y] = y_0$.

When we exit the outer loop, summarization of that loop removes the constraints at GX and GZ from the path constraint, and replaces those with the loop precondition $(x_0 > 0) \wedge (x_0 \leq z_0)$ (since $GX < GZ$). At line 18, the conditional statement is executed and the constraint $y_0 * x_0 \neq y_0 * 101$ is added to the path constraint. Later, when the constraint $y_0 * x_0 \neq y_0 * 101$ is negated, a solution to the new path constraint $(x_0 > 0) \wedge (x_0 \leq z_0) \wedge (y_0 * x_0 = y_0 * 101)$, for instance $x_0 = 101, y_0 = 1, z_0 = 101$, leads to a test hitting the error statement.

5. IMPLEMENTATION ISSUES

Here we remove the rest of the simplifying assumptions from Section 3 regarding the program structure. We no longer assume that we have knowledge about the static structure of the program or size information for variables.

5.1 Detecting Loops Dynamically

In practice we do not have the control flow graph for program functions and we want to infer loop information dynamically. The *Dynamic CFG* (DCFG) is the dynamically built CFG whose nodes are the statements executed so far in the current function. It is represented as a regular CFG together with a *current node* which corresponds to the current pc. There is one DCFG per function activation record, shared by all loop records inside that function. When we create a new loop record for a function, we also create a new DCFG which contains only the root node. For each instruction, when we call *getCurrentLoop*, it first updates the DCFG before it uses it. If there is no node corresponding to the current pc, we create one. We add an edge in the current DCFG between the nodes corresponding to the previous pc and the current one, if such an edge does not already exist. The node corresponding to the current statement becomes the current one in the DCFG. We avoid the expensive computation of the loop tree and only invoke it when a new edge is added between two existing nodes. When an existing edge between existing nodes is traversed, there is no need to change the DCFG or its loop structure; we just update the current node. For edges between an existing node and a new one, we simply assume that the new node is part of the current loop (potentially subject to the *uncertain exit* issue described bellow) and we patch the loop information correspondingly, in constant time.

LEMMA 4. *Let L be the innermost loop containing the current node of a DCFG built based on a program execution. Then the addition of the next node in the execution does not change the header of L or any of its enclosing loops.*

Therefore the changes in the DCFG are consistent with the loop stack, by not changing the headers of the active loops.

When compared to static loop detection, dynamic loop detection suffers from two limitations. The first one is that we only detect a loop L when we execute its header for the second time. As a result, the entire first iteration of L looks as if it was unrolled and part of L's parent loop. We call this a *lost iteration*. The effect on summarization is that detecting a loop requires an additional iteration for the "unrolled" loop, and summarization covers only the remainder of that loop's activation. The unrolled part of a loop executes in the context of the parent and does not impact its summarization. The IV candidates of the parent are not affected, because all the writes inside the parent's activation (which contains the unrolled sequence) must be accounted for anyway. Guard candidates inserted in parent's GT are later discarded based on mismatches between the number of iterations and the hit counter.

The second limitation is that when we execute a new statement not already in the DCFG, we cannot tell whether it belongs to the last active loop or not. As a result, there are cases when we do not know that we actually have exited a loop until we reach a return statement, or the header of a parent loop. Whether we are still inside of the loop or not depends on future instructions, which we have not seen yet. We call this issue *uncertain exit*. For instance, when the control reaches line 5 from Figure 2 we cannot tell just by looking at the DCFG if we are still inside the loop or not: at line 6 there could be any instruction, including a *return* or *continue*. The effect on summarization is that we have to be conservative at line 5 of *updateGT* and treat all conditional statements as possible guards. We also have to be conservative and consider that we do not exit a loop L unless we follow an existing edge in the DCFG that leads outside L, or we reach a *return*, or we reach a statement that dominates L's header.

5.2 Variable Sizes

When a memory location is modified for the first time in a context, we also record in the IVT the number of bytes written. For an IV candidate v, this is the inferred size of the corresponding variable and it is used whenever we need to find the amount of memory to read to obtain the concrete value $M[v]$. If a variable v has a size larger than the maximum size that can be written in a single statement, then a logical update of v may be performed in several write operations, say a write to $v.low$ and $v.high$. Our current implementation (see next section) treats $v.low$ and $v.high$ as distinct IV candidates. In practice, the probability for errors due to this scenario is small. If v is not an IV, then the likelihood that either of $v.low$ or $v.high$ behaves as an IV is small. If v is indeed an IV and the increment is not very large, then most likely only $v.low$ changes and is detected as an IV.

6. EXPERIMENTAL RESULTS

We have implemented the algorithms presented in the previous section in the whitebox fuzzer SAGE [12], which uses the Z3 SMT solver [7]. SAGE performs dynamic symbolic execution at the x86 binary level, does not require source code, and is optimized to scale to very long program executions possibly with billions of x86 instructions.

We report in this section preliminary experiments conducted with our prototype implementation and the ANI image parser embedded in Windows 7. Running this parser with a sample well-formed ANI (ANimated Icon) input file of 13,302 bytes results in a program execution with 1,874,649 x86 instructions executed, including 1,417,441 instructions executed after the first input byte is being read from the input file. The number of unique x86 instructions executed is about 30,000 which are spread over 15 different Windows dlls.

The Table in Figure 12 presents experimental data obtained during a single dynamic symbolic execution and constraint solving along the program execution defined when parsing this sample ANI file. The table presents data for both regular SAGE and with loop summarization. A "-" in the table means the data is undefined or unknown.

Mode	Loop Sum.	Regular
Unique branches	5,455	-
Unique symbolic branches	300	-
Unique loops	231	-
Unique loops w. symb. cond.	19	-
Unique loops w. right guesses	6	-
Loop summarizations	25	-
Constraints removed	78	-
Total symbolic updates	56	-
Constraint cache queries	26,260	26,198
Total solver queries	9,163	9,155
Solver queries SAT	383	384
Solver queries UNSAT	8,780	8,771
Solver queries timeout	0	0
Total seconds spent in solver	858	769
Total analysis time (secs)	2,600	1,658
Peak memory usage (Mb)	512	366

Figure 12: Experimental results for ANI.

During the entire program execution, 5,455 unique conditional statements are executed, among which 300 are symbolic, and 231 unique loops are detected dynamically by our implementation (see the previous section). Among these 231 loops, 19 are detected to contain one or more symbolic guards and are thus possibly input-dependent. Our algorithm is able to successfully guess the number of iterations for 6 of those 19 loops. Those 6 loops are summarized 25 times in total, i.e., there are 25 summarized loop activations (some of 6 loops are thus executed and summarized successively more than once during the entire program execution). These 25 summarizations remove a total of 78 constraints out of the path constraint and perform a total of 56 symbolic memory updates (i.e., there are 56 instances of induction variables modified during the 25 loop summarizations). The total number of constraints in the path constraint is 26,260, but only 9,183 are unique (duplicates are removed as cache hits). To solve those constraints, 9,183 calls are made to the Z3 SMT solver, resulting in 383 satisfiable constraints, 8,780 unsatisfiable constraints and no (5secs) timeouts. The total execution time spent in Z3 is 858 secs. Overall, symbolic execution and constraint solving takes 2,600 secs and requires 521 Mb of memory.

We observe that few loops are detected to have symbolic guards (19 out of 231), among which only 33% (6) of those are guessed correctly and hence summarized. To find out why, we visually inspected these 19 loops and collected additional statistics (not shown here). Some of those 19 loops that are not summarized have non-IV guards typically involving pointers as in the following pattern

```
...
for (j = 0; j < x; j++) { // x is input-dependent
   if (array != NULL) // non IV-dependent loop guard
      array[j] = data;
...
```

Another reason is that some input-dependent loops are executed only once or twice, which is insufficient for our dynamic loop detection and summarization algorithms to kick in. We also tried 12 other ANI input files, but were unable to exercise such loops a larger number of times – see the table in Figure 13.

Input size (bytes)	2504	11326	7908	3272	4592	11346	1700	800	15902	6362	4100	818
Total analysis time (secs)	391	649	958	465	205	746	362	10	22	1611	555	249
Unique branches	5459	5446	5479	5461	5460	5440	5451	3008	3146	5454	5474	5447
Unique symbolic branches	272	269	263	271	275	272	254	7	105	288	272	282
Unique loops	231	228	232	230	229	228	229	125	133	229	232	230
Unique loops w. symb. cond.	17	19	16	17	19	20	16	0	4	18	17	17
Unique loops w. right guesses	2	5	5	5	1	5	0	0	0	5	5	2
Loop summarizations	22	23	30	36	2	17	0	0	0	48	23	5

Figure 13: Experimental results for 12 other ANI input files.

We also observe that the number of iterations for the loops that are summarized is low, resulting in only 78 constraints being removed from the path constraint. However, the total number of unique constraints (total solver queries) remains about the same: the removal of constraints due to loop summaries is offset by new constraints due to program branches testing IV-values part of post-conditions and made symbolic by the 56 symbolic updates. Moreover, most of the removed constraints and of the new constraints on IV-values are satisfiable, so the overall number of SAT constraints remains about the same (for this experiment with a small well-formed input file where symbolic loops are executed only a small number of times).

Remember the model of each SAT constraint is used to define a new test input file. Interestingly, the SAT constraints with loop summarization gives the systematic search a head start as the new constraints immediately exercise more new code: the incremental instruction coverage obtained by running all 383 generation-1 children tests with loop summarization is about 33% higher than the incremental instruction coverage obtained by running all 384 generation-1 children tests without loop summarization.

This benefit comes comes at the cost of an about 50% overhead for both total runtime and peak memory usage. This overhead is due to all the extra book-keeping needed by our loop detection and summarization algorithm, and is *not* due to constraint solving, which remains roughly the same. We emphasize that our current prototype is a first non-optimized implementation of our new algorithm – further optimizations might be able to reduce this cost.

Note that the long execution time of 1,658 secs for regular SAGE is due to turning off all unsound optimizations and heuristics (such as bounds on the number of constraints at specific branches, constraint subsumption, etc.) used by default in SAGE (see [12]).

The results of similar experiments with 12 other ANI input files (including a randomly-generated bogus 800-bytes file) are shown in Figure 13. These numbers confirm the general trends observed above.

7. RELATED WORK AND CONCLUSIONS

The closest work related to ours is [16]. These authors also observe that standard symbolic execution does not track control dependences and therefore forces loops to execute a fixed number of iterations as in a concrete execution. For each program loop, they define a *trip count* as a symbolic variable which represents the number of times the loop is executed. They then propose to run a separate static abstract-interpretation-based analysis to determine linear relations between program variables and trip counts. Trip counts are themselves related to the input using a grammar describing the input format and supplied to the analysis. In contrast, our approach is simpler: linear relations among induction variables are inferred dynamically during a single dynamic symbolic execution, made explicit by symbolic store updates at loop summarization, and then propagated forward by regular symbolic execution. Also, we infer loop counts based on simple pattern matching, and do not require an input grammar. Our method however may miss loop counts in cases of loops over delimited fields in the input, which can be handled in [16] thanks to the input grammar; combining our technique with the orthogonal *length abstraction* technique of [19] could remove this limitation while still not relying on any input grammar. Another difference is that loop structures are defined/detected in [16] using static (binary) analysis. Instead, we build the program's loop structure on the fly, without additional tools for static analysis. The price we pay is that sometimes we may lose one iteration before detecting a loop, and that loop exits are harder to detect (see Section 5.1). The computation overhead introduced by our technique is reasonable, about 50% runtime and memory, and we believe that optimizations can reduce it. Similar measurements are not available from [16] which does not present results of controlled experiments hightlighting the specific contribution of the loop treatment (i.e., turned on versus off) in a dynamic test generation tool. We use a context stack to handle arbitrary nested loops and recursive functions; such cases are not discussed in [16] addresses such cases. Finally, [16] does not define when (i.e., under which assumptions) their approach is sound and/or complete.

In program verification using verification-condition generation, a static program analysis generates a single logic formula representing the entire program (e.g., [4]). This formula typically uses one symbolic variable for each program variable, and captures all control and data dependencies. This approach also typically requires the user to provide a loop invariant for each program loop, as well as pre and postconditions for individual functions. This in turns allows for modular program reasoning. In contrast, dynamic test generation [11, 6] generates logic formulas representing individual whole-program paths one at a time. This allows logic encodings of very long program executions [12], but suffers from path explosion since many paths need be considered. Compositional dynamic test generation [9, 1, 13] provide a practical trade-off between these two extreme logic program representations: intuitively, sets of (say intraprocedural) sub-paths can be bundled together in logic program summaries using disjunctions of (intraprocedural) sub-path constraints, and injected in regular (interprocedural) whole-program path constraints. Prior work on summarization in dynamic test generation describes algorithms for memoizing sub-path constraints, for incrementally bundling them into logic summaries, and for hierarchical search space exploration, but does not prescribe any specific procedure for dealing with loops, unlike our loop generalization and summarization which can encode in one logic formula possibly infinitely many loop executions.

Automatic loop invariant generation and summarization has been discussed in numerous papers in the context of static program analysis, including for infinite-state model checking (e.g., [5]), predicate abstraction (e.g., [3]), and termination analysis (e.g., [18]), to name a few. In comparison, the main originality of our work is that it is based on detecting loop structures, induction variables, IV-dependent guards and linear relationships between those in a *completely dynamic way* – our algorithm does not require any static program analysis. We are not aware of any other entirely-dynamic loop-invariant generation algorithm.

Over the last few years, dynamic test generation has been extended in various ways and applied to other application domains (e.g., [8, 17, 2, 15] among others). However, except for [16] and this paper, we are not aware of any other paper specifically focused on how to deal with input-dependent loops in dynamic test generation.

Acknowledgments. We thank Sumit Gulwani, Francesco Logozzo and Cindy Rubio-Gonzalez for helpful comments on preliminary versions of this work.

8. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proceedings of TACAS'2008 (14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381, Budapest, April 2008. Springer-Verlag.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Software Eng.*, 36(4):474–494, 2010.

[3] T. Ball, O. Kupferman, and M. Sagiv. Leaping Loops in the Presence of Abstraction. In *Proceedings of CAV'2007 (19th Conference on Computer Aided Verification)*, Berlin, July 2007.

[4] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO'2005 (4th International Symposium on Formal Methods for Components and Objects)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer-Verlag, September 2006.

[5] B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs. In *Proceedings of CAV'96 (8th Conference on Computer Aided Verification)*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12, New Brunswick, August 1996. Springer-Verlag.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.

[7] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS'2008 (14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, April 2008. Springer-Verlag.

[8] M. Emmi, R. Majumdar, and K. Sen. Dynamic Test Input Generation for Database Applications. In *Proceedings of ISSTA'2007 (International Symposium on Software Testing and Analysis*, pages 151–162, 2007.

[9] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages)*, pages 47–54, Nice, January 2007.

[10] P. Godefroid. Software Model Checking Improving Security of a Billion Computers. In *Proceedings of SPIN'2009 (16th International SPIN Workshop on Model Checking of Software)*, volume 5578 of *Lecture Notes in Computer Science*, page 1, Grenoble, June 2009. Springer-Verlag.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)*, pages 213–223, Chicago, June 2005.

[12] P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, pages 151–166, San Diego, February 2008.

[13] P. Godefroid, A.V. Nori, S.K. Rajamani, and S.D. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proceedings of POPL'2010 (37th ACM Symposium on Principles of Programming Languages)*, pages 43–55, Madrid, January 2010.

[14] 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*, Aug 2009.

[15] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[16] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-Extended Symbolic Execution on Binary Programs. In *Proceedings of ISSTA'2009 (International Symposium on Software Testing and Analysis*, pages 225–236, Chicago, July 2009.

[17] N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of TAP'2008 (2nd International Conference on Tests and Proofs)*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer-Verlag, April 2008.

[18] A. Tsitovich, N. Sharygina, Ch. Wintersteiger, and D. Kroening. Loop Summarization and Termination Analysis. In *To appear in Proceedings of TACAS'2011 (17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, April 2011.

[19] R. Xu, , P. Godefroid, and R. Majumdar. Testing for Buffer Overflows with Length Abstraction. In *Proceedings of ISSTA'2008 (International Symposium on Software Testing and Analysis)*, pages 27–38, Seattle, July 2008.