IC-Cut: A Compositional Search Strategy for Dynamic Test Generation

Maria Christakis[1]* and Patrice Godefroid[2]

[1] Department of Computer Science
ETH Zurich, Switzerland
maria.christakis@inf.ethz.ch
[2] Microsoft Research
Redmond, USA
pg@microsoft.com

Abstract. We present IC-Cut, short for "Interface-Complexity-based Cut", a new compositional search strategy for systematically testing large programs. IC-Cut dynamically detects function interfaces that are simple enough to be cost-effective for summarization. IC-Cut then hierarchically decomposes the program into units defined by such functions and their sub-functions in the call graph. These units are tested independently, their test results are recorded as low-complexity function summaries, and the summaries are reused when testing higher-level functions in the call graph, thus limiting overall path explosion. When the decomposed units are tested exhaustively, they constitute verified components of the program. IC-Cut is run dynamically and on-the-fly during the search, typically refining cuts as the search advances.
We have implemented this algorithm as a new search strategy in the whitebox fuzzer SAGE, and present detailed experimental results obtained when fuzzing the ANI Windows image parser. Our results show that IC-Cut alleviates path explosion while preserving or even increasing code coverage and bug finding, compared to the current generational-search strategy used in SAGE.

1 Introduction

Systematic dynamic test generation [13, 7] consists of symbolically executing a program dynamically, while collecting constraints on inputs from branch statements along the execution. These constraints are systematically negated and solved with a constraint solver to infer variants of the previous inputs, which will exercise alternative execution paths of the program. The process is systematically repeated with the goal of exploring the entire set (in practice, a subset) of all feasible execution paths of the program. This approach to automatic test case generation has been implemented in many popular tools over the last decade, such as EXE [8], jCUTE [20], Pex [22], KLEE [6], BitBlaze [21], and Apollo [2], to name a few. Although effective in detecting bugs, these testing tools have never

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

been pushed toward program verification of a large and complex application, i.e., toward proving that the application is free of certain classes of errors.

We have recently used the whitebox fuzzer SAGE [15] to show how systematic dynamic test generation can be extended toward program verification of the ANI Windows image parser [10]. In this previous work, we limit path explosion in the parser with user-guided program decomposition and summarization [11, 1]. In particular, we *manually* identify functions for summarization whose input/output interfaces with respect to higher-level functions in the call graph are not too complex, so that the logic encoding of their summaries remains simple. Indeed, we find that it is common for functions to return a single "success" or "failure" value. If "failure" is returned, the higher-level function typically terminates. If "success" is returned, parsing proceeds with new chunks of the input, that is, completely independently of the specific path taken in the function being summarized. We, therefore, decompose the program at very few interfaces, of functions that parse independent chunks of the input and return a single "success" or "failure" value.

Based on these previous insights, we now define a new compositional search strategy for *automatically* and dynamically discovering simple function interfaces, where large programs can be effectively decomposed. IC-Cut, short for "Interface-Complexity-based Cut", tests the decomposed program units independently, records their test results as low-complexity function summaries (that is, summaries with simple logic encoding), and reuses these summaries when testing higher-level functions in the call graph, thus limiting overall path explosion. IC-Cut runs on-the-fly during the search to incrementally refine interface cuts as the search advances. In short, IC-Cut is inspired by compositional reasoning, but is only a search strategy, based on heuristics, for decomposing the program into independent units that process different chunks of the input. We, therefore, do not perform compositional verification in this work, except when certain particular restrictions are met (see Sects. 3.4 and 4).

The main contributions of this paper are:

– We present an attractive and principled alternative to ad-hoc state-of-the-art search heuristics for alleviating path explosion.
– As our experiments show, IC-Cut preserves or even increases code coverage and bug finding in significantly less time, compared to the current generational-search strategy of SAGE.
– IC-Cut can identify which decomposed program units are exhaustively tested and, thus, *dynamically verified*.

This paper is organized as follows. In Sect. 2, we recall basic principles of systematic dynamic test generation and whitebox fuzzing, and give an overview of the SAGE tool used in this work. Sect. 3 explains the IC-Cut search strategy in detail. In Sect. 4, we present our experimental results obtained when fuzzing the ANI Windows image parser. We review related work in Sect. 5 and conclude in Sect. 6.

2 Background

We consider a sequential deterministic program P, which is composed of a set of functions and takes as input an input vector, that is, multiple input values. The determinism of the program guarantees that running P with the same input vector leads to the same program execution.

We can systematically explore the state space of program P using *systematic dynamic test generation* [13, 7]. Systematic dynamic test generation consists of repeatedly running a program both concretely and symbolically. The goal is to collect symbolic constraints on inputs, from predicates in branch statements along the execution, and then to infer variants of the previous inputs, using a constraint solver, in order to steer the next execution of the program toward an alternative program path.

Symbolic execution means executing a program with symbolic rather than concrete values. A symbolic variable is, therefore, associated with each value in the input vector, and every constraint is on such symbolic variables. Assignment statements are represented as functions of their (symbolic) arguments, while conditional statements are expressed as constraints on symbolic values. 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. For a program path w, a *path constraint* ϕ_w is a logic formula that characterizes the input values for which the program executes along w. Each symbolic variable appearing in ϕ_w is, thus, a program input. Each constraint is expressed in some theory[3] T decided by a constraint solver, i.e., an automated theorem prover that can return a satisfying assignment for all variables appearing in constraints it proves satisfiable.

Whitebox fuzzing is an application of systematic dynamic test generation for detecting security vulnerabilities. In particular, *whitebox file fuzzing* explores programs that take as input a file, all bytes of which constitute the input vector of the program. SAGE [15] is a whitebox file fuzzing tool for security testing, which implements systematic dynamic test generation and performs dynamic symbolic execution at the x86 binary level. It is optimized to scale to very large execution traces (billions of x86 instructions) and programs (like Excel). Notably, SAGE is credited to have found roughly one third of all the security bugs discovered by file fuzzing during the development of Microsoft's Windows 7 [5].

Obviously, testing and symbolically executing *all* feasible program paths is not possible for large programs. Indeed, the number of feasible paths can be exponential in the program size, or even infinite in the presence of loops with an unbounded number of iterations. In practice, this *path explosion* is alleviated using *heuristics* to maximize code coverage as quickly as possible and find bugs faster in an incomplete search. For instance, SAGE uses a *generational-search strategy* [15], where all constraints in a path constraint are negated one by one in order to maximize the number of new tests generated per symbolic execution.

[3] A theory is a set of logic formulas.

Alg. 1 The IC-Cut search algorithm.

```
function IC-Cut(p, seeds)
  cg ← CreateCallgraph(p, seeds)
  summaries ← {}
  Explore(cg, p, summaries)

function Explore(cg, p, summaries)
  workQueue ← GetLeaves(cg)
  while IsNotEmpty(workQueue) do
    f ← Peek(workQueue)
    cg′, summaries ← Process(f, p, summaries)
    if cg′ == cg then
      workQueue ← Dequeue(workQueue)
      predecessors ← GetPredecessors(f, cg)
      workQueue ← Enqueue(predecessors, workQueue)
    else
      newFunctions ← GetNewFunctions(cg, cg′)
      workQueue ← AddFirst(newFunctions, workQueue)
      cg ← cg′

function Process(f, p, summaries)
  seed ← GetSeed(f)
  interface, cg′ ← Fuzz(f, p, seed, summaries)
  if IsSummarizable(interface) then
    summary ← GenerateSummary(interface)
    summaries ← PutSummary(f, summary, summaries)
  return cg′, summaries
```

This search strategy is combined with simple heuristics that guide the search toward least covered parts of the search space and prune the search space using *flip count limits* and *constraint subsumption* (see Sects. 3.3 and 4). Other related industrial-strength tools like Pex [22] use similar techniques. In this paper, we explore a different approach to alleviate path explosion.

3 The IC-Cut search strategy

In this section, we present the IC-Cut search algorithm, precisely define the low-complexity function summaries of IC-Cut, and discuss its correctness guarantees and limitations.

3.1 Algorithm

Alg. 1 presents the IC-Cut search strategy. IC-Cut consists of three phases, which are overlapping: learning, decomposition, and matching.

Learning The learning phase of IC-Cut runs the program under test on a *training set* of seed inputs. The goal is to learn as much of the call graph of the program. As a result, the larger the training set, the more detailed is the global view that IC-Cut has of the program, and the fewer new functions are discovered in the next phase.

On line 2 of Alg. 1, function CreateCallgraph returns the call graph of the program that is learned, dynamically and incrementally, by running the program on the seed inputs. Each node in the call graph represents a function of the program, and contains the function name and one seed input that steers execution of the program through this function. Each edge (f, g) in the call graph denotes that function f calls function g. Note that we assume no recursion.

Handling recursion is conceptually possible [11]. In practice, it is not required for the application domain of binary image parsers. Recursion in such parsers is very rare due to obvious performance, scalability, and reliability reasons, which is why we do not address it in this work.

Decomposition During the decomposition phase, IC-Cut fuzzes (that is, explores using dynamic symbolic execution) one function at a time, starting at the bottom of the learned call graph, and potentially records the function test results as a low-complexity summary (that is, a summary with a simple logic encoding, as defined in Sect. 3.2). This is done in function Explore of Alg. 1, which is called on line 4 and takes as arguments the call graph *cg*, the program under test *p*, and an empty map from call-graph nodes to function summaries *summaries*.

In particular, IC-Cut selects a function from the bottom of the call graph that has not been previously fuzzed. This is shown on line 7 of Alg. 1, in function Explore, where we create a *workQueue* of the call graph leaf-nodes, and on line 9, where a function f is selected from the front of the *workQueue*. The selected function is then tested independently (in function Process) to determine whether its interface is simple enough to be cost-effective for summarization. To test the selected function, IC-Cut chooses an appropriate seed input, which in the previous phase has been found to steer execution of the program through this function (line 21 of Alg. 1). Subsequently, on line 22, IC-Cut fuzzes the program with this seed input.

However, while fuzzing the program, not all symbolic constraints that IC-Cut collects may be negated; we call the constraints that may be negated *open*, and all others *closed*. Specifically, the constraints that are collected until execution encounters the *first* call to the selected function are closed. Once the function is called, the constraints that are collected until the function returns are open. As soon as the function returns, symbolic execution terminates. This means that IC-Cut fuzzes only the selected function and for a single calling context of the program. Note that the function is fuzzed using a *generational search*.

While fuzzing the selected function, IC-Cut dynamically determines the complexity of its interface, as defined in Sect. 3.2. If the function interface is simple enough to be cost-effective for summarization (line 23 of Alg. 1), the test results of the function are recorded as a summary. On line 24, we generate the function summary, and on line 25, we add it to the *summaries* map. Note that function PROCESS describes our algorithm in a simplified way. If a function interface is found to be suitable for summarization, IC-Cut actually records the summary *while* fuzzing the function. If this is not the case, IC-Cut aborts fuzzing of this function. How summaries are generated is precisely documented in Sect. 3.2.

It is possible that new functions are discovered during fuzzing of the selected function, i.e., functions that do not appear in the call graph of the learning phase. When this happens, IC-Cut updates the call graph. Of course, these new functions are placed lower in the call graph than the currently-fuzzed function, which is their (direct or indirect) caller. IC-Cut then selects a function to fuzz from the bottom of the updated call graph.

This is shown on lines 11–18 of Alg. 1. If no new functions are discovered during fuzzing of the selected function (line 11), we remove this function from the *workQueue*, and add its predecessors in the call graph at the end of the *workQueue* (lines 12–14). When IC-Cut explores these predecessors, their callees will have already been fuzzed. If, however, new functions are discovered (lines 15–16), we add these functions at the front of the *workQueue* (line 17), and update the call graph (line 18). Note that when new functions are discovered, IC-Cut aborts exploration of the currently-fuzzed function; this is why this function is not removed from the *workQueue* on line 17.

The above process highlights the importance of the training set of seed inputs in the learning phase: the better the training set of seed inputs is in call-graph coverage, the less time is spent on switches between the decomposition and learning phases of IC-Cut.

Matching In general, summaries can be reused by callers to skip symbolic execution of a summarized callee and, hence, alleviate path explosion caused by inlining the callee, i.e., by re-exploring all callee paths.

The matching phase decides whether a recorded summary may be reused when testing higher-level functions in the call graph. This is why function FUZZ of Alg. 1 (line 22) takes the *summaries* map as argument. On the whole, FUZZ explores (using dynamic symbolic execution) one function at a time, records its interface, and reuses previously-computed summaries.

In our context, while fuzzing a higher-level function in the decomposition phase, the exploration might come across a call to a function for which a summary has already been computed. Note, however, that this summary has been computed for a particular calling context. Therefore, the matching phase determines whether the encountered calling context of the function matches (precisely defined in Sect. 3.2) the old calling context for which the summary has been computed. If this is the case, it is guaranteed that all execution paths of the function for the encountered calling context are described by the recorded summary. Consequently, the summary may be reused, since no execution paths of the function will be missed. If, on the other hard, the calling contexts do not match, the called function is fuzzed as part of the higher-level function (that is, it is inlined to the higher-level function) as if no summary had been recorded, to avoid missing execution paths or generating false alarms. In other words, IC-Cut allows that a function is summarized only for a single calling context, and summary reuse must be calling-context specific.

3.2 Function summaries

Before describing which constraints on interface complexity a function must satisfy to be summarized, we first precisely define function inputs and outputs.

Function inputs and outputs

– An *input* i_f of function f is any value that is read and tested by f. In other words, the value of i_f is not only read in f, but also affects which execution path of the function is taken at runtime.
– An input i_f of f is *symbolic* if it is a function of any whole-program inputs; otherwise, i_f is *concrete*.
– A *candidate output* co_f of function f is any value that is written by f.
– An *output* o_f of function f is any candidate output of f that is tested later in the program.

Consider program `P` below, which expects two non-negative inputs `a` and `b`:

```
int is_less(int x, int y) {
  if (x < y)
    return 1;
  return 0;
}

void P(int a, int b) {
  if (is_less(a, 0) || is_less(b, 0))
    error();
  ...
}
```

For both calling contexts of function `is_less` in program `P`, `is_less` has one symbolic input (that is, `a` or `b`), one concrete input (that is, `0`), and one output (which is `0` or `1` and tested by the if-statement in `P`).

Generating summaries In compositional symbolic execution [11, 1], a summary ϕ_f for a function f is defined as a logic formula over constraints expressed in a theory T. Summary ϕ_f may be computed by symbolically executing all paths of function f, generating an input precondition and output postcondition for each path, and gathering all of these path summaries in a disjunction.

Precisely, ϕ_f is 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 intra-procedural path in f, pre_{w_f} is a conjunction of constraints on the inputs of f, and $post_{w_f}$ a conjunction of constraints on the outputs of f. For instance, a summary ϕ_f for function `is_less` is

$$\phi_f = (x < y \land ret = 1) \lor (x \geq y \land ret = 0)$$

where *ret* denotes the value returned by the function. This summary may be reused across different calling contexts of `is_less`. In practice, however, these disjunctions of conjunctions of constraints can become very large and complex, thus making summaries expensive to compute. For this reason, IC-Cut generates only low-complexity function summaries for specific calling contexts.

For a given calling context, a function f is summarized by IC-Cut only if the following two conditions are satisfied:

– All symbolic inputs of f are unconstrained, that is, they are completely independent of the execution path taken in the program until function f is called. In particular, the symbolic inputs of f do not appear in any of the closed constraints collected before the call to f. Therefore, the input precondition of f must be true.
– Function f has at most one output o_f.

If the above conditions are not satisfied, function f is inlined to its calling contexts (that is, not summarized). As an example, consider again program `P`. For the first calling context of function `is_less` in `P` (that is, `is_less(a, 0)`), the symbolic input of `is_less` is unconstrained, and the function has exactly one output. As a result, `is_less` is summarized by IC-Cut for this *first* calling context, as described in Sect. 3.1.

As a consequence of these conditions, the summaries considered in this work have a single precondition on all symbolic inputs, which is true, and a single precondition on all concrete inputs, which is of the form

$$\bigwedge_{0 \leq j < N} i_j = c_j$$

where i_j is a concrete input, c_j a constant representing its concrete value, and N the number of concrete inputs. Moreover, the summaries in this work have no output postconditions, as explained later in this section. As a result, when IC-Cut generates a summary for a function f, it actually records a precondition of the above form on all concrete inputs of f; this precondition also represents the current calling context of f. In this paper, we abuse terminology and call such preconditions "summaries", although we do not record any disjunctions or postconditions. For example, in the program `P` above, IC-Cut generates the following summary for the first calling context of function `is_less`

$$y = 0$$

which denotes that all inputs of `is_less` except for y are symbolic and unconstrained, and that y is a concrete input whose value is 0 in the particular calling context. This summary indicates that function `is_less` has been fuzzed for a calling context in which x may take any value, while y must have the value 0.

Reusing summaries While fuzzing a higher-level function in the decomposition phase of IC-Cut, the exploration might come across a call to a function for which a summary has already been generated. Then, the matching phase determines if this summary may be reused by checking whether the new calling context of the function matches, i.e., is equally or more specific than, the old calling context for which the summary has been recorded (see Sect. 3.1).

- The new calling context is *as specific* as the old calling context only if (1) the function inputs that are symbolic and unconstrained in the old calling context are also symbolic and unconstrained in the new calling context, and (2) all other function inputs are concrete and have the same values across both calling contexts, except in the case of non-null *pointers* whose concrete values may differ since dynamic memory allocation is nondeterministic (see Sect. 3.4 for more details).
- The new calling context is *more specific* than the old calling context only if (1) the function inputs that are concrete in the old calling context are also concrete in the new calling context and have the same values (except in the case of non-null pointers), and (2) one or more function inputs that are symbolic and unconstrained in the old calling context are either symbolic and constrained in the new calling context or they are concrete.

Recall that, in our previous example about program `P`, IC-Cut records a summary for the first calling context of function `is_less` in `P`. This summary is then reused in the second calling context of `is_less` in `P` (that is, `is_less(b, 0)`), which is as specific as the first.

After having described *when* a recorded summary may be reused, we now explain *how* this is done. When the matching phase of IC-Cut determines that a function summary matches a calling context of the function, the following two steps are performed:

1. The function is executed only concretely, and not symbolically, until it returns.
2. The function candidate outputs are associated with fresh symbolic variables.

Step (1) is performed because all execution paths of the function have already been explored when testing this function independently for an equally or more general calling context. Step (2) is used to determine whether the function has at most one output, as follows.

When testing a function f for a given calling context, we can determine all values that are written by f, which we call *candidate outputs*. Yet, we do not know whether these candidate outputs are tested later in the program, which would make them *outputs* of f. Therefore, when reusing a summary of f, we associate fresh symbolic variables with all of its candidate outputs. We expect that at most one of these candidate outputs is ever tested later in the program. If this condition is not satisfied, the summary of f is invalidated. In this case, the higher-level function that reused the summary of f is tested again, but this time, f is inlined to its calling contexts instead of summarized.

When reusing the summary of function `is_less` in program `P`, we associate a symbolic variable with the function's only candidate output, its return value. This symbolic variable is tested by function `P`, in the condition of the if-statement, thus characterizing the return value of `is_less` as a function output.

3.3 Input-dependent loops

We use constraint subsumption [15] to automatically detect and control input-dependent loops. Subsumption keeps track of the constraints generated from a given branch instruction. When a new constraint c is generated, SAGE uses a fast syntactic check to determine whether c implies or is implied by a previous constraint, generated from the same instruction during the execution, most likely due to successive iterations of an input-dependent loop. If this is the case, the weaker (implied) constraint is removed from the path constraint.

In combination with subsumption, which eliminates the weaker constraints generated from the same branch, we can also use *constraint skipping*, which never negates the remaining stronger constraints injected at this branch. When constraint subsumption and skipping are both turned on, an input-dependent loop is concretized, that is, it is explored only for a fixed number of iterations.

3.4 Correctness

We now discuss the correctness guarantees of the IC-Cut search strategy. The following theorems hold assuming symbolic execution has perfect precision, i.e., that constraint generation and solving are sound and complete for all program instructions.

We define an *abort-statement* in a program as any statement that triggers a program error.

Theorem 1. *(Soundness) Consider a program P. If IC-Cut reaches an abort, then there is some input to P that leads to an abort.*

Proof sketch. The proof is immediate by the soundness of dynamic symbolic execution [13, 11]. In particular, it is required that the summaries of IC-Cut are not over-approximated, but since these summaries are computed using dynamic symbolic execution, this is guaranteed. □

Theorem 2. *(Completeness) Consider a program P. If IC-Cut terminates without reaching an abort, no constraints are subsumed or skipped, and the functions whose summaries are reused have no outputs and no concrete non-null pointers as inputs, then there is no input to P that leads to an abort.*

Proof sketch. The proof rests on the assumption that any potential source of incompleteness in the IC-Cut summarization strategy is conservatively detected. There are exactly two sources of incompleteness: (1) constraint subsumption and skipping for automatically detecting and controlling input-dependent loops, and

(2) reusing summaries of functions that have a single output and concrete non-null pointers as inputs.

Constraint subsumption and skipping remove or ignore non-redundant constraints from the path constraint to detect and control successive iterations of input-dependent loops. By removing or ignoring such constraints, these techniques omit certain execution paths of the program, and are therefore incomplete.

When reusing the summary of a function with a single output, certain execution paths of the program might become infeasible due to the value of its output. As a result, IC-Cut might fail to explore some execution paths. On the other hand, summaries of functions with no outputs are completely independent of the execution paths taken in the program. Therefore, when such summaries are reused, no paths are ever missed. Note that by restricting the function outputs to at most one, we set an upper bound to the number of execution paths that can be missed, that is, in comparison to reusing summaries of functions with more than one output.

When reusing the summary of a function that has concrete non-null pointers as inputs, execution paths that are guarded by tests on the values of these pointers might be missed, for instance, when two such pointers are compared for aliasing. This is because we ignore whether the values of such inputs actually match the calling context where the summary is reused, to deal with the nondeterminism of dynamic memory allocation.

The program units for which the exploration of IC-Cut is complete are dynamically verified. □

3.5 Limitation: Search redundancies

It is worth emphasizing that IC-Cut may perform redundant sub-searches in two pathological cases: (1) incomplete call graph, and (2) late summary mismatch, as detailed below. However, as our evaluation shows (Sect. 4), these potential limitations seem outweighed by the benefits of IC-Cut in practice.

Incomplete call graph This refers to discovering functions during the decomposition phase of IC-Cut that do not appear in the call graph built in the learning phase. Whenever new functions are discovered, fuzzing is aborted in order to update the call graph, and all test results of the function being fuzzed are lost.

Late summary mismatch Consider a scenario in which function `foo` calls function `bar`. At time t, `bar` is summarized because it is call-stack deeper than `foo` and the interface constraint on `bar`'s inputs is satisfied. At time $t + i$, `foo` is explored while reusing the summary for `bar`, and `bar`'s candidate outputs are associated with symbolic variables. At time $t + i + j$, while still exploring `foo`, the interface constraint on `bar`'s outputs is violated, and thus, the summary of `bar` is invalidated. Consequently, fuzzing of `foo` is aborted and restarted, this time by inlining `bar` to its calling context in `foo`.

4 Experimental evaluation

In this section, we present detailed experimental results obtained when fuzzing the ANI Windows image parser, which is available on every version of Windows.

This parser processes structured graphics files to display "ANImated" cursors and icons, like the spinning ring or hourglass on Windows. The ANI parser is written mostly in C, while the remaining code is written in x86 assembly. It is a large benchmark consisting of thousands of lines of code spread across hundreds of functions. The implementation involves at least 350 functions defined in five Windows DLLs. The parsing of input bytes from an ANI file takes place in at least 110 functions defined in two DLLs, namely, in `user32.dll`, which is responsible for 80% of the parsing code, and in `gdi32.dll`, which is responsible for the remaining 20% [10].

Our results show that IC-Cut alleviates path explosion in this parser while preserving or even increasing code coverage and bug finding, compared to the current generational-search strategy used in SAGE. Note that by "generational-search strategy used in SAGE", we mean a monolithic search in the state space of the entire program.

For our experiments, we used five different configurations of IC-Cut, which we compared to the generational-search strategy that is implemented in SAGE. All configurations are shown in Tab. 1. For each configuration, the first column of the table shows its identifier and whether it uses IC-Cut. Note that configurations A–E use IC-Cut, while F uses the generational-search strategy of SAGE. The second column shows the maximum runtime for each configuration: configurations A–E allow for a maximum of three hours to explore each function of the parser, while F allows for a total of 48 hours to explore the entire parser. The four rightmost columns of the table indicate whether the following options are turned on:

– *Summarization at maximum runtime*: Records a summary for the currently-fuzzed function when the maximum runtime is exceeded if no conditions on the function's interface complexity have been violated;

Configuration		**Maximum runtime**	**Summarization at maximum runtime**	**Constraint subsumption**	**Constraint skipping**	**Flip count limit**
ID	IC-Cut					
A	✓	3h/function		✓	✓	
B	✓	3h/function		✓	✓	✓
C	✓	3h/function	✓	✓	✓	
D	✓	3h/function		✓		
E	✓	3h/function	✓	✓		
F		48h		✓		✓

Tab. 1: All configurations used in our experiments; we used five different configurations of IC-Cut (A–E), which we compared to the generational-search strategy of SAGE (F).

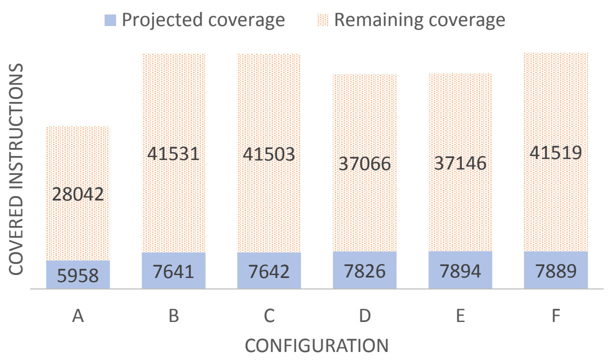

Fig. 1: The instructions of the ANI parser that are covered by each configuration. The projected instruction coverage is critical for bug finding.

- *Constraint subsumption*: Eliminates weaker constraints implied by stronger constraints generated from the same branch instruction, most likely due to successive iterations of an input-dependent loop (see Sect. 3.3);
- *Constraint skipping*: Does not negate stronger constraints that imply weaker constraints generated from the same branch instruction (see Sect. 3.3);
- *Flip count limit*: Establishes the maximum number of times that a constraint generated from a particular program instruction may be negated [15].

Note that F is the configuration of SAGE that is currently used in production.

Fig. 1 shows the instructions of the ANI parser that are covered by each configuration. We partition the covered instructions in those that are found in `user32.dll` and `gdi32.dll` (projected coverage), and those that are found in the other three DLLs (remaining coverage). Note that the instructions in `user32.dll` and `gdi32.dll` are responsible for parsing *untrusted* bytes and are, therefore, critical for bug finding. As shown in Fig. 1, configuration E, for which options “summarization at maximum runtime” and “constraint subsumption” are turned on, achieves the highest projected coverage. Configuration D, for which only “constraint subsumption” is turned on, achieves a slightly lower coverage. This suggests that summarizing when the maximum runtime is exceeded helps in guiding the search toward new program instructions; in particular, it avoids repeatedly exploring the code of the summarized functions. In contrast, configurations A–C, for which “constraint skipping” is turned on, achieve the lowest projected coverage. This indicates that testing input-dependent loops for more than just a single number of iterations is critical in increasing code coverage.

Fig. 2 shows the time (in minutes) it takes for each configuration to stop exploring the ANI parser. Note that configuration B stops in the smallest amount of time (approximately 15 hours); this is because too many constraints are pruned due to options “constraint subsumption”, “constraint skipping”, and “flip count limit”, which are turned on. Configuration E, which achieves the highest projected coverage (Fig. 1), stops exploring the parser in the second smallest

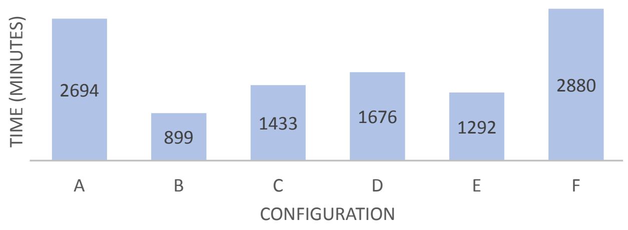

Fig. 2: The time it takes for each configuration to stop exploring the ANI parser.

amount of time, that is, in approximately 21.5 hours—roughly 55% faster than the generational-search strategy used in production (configuration F).

In this amount of time, configuration E also detects the largest number of unique first-chance exceptions in the ANI parser. This is shown in Fig. 3, which presents how many unique exceptions are detected by each configuration. A first-chance exception is an exception (similar to an assertion violation) thrown at runtime (by the operating system) during program execution, but caught by the program using a C/C++ try/catch-mechanism (see [10]). Note that the nine exceptions found by configuration E are a superset of all other exceptions detected by the remaining configurations.

In summary, configuration E detects more unique exceptions than all other configurations combined. Compared to configuration F (generational search), E finds more exceptions (Fig. 3) and achieves the same projected instruction coverage (Fig. 1) in less than half the time (Fig. 2). E is the most effective configuration against path explosion.

Tab. 2 shows how the winner-configuration E performs when the maximum runtime per function of the parser is one minute, 90 minutes, and three hours, respectively. Performance is measured in terms of covered instructions, total exploration time of the parser, and detected first-chance exceptions. As shown in the table, IC-Cut performs better than configuration F even for a maximum runtime of 90 minutes per function: there is a noticeable improvement in projected code coverage and bug finding, which is achieved in approximately eleven hours (roughly 76% faster than configuration F). This is a strong indication of how much the summarization strategy of IC-Cut can alleviate path explosion.

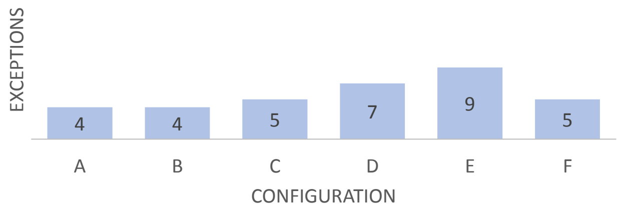

Fig. 3: The number of unique exceptions that are detected by each configuration.

Maximum runtime	Coverage		Total time	First-chance exceptions	
	projected	remaining	(in minutes)	unique	duplicate
1 minute	5,421	36,250	23	0	0
90 minutes	7,896	37,183	683	8	7
3 hours	7,894	37,146	1292	9	10

Tab. 2: Performance of the winner-configuration E when the maximum runtime per function of the parser is one minute, 90 minutes, and three hours, respectively. Performance is measured in terms of covered instructions, total exploration time of the parser, and detected first-chance exceptions.

Fig. 4 shows the number of functions that are explored by the winner-configuration E when the maximum runtime per function of the parser is one minute, 90 minutes, and three hours, respectively. This figure shows only functions for which SAGE generated symbolic constraints. The functions are grouped as follows: exhaustively tested and summarized, summarized despite constraint subsumption or an exceeded runtime, not summarized because of multiple outputs or constrained symbolic inputs. The functions in the first group constitute verified program components (according to Thm. 2), highlighting a *key originality of IC-Cut*, namely, that it can dynamically verify sub-parts of a program during fuzzing. As expected, the larger the maximum runtime, the more functions are discovered, the fewer functions are summarized at maximum runtime, and the more functions are verified. Interestingly, the functions that are not

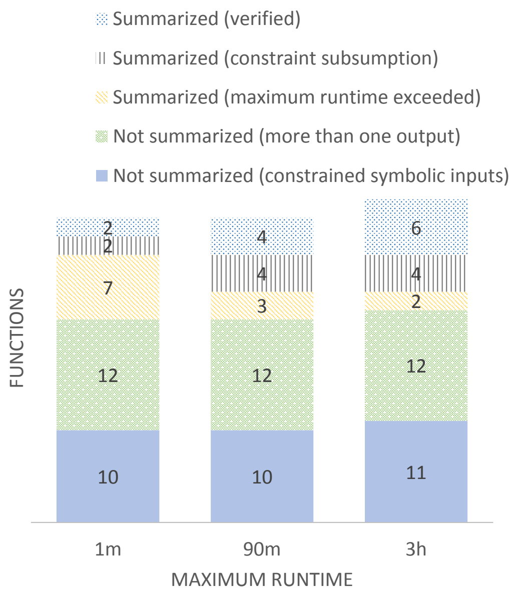

Fig. 4: How many functions are explored by the winner-configuration E when the maximum runtime per function of the parser is one minute, 90 minutes, and three hours, respectively. Only functions for which SAGE generated symbolic constraints are shown.

summarizable because of multiple outputs or constrained symbolic inputs are identified immediately, even for a maximum runtime of one minute per function.

We also used IC-Cut to fuzz other image parsers, namely, GIF and JPEG. Unfortunately, our prototype implementation could not handle the size of these larger parsers. However, preliminary experiments showed that our restrictions for summarization on function interfaces apply to both GIF and JPEG. For instance, when running on GIF with a time-out of three hours per function, 16 out of 140 functions (with symbolic constraints) were summarized. When running on JPEG with the same time-out, 27 out of 204 functions were summarized.

5 Related work

Automatic program decomposition for effective systematic dynamic test generation [9] is not a new idea. Moreover, compositional symbolic execution [11, 1] has already been shown to alleviate path explosion. However, when, where, and how compositionality is most effective in practice is still an open problem.

Algorithms for automatic program summarization have been proposed before [11, 1, 17]. SMART [11] tests all program functions in isolation, encodes their test results as summaries expressed using input preconditions and output postconditions, and then reuses these summaries when testing higher-level functions. Demand-driven compositional symbolic execution [1] generates incomplete summaries that describe only a subset of all paths in a function and can be expanded lazily. SMASH [17] computes both may and must information compositionally using both may and must summaries. IC-Cut is inspired by this compositional reasoning and summarization although it does not generate full-fledged function summaries. Instead, IC-Cut records a single precondition on all concrete function inputs without disjunctions or postconditions. In contrast to SMART, IC-Cut generates summaries only for functions with low interface complexity. Similarly to demand-driven compositional symbolic execution, our summaries are incomplete in that they describe a single calling context. Furthermore, when testing a function in isolation, the closed symbolic constraints that IC-Cut collects before the first call to the function are similar to the lazily-expanded dangling nodes in the demand-driven approach.

Other closely related techniques [18, 3, 4, 19] can be considered as approximations of sub-program summarization. Dynamic state merging and veritesting [18, 3] merge sub-program searches, and RWset [4] prunes searches by dynamically computing variable liveness. Information partitions [19] are used to identify "non-interfering" input chunks such that symbolically solving for each chunk while keeping all other chunks fixed to concrete values finds the same bugs as symbolically solving for the entire input. Similarly to these techniques, our work also approximates sub-program summarization. Moreover, IC-Cut is closely related to reducing test inputs using information partitions. Both techniques exploit independence between different parts of the program input. However, IC-Cut does not require that the input is initially partitioned, and avoids the overhead of dynamically computing data and control dependencies between input chunks.

Overall, our algorithm does not require any static analysis and uses very simple summaries, which are nevertheless sufficient to significantly alleviate path explosion. As a result, it is easy to implement on top of existing dynamic test generation tools. Our purely dynamic technique can also handle complicated ANI code patterns, such as stack-modifying, compiler-injected code for structured exception handling, and stack-guard protection, which most static analyses cannot handle. Furthermore, a static over-approximation of the call graph might result in testing more functions than necessary and for more calling contexts. With an over-approximation of function interfaces, we would summarize fewer functions, given the restrictions we impose on function inputs and outputs, thus fighting path explosion less effectively.

In addition to our low-complexity function summaries, SAGE implements other specialized forms of summaries, which deal with floating-point computations [12], handle input-dependent loops [16], and can be statically validated against code changes [14].

6 Concluding remarks

We have presented a new search strategy inspired by compositional reasoning at simple function interfaces. However, we do not perform compositional verification in this work, except when certain particular restrictions are met, as detailed in Thm. 2 (see also Sect. 4).

IC-Cut uses heuristics about interface complexity to discover, dynamically and incrementally, independent program units that process different chunks of the input vector. Our search strategy is sound for bug finding, while limiting path explosion in a more principled and effective manner than in the current implementation of SAGE, with its simple, yet clever, search heuristics. Indeed, compared to the generational-search strategy of SAGE, our experiments show that IC-Cut preserves code coverage and increases bug finding in significantly less exploration time.

IC-Cut generates low-complexity summaries for a single calling context of functions with unconstrained symbolic inputs and at most one output. Our previous work on proving memory safety of the ANI Windows image parser [10] shows that such simple interfaces exist in real, complex parsers, which is why we chose the above definition. However, our definition could be relaxed to allow for more than one calling context or function output, although our experiments show that this definition is already sufficient for large improvements. We leave this for future work.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008.
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. *TSE*, 36:474–494, 2010.

3. T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *ICSE*, pages 1083–1094. ACM, 2014.
4. P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS*, volume 4963 of *LNCS*, pages 351–366. Springer, 2008.
5. E. Bounimova, P. Godefroid, and D. A. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *ICSE*, pages 122–131. ACM, 2013.
6. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX, 2008.
7. C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, volume 3639 of *LNCS*, pages 2–23. Springer, 2005.
8. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335. ACM, 2006.
9. A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In *EMSOFT*, pages 262–271. ACM, 2006.
10. M. Christakis and P. Godefroid. Proving memory safety of the ANI Windows image parser using compositional exhaustive testing. In *VMCAI*, volume 8931 of *LNCS*, pages 370–389. Springer, 2015.
11. P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54. ACM, 2007.
12. P. Godefroid and J. Kinder. Proving memory safety of floating-point computations by combining static and dynamic program analysis. In *ISSTA*, pages 1–12. ACM, 2010.
13. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.
14. P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, volume 6887 of *LNCS*, pages 112–128. Springer, 2011.
15. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151–166. The Internet Society, 2008.
16. P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA*, pages 23–33. ACM, 2011.
17. P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, pages 43–56. ACM, 2010.
18. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, pages 193–204. ACM, 2012.
19. R. Majumdar and R.-G. Xu. Reducing test inputs using information partitions. In *CAV*, volume 5643 of *LNCS*, pages 555–569. Springer, 2009.
20. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.
21. D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, volume 5352 of *LNCS*, pages 1–25. Springer, 2008.
22. N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.