Trace Generalization via Loop Compression

Joxan Jaffar* and Vijayaraghavan Murali†
National University of Singapore
*joxan@comp.nus.edu.sg
† m.vijay@comp.nus.edu.sg

***Abstract*—We present a new method to generalize execution traces by compressing loop iterations in them using loop invariants. The invariants discovered are "safe" such that the resulting compressed trace also satisfies certain target properties which the original trace satisfied (e.g., an assertion at the end). This results in a concise trace that captures the semantics of the original trace w.r.t. the target but without needing to unroll the loops fully. A central feature is the use of a canonical loop invariant discovery algorithm which preserves all atomic formulas in the representation of a symbolic state which can be shown to be invariant. If this fails to provide a safe invariant, then the algorithm dynamically unrolls the loop and attempts the discovery at the next iteration, where it is more likely to succeed as the loop stabilizes towards an invariant. We show using realistic benchmarks that the end result, a generalization of the original trace, is significantly more succinct.**

I. Introduction

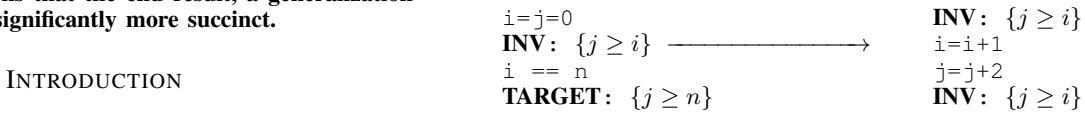
The analysis of execution traces has become an integral part of software engineering. A brief look at the past decade shows a large number of tools and techniques that were developed to reason about execution traces. Dynamic Symbolic Execution (DSE) [15], [33], [6], [5], [23] executes the program both concretely and symbolically, analyzes the resulting trace and generates inputs to execute the next trace that *deviates slightly from the current one*. Several fault localization techniques analyze error traces in particular to identify suspicious statements that may be the cause for the error. For instance, `explain` [16], [17] uses "distance metrics" to find another execution trace that is *as close as possible* to the failed trace, and uses the differences to explain the error. DARWIN [29], [30] tries to generate an alternate input that fails *in a similar way* as the given error trace, aligns them and generates a bug report from the differences. Delta debugging [9], [36], [37] extrapolates between failing and successful test cases to find *similar execution traces*.

A central theme in these techniques, as emphasized informally in the text, is to analyze an execution trace and explore its "neighborhood" of traces, formalized in [34]. In other words, their goal is to analyze a particular trace by *generalizing* it to a set of traces that exhibit similar behavior. In this paper, we propose a representation for such generalized execution traces that exhibit similar behavior, and a method to obtain the representation by compressing *loop iterations* in the trace using invariants. Consider as a motivating example the program below.

Running this program with input `n=100` would produce a trace that assigns `i` and `j` to 0, iterates through the loop

```
n=read() // assume non-negative
i=j=0
while (i != n) do
    i=i+1
    j=j+2
done
TARGET: {j ≥ n}
```

100 times, in the end assigning `i` and `j` to 100 and 200 respectively, thus implying the **TARGET**[1] that `j` $\geq$ `n`. In this contrived example though, any trace that executes the loop any number of times will imply the target in a similar way. We propose to capture this using the following *generalized trace*:



On the left is the actual generalized trace. It explains that initially the trace assigned `i` and `j` to 0. Then, on reaching the loop the *invariant* that $j \geq i$ held at the loop header at every iteration, and eventually the exit condition `i==n` was taken. These two pieces of information logically entail that `j`$\geq$`n`, the target. Note that this generalized trace is actually a collection of traces that imply the target after executing, in this case, 0 or more number of loop iterations.

In general, loops provide a natural and intuitive way to define similarity between traces. That is, two traces can be considered similar if they imply the same target at the end but just differ in the number of times they executed a loop. This occurs quite often in practice. For instance, a bug may be caused due to an input of size 100, which executed a loop 100 times. If the same bug is caused by an input of size 5, which executed the same loop 5 times, intuitively, they can be considered similar. A debugging programmer would indeed try to search for such a "similar" input that reproduces the bug.

Our generalized trace groups together such traces using a loop invariant. As a result, it exhibits semantically similar behavior as the original trace but without unrolling the loops fully. The most important benefit, as we show experimentally in Section VI, is that the generalized trace is much more concise than the original, and is therefore easier to analyze or comprehend. For this reason, we refer to the generalized trace as a "compressed trace". To motivate the application of a such a compressed trace, consider two scenarios.

[1]In general the target can be any formula that is implied by the trace. For an error trace, the target would be the negation of the violated assertion.

Scenario 1: A DSE search procedure is invoked on the above program, beginning with the (random) input `n=100`. To execute the next path, DSE would negate one of the 100 branches along this path and solve for the next input. Unfortunately, the unbounded loop (i.e., a loop whose iteration count is dependent on a non-deterministic input) makes the symbolic execution tree *infinite*, causing DSE to never terminate as it keeps exploring deeper and deeper iterations. However, the corresponding compressed trace in this case does not contain any loop iteration, but simply an invariant. In other words, there are *no branches* to negate, and so providing this to DSE instead can make it terminate. Of course, the DSE process has to be tweaked to understand compressed traces. For instance, the "branch" `i==n` is to be ignored as it is the exit condition of an already compressed loop.

Scenario 2: A programmer is trying to debug the program with the error trace obtained from `n=100` (assume the target is an error). To understand the trace the programmer needs to go through the 100 iterations. Note that techniques such as dynamic slicing [25] do not help, as no statement would be sliced away. The compressed trace on the other hand, is much easier to comprehend due to its succinctness and the loop invariant, which captures very closely the intuition that a human would have when analyzing the trace.

On the surface, this technique of compressing traces with loop invariants may sound quite simple. However, a number of challenges arise when delving deep into it:

A. What kind of invariants do we need to discover?

In the above example, all the following formulas are invariants for the loop: $i \geq 0$, $0 \leq j \leq 2n$, *true* (trivially an invariant for any loop), etc. But not all invariants can be used for this purpose – the invariant used in the compressed trace needs to *imply the target* in the end, similar to the original trace. Only if it does so, in which case we call it a "safe invariant", we can claim the compressed trace is a generalization of the original trace. For instance, the invariant $0 \leq j \leq 2n$ is not safe because combined with the loop's exit condition `i==n`, it does not imply the target $j \geq n$. Thus, discovering safe invariants is not trivial and is a primary challenge.

B. What if we are unable to discover a safe invariant?

In the above example, the invariant $j \geq i$ conveniently happened to be safe. But in many cases, a safe invariant could be very hard to discover. In our experiments in Section VI, we noticed several state-of-the-art invariant generators unable to find a safe invariant for our benchmarks. Thus, we need measures to handle such cases when the discovered invariant turns out to be unsafe. In our method, we dynamically *unroll* the loop and attempt the invariant discovery at the next iteration. This is a reasonable measure in practice as many loops converge towards an invariant as they iterate. For instance, loops typically either keep increasing or decreasing values of variables, and after a few unrolls we can discover lower or upper bounds on those variables, forming an invariant.

C. Are the discovered invariants relevant to the target?

When discovering invariants, we have the choice to aim for stronger or weaker invariants (logically). Weaker invariants are less likely to be safe because they abstract more information about the loop. Stronger invariants may carry too much information about the loop, many of which may be irrelevant to the target, making the generalization too specific. For instance, in the above example, $j \geq i \wedge n = 100$ is a stronger invariant than just $j \geq i$ but in fact, $n = 100$ is not needed to imply the target. Our method provides a fine balance between the two – first, we aim to discover strongest invariants in order to increase the likelihood of being safe, and once we reach the target along the trace, we *generalize* (weaken) the discovered invariants, removing information from them that were not really needed to imply the target.

D. Is there a proof of validity for our invariants?

The fact that the discovered formulas are indeed invariants comes from the soundness of our algorithm. However, in case this is challenged, we can provide a *proof of invariance* in the form of a Hoare-proof [19] consisting of Hoare-triples annotated at each statement in the loop. In the above example, to check if $j \geq i$ is indeed invariant, we can provide a Hoare-proof as shown on the right of the compressed trace. Starting with the "pre-condition" $j \geq i$, executing the statements in the loop body results in the "post-condition" $j \geq i$ again, thus the formula is invariant. This proof can be *checked* easily using automated proof assistants such as Coq [2]. If the proof fails, a bug in the implementation of our algorithm is detected.

The paper is organized as follows. Section II explains the basic idea of our algorithm with examples, Section III looks at some related work, Section IV establishes formal background for our method, Section V presents the main algorithm, and Section VI demonstrates our experimental evaluation.

II. Overview

Consider a more sophisticated example, the benchmark program `for_bounded_loop1.c`, taken from the Software Verification Competition 2013 (SV-COMP13) [4].

```
ℓ1  x=0, y=0, err=0 /* assume n > 0 */
ℓ2  for (i=0; i < n; i++) do
ℓ3      x = x-y
ℓ4      if (x != 0) err = 1 else skip
ℓ5      y = read() /* assume read does not return a 0 */
ℓ6      x = x+y
ℓ7      if (x == 0) err = 1 else skip
    done
ℓ8  TARGET: {x ≠ 0, err ≠ 1}
```

Assume that the program is executed with input `n=10` and `y`, read at ℓ_5, is never input as 0. This will produce a trace that iterates through the loop 10 times, and exits with `x` being non-zero and `err` being 0, implying the target $\{x \neq 0, err \neq 1\}$.

Our method performs *symbolic execution* on the trace, collecting the constraints into a *symbolic state* at each location. For instance, the symbolic state at ℓ_2 would be the set of constraints $\{x = 0, y = 0, err = 0, i = 0\}$. When a loop

is encountered, it attempts to discover an invariant. While technically any invariant generation algorithm can be used in this step, we describe a specific one suitable to our overall method. Briefly, we symbolically execute every path in the loop exercised by the trace, and delete individual constraints from the loop header's symbolic state that do not still hold at the end of each path. The whole process is repeated till no deletions are made, at which point the remaining constraints form the loop invariant. We call this technique *basic individually invariant discovery* (BIID).

BIID works well with our unrolling mechanism because the symbolic state at the loop header is very likely to change after unrolling, at which point BIID checks for invariance *considering the changes in the new state*. This is important, as unrolling may expose constraints that were not originally invariant before but are indeed invariant from now on. Hence it is a "basic" requirement to check if individual constraints in the new symbolic state are now invariant. More sophisticated methods may consider *disjunctions* of constraints, linear relationships between variables, user-provided predicates, etc., which may result in better (stronger) invariants. Although these techniques are admitted in our algorithm, we found that in practice BIID is fast and yet smart enough to capture safe invariants. Note that BIID is dependent on the internal representation of a symbolic state in choosing the "candidate" invariant constraints. For instance, the symbolic states $\{v = 5, w = v\}$ and $\{v = 5, w = 5\}$ are semantically equivalent, but BIID may return different invariants if, for e.g., $\{w = v\}$ is not invariant in the loop but $\{w = 5\}$ is. This depends on the implementation of the underlying symbolic execution engine.

Coming back to the example, let us see how BIID finds an invariant at ℓ_2. Since both `x` and `y` are zero at ℓ_2, `x=x-y` makes no change to the state, so we reach ℓ_4 with the state $\{x = 0, y = 0, err = 0, i = 0\}$. This makes the branch condition *false*, so we continue along the path to read `y`, which we assume is non-zero. Since `y` is now non-zero and `x` is zero, executing `x=x+y` at ℓ_6 results in the state $\{x = y, y \neq 0, err = 0, i = 0\}$ at ℓ_7. This makes the branch condition *false*, and we reach the looping point ℓ_2 again with the state $\{x = y, y \neq 0, err = 0, i = 1\}$.

Recall that the initial symbolic state at ℓ_2 was $\{x = 0, y = 0, err = 0, i = 0\}$. From this, BIID deletes constraints that do not still hold. $x = 0$ and $y = 0$ are deleted because both `x` and `y` have now become non-zero. Similarly, $i = 0$ is deleted because it has been incremented to 1. Hence, we get the new state $\{err = 0\}$ at ℓ_2. Since some deletions were made, we repeat this process again with the new state, but it can be seen that without constraints on `x` and `y`, the branches at ℓ_4 and ℓ_7 can also be evaluated to *true*, and so $err = 0$ cannot be invariant. Hence we delete it to get the invariant *true*, which fails to imply the target $\{x \neq 0, err \neq 1\}$ (i.e., it is unsafe).

This triggers our failsafe that basically discards the above work, *unrolls* the loop and attempts the same process at the next iteration. It is as though we roll back in time and do not invoke BIID, because we now know that it will fail. Thus, we follow the trace till the end of the first iteration, executing the statements from ℓ_2 to ℓ_7 and then reaching ℓ_2 this time with the state $\{x = y, y \neq 0, err = 0, i = 1\}$. Now, we trigger BIID again with this state. Since `x` and `y` are equal, `x=x-y` at ℓ_3 results in the state $\{x = 0, y \neq 0, err = 0, i = 1\}$ at ℓ_4. This makes the branch condition at ℓ_4 false, so we read `y` (non-zero) at ℓ_5, which makes no change to the state. Therefore the statement `x=x+y` produces the state $\{x = y, y \neq 0, err = 0, i = 1\}$ at ℓ_7, which makes the branch condition false. Thus we reach the loop header ℓ_2 with the state $\{x = y, y \neq 0, err = 0, i = 2\}$ after incrementing `i`.

Comparing this with the original state $\{x = y, y \neq 0, err = 0, i = 1\}$ at ℓ_2, BIID would only delete the constraint $i = 1$ to end up with the invariant $\{x = y, y \neq 0, err = 0\}$. This is a safe invariant because combined with the rest of the trace after the loop, it implies the target $\{x \neq 0, err \neq 1\}$. Thus our method compresses the remaining 9 iterations of the loop. The final compressed trace returned is shown below:

```
ℓ1  x=0, y=0, err=0
ℓ2  i=0, i < n
ℓ3       x = x-y
ℓ4       x == 0, skip
ℓ5       y = read(), y != 0
ℓ6       x = x+y
ℓ7       x != 0, skip
ℓ2  INV: {x = y, y ≠ 0, err = 0}
ℓ2  i ≥ n
ℓ8  TARGET: {x ≠ 0, err ≠ 1}
```

This compressed trace is a collection of similar traces that imply the target after executing at least one loop iteration. Note that we were unable to discover a safe invariant initially but were able to after one unroll. One may be tempted to replace $\{x = y, y \neq 0\}$ with $\{x \neq 0\}$, as the target is only on `x`. However $\{x \neq 0\}$ by itself is **not** an invariant for the loop.

A. Compressing Traces with Nested Loops

Nested loops pose important technical challenges to our method because when attempting invariant discovery for the outer loop, we may encounter another inner loop. Consider the program in Fig. 1. There are two nested loops, the outer running `i` from 0 to 10 and the inner running `j` from `i` to 10 every iteration. The inner loop assigns `x` to 1 or -1, depending on the value of `j`. A final loop outside the nest adds `x` to `y` (initialized to 0), and the target is that `y` is non-negative. This of course mandates that the value of `x` is non-negative. When the program is executed, it would produce a trace containing 65 loop iterations. For demonstration, assume that we are allowed to slacken [22] constraints in this example, i.e., consider a constraint $\{v = w\}$ as the conjunction $\{v \leq w, v \geq w\}$ so that BIID has more candidates to test for invariance.

Our algorithm would follow the trace till ℓ_2 when it encounters the loop with the state $\{i = 0\}$. At ℓ_2 we would invoke BIID and encounter the inner loop at ℓ_4 with the state $\{i = 0, x = 0, j = i\}$. Now, it is *required* for us to find an invariant for the inner loop in order to proceed symbolically executing the rest of the outer loop's body. That is, *we cannot*

```
ℓ1  i=0
ℓ2  while (i < 10) do
ℓ3      x=0, j=i
ℓ4      while(j < 10) do
ℓ5          if (j ≥ 1) x=1 else x= -1
ℓ6          j=j+1
        done
ℓ7      i=i+1
    done
ℓ8  for (y=0,k=0; k < 10; k++) do
ℓ9      y=y+x
    done
ℓ10 TARGET: {y ≥ 0}
```

Fig. 1. Example with nested loops

```
ℓ1   i=0
ℓ2   i < 10
ℓ3       x=0, j=i
ℓ4       INV: {i = 0}
ℓ4       j ≥ 10
ℓ7       i=i+1
ℓ2   i < 10
ℓ3       x=0, j=i
ℓ4       INV: {i = 1, j ≥ i, x ≥ 0}
ℓ4       j ≥ 10
ℓ7       i=i+1
ℓ2   INV: {i ≥ 1, x ≥ 0}
ℓ2   i ≥ 10
ℓ8   y=0, k=0
ℓ8   INV: {i ≥ 1, x ≥ 0, y ≥ 0, k ≥ 0}
ℓ8   k ≥ 10
ℓ10  TARGET: {y ≥ 0}
```

Fig. 2. Compressed trace for the program in Fig. 1

unroll the inner loop during the process of discovering an invariant for the outer loop, because in general, this can lead to unbounded unrolling. For instance, if the inner loop's iteration count depended on a variable whose constraints were deleted by BIID during the outer loop's invariant discovery, unrolling the inner loop may not terminate. Thus, unrolling a loop is only allowed when we are not already within a BIID attempt.

BIID would discover the inner loop invariant $\{i = 0, j \geq i\}$ given the state at ℓ_4, but we cannot say anything about x as it may be 1 or -1. Thus the state at ℓ_7 is $\{i = 0, j \geq i\}$, and after executing i=i+1, we reach ℓ_2 with the state $\{i = 1\}$. We dropped the constraint $\{j \geq i\}$ as i has been incremented. Comparing this with the original state $\{i = 0\}$ at ℓ_2, BIID would produce $\{i \geq 0\}$ as the outer loop invariant. Now, to check if an invariant is safe, we symbolically execute the rest of the trace after the loop starting with the invariant as the symbolic state. If the target is implied at the end, the invariant is safe, otherwise it is not. In this case, the rest of the trace contains 10 iterations of the loop at ℓ_8. Symbolically executing it with $\{i \geq 0\}$ would not imply $\{y \geq 0\}$, because the invariant did not capture any constraint on x, which is added to y.

Our failsafe mechanism now unrolls the outer loop's first iteration, as though it was placed outside the loop in the program. Now, we would again reach ℓ_4 with the state $\{i = 0, x = 0, j = i\}$, and discover the invariant $\{i = 0\}$. Note that we now discovered an invariant only for the inner loop at ℓ_4, and so the rest of the trace would contain the remaining 9 outer loop iterations. This would finally assign x to 1 and cause y to be incremented, implying the target. Hence, $\{i = 0\}$ is indeed a safe invariant for the inner loop. Thus, even though BIID was unable to discover an invariant for the outer loop immediately, it unrolled its first iteration and managed to compress the inner loop in that iteration with the invariant $\{i = 0\}$.

Next, we follow the trace executing i=i+1, and reach the second iteration of the outer loop at ℓ_2 with the state $\{i = 1\}$. Note what happens this time, when we again trigger BIID at ℓ_2. We reach ℓ_4 with the state $\{i = 1, x = 0, j = i\}$. As before, j is indeed incremented in the loop, but the symbolic state guarantees that it will always be greater than or equal to 1. This makes the "else" branch at ℓ_5 an *infeasible* path, which ensures that x would never be assigned -1. Therefore, this time we discover the invariant $\{i = 1, j \geq i, x \geq 0\}$ for the inner loop (we could not have deduced this the first time because the state at ℓ_4 was $\{i = 0, x = 0, j = i\}$, which does not make the "else" branch infeasible). Now, executing i=i+1, we reach ℓ_2 with the state $\{i = 2, x \geq 0\}$. Now, we can discover the outer loop invariant $\{i \geq 1, x \geq 0\}$ for its remaining iterations. This is a *safe* invariant, as executing the rest of the trace with this state will imply the target $\{y \geq 0\}$.

Thus, the compressed trace (Fig. 2) would contain 2 iterations of the outer loop, both of which contain a compression of the inner loop with the invariants $\{i = 0\}$ and $\{i = 1, j \geq i, x \geq 0\}$ respectively, followed by a compression of the outer loop with the invariant $\{i \geq 1, x \geq 0\}$. It can be seen that symbolically executing the third loop at ℓ_8 with this state, BIID can discover the (safe) invariant $\{i \geq 1, x \geq 0, y \geq 0, k \geq 0\}$, and compress it without any unrolling.

Now, our invariant generalization proceeds as follows. It computes the *weakest precondition* (WP) along the compressed trace starting from the target. At loop headers, it attempts to weaken the discovered invariant as long as the post-condition is still implied. For instance, the weakest precondition of the trace in Fig. 2 at ℓ_8 is $\{y \geq 0\}$. Now, it deletes constraints from the invariant $\{i \geq 1, x \geq 0, y \geq 0, k \geq 0\}$ as long as the resulting formula is still invariant and implies $\{y \geq 0\}$. This way, it can delete $\{i \geq 1\}$ and $\{k \geq 0\}$ as the resulting invariant $\{x \geq 0, y \geq 0\}$ implies $\{y \geq 0\}$. However it cannot delete $\{x \geq 0\}$ because without it, $\{y \geq 0\}$ cannot be invariant, and hence cannot imply the post-condition. Thus, it ends up with the now generalized invariant $\{x \geq 0, y \geq 0\}$ at ℓ_8. The WP of this formula is then passed up along the trace, and the process repeats till the top is reached. Branch statements in the trace are ignored during WP propagation, as they have all been evaluated to *true*.

III. Related Work

The works that try to compress traces by removing irrelevant information are most closely related to ours. Dynamic

slicing [25] is the traditionally used technique that uses dependency information to remove statements from the trace not contributing to the target. Some enhancements to the pruning power of dynamic slicing were proposed in [38], [39], [40]. However, compared to loop invariants, dependency information is limited in its ability to compress loop iterations. In Fig. 1, dynamic slicing cannot remove any iteration, as the target variable y is either data- or control-dependent on every program statement.

Recently, more intelligent methods to find irrelevant statements in the trace were proposed in [11], [7], using "error invariants". These are abstractions of the program state at each point that, combined with the rest of the trace, will imply the target. If the error invariant at two points is the same, the code between them is deemed irrelevant. The most important difference with our work is that "error invariants" are not guaranteed to be loop invariants even at looping points, whereas the whole purpose of our paper is to find loop invariants. This difference is because fundamentally, [11], [7] consider the trace simply as a sequence of transitions and are agnostic about loops.

Other related work include those that generate loop invariants, but may not be concerned with compressing execution traces. For instance, [3] uses a template-based invariant generation technique [10] to refine counter-example paths in the context of CEGAR [8]. Contrary to execution traces that terminate, CEGAR systems generate *abstract* counter-examples, and so [3] does not unroll loops, for it may go into non-termination. As a result, in case a safe invariant could not be found, [3] simply tries a different template, whereas we unroll the loop, dynamically exposing more constraints for invariance. Other works that discover invariants using program analysis such as [18], [24] do not guarantee to find a safe invariant. In our experiments, we used the tools in [18] and [26] to generate invariants for some of our benchmarks, only to find that they did not return safe invariants.

Daikon [12] discovers "likely invariants" by instrumenting the program with predicates, executing test cases and checking which predicates are not falsified upto some sufficient degree of tests. The main difference with our work is that we strive to produce *safe* invariants, whereas [12] only attempts to produce predicates that are "likely" to be invariants. Having said that, their technique can still be utilized in our method to discover more sophisticated invariants, for instance, by providing BIID with likely predicates inferred by their method to test for invariance. Another related work [14] presents several heuristics for computing invariants by mutating postconditions of loops. However their method performs no unrolling of loops in case the invariant was found to be unsafe.

There are numerous other works [31], [13], [35], [27], [32] that discover invariants through static/dynamic analysis, testing, constraint solving, heuristics, etc. Invariant discovery is a heavily studied area for decades and so it is formidable to compare extensively with every technique proposed. However, we make it clear that in our paper, any invariant discovery method can be applied, i.e., our proposed BIID method can be augmented with any amount of sophistication to make our trace compression algorithm better. The main contribution of this paper is not BIID itself, but an algorithm to *utilize* an invariant discovery technique such as BIID to compress and generalize traces, along with a novel backup mechanism if the invariant fails, and provide a proof of invariance.

Finally, our work is inspired from [20], which performs full symbolic execution of the program in the context of program verification. It however does not handle traces, and therefore has no way of checking if an invariant is safe until it discovers invariants for all loops in the program. Then, if the target is not implied, it needs a "refinement step" (similar to CEGAR) to find the unsafe invariant. Moreover, since it performs unbounded symbolic execution on the program, it does not guarantee to terminate. Nevertheless, our work can be considered an adaptation of [20] for traces.

IV. BACKGROUND

Syntax. To simplify the formalism, we restrict our presentation to a simple imperative programming language where all basic operations are either assignments or assume operations, and the domain of all variables are integers. The set of all program variables is denoted by *Vars*. An *assignment* $\mathsf{x} := \mathsf{e}$ corresponds to assign the evaluation of the expression e to the variable x. In the *assume* operator, $\mathsf{assume}(\mathrm{c})$, if the boolean expression c evaluates to *true*, then the program continues, otherwise it halts. The set of operations is denoted by *Ops*. We then model a program by a *transition system*. A transition system $\mathcal{P}$ is defined by the tuple $\langle \Sigma, \longrightarrow \rangle$ where Σ is the set of program locations, and $\longrightarrow \subseteq \Sigma \times \Sigma \times \mathit{Ops}$ is the transition relation that relates a program location to its (possible) successors executing operations. This transition relation models the operations that are executed when control flows from one program location to another. We shall use $\ell \xrightarrow{\mathsf{op}} \ell'$ to denote a transition relation from $\ell \in \Sigma$ to $\ell' \in \Sigma$ executing the operation $\mathsf{op} \in \mathit{Ops}$.

Symbolic Execution. A *symbolic state* σ is a tuple $\langle \ell, \mathcal{C} \rangle$. The symbol $\ell \in \Sigma$ corresponds to the current program location (with special symbols for initial location, ℓ_{start}, and final location, ℓ_{end}). $\mathcal{C}$ is a set of constraints on the program variables at the location ℓ, which is to be interpreted as a conjuncted first-order logic formula (e.g., $\mathcal{C} = \{x > 5, y < 3\}$ is interpreted as the formula $x > 5 \wedge y < 3$) that must be satisfied for symbolic execution to follow the particular corresponding path. The set of first-order formulas and symbolic states are denoted by *FOL* and *SymStates*, respectively.

Given a transition system $\langle \Sigma, \longrightarrow \rangle$ and a state $\sigma \equiv \langle \ell, \mathcal{C} \rangle \in \mathit{SymStates}$, the symbolic execution of $\ell \xrightarrow{\mathsf{op}} \ell'$ returns another symbolic state defined as:

$$\mathsf{SYMSTEP}(\sigma, \ell \xrightarrow{\mathsf{op}} \ell') \triangleq \begin{cases} \langle \ell', \mathcal{C} \cup \{\overline{c}\} \rangle & \text{if } \mathsf{op} \equiv \mathsf{assume}(\mathrm{c}) \text{ where } \overline{c} \text{ is c with proper renaming} \\ \langle \ell', \mathcal{C} \cup \{x_k = \overline{e}\} \rangle & \text{if } \mathsf{op} \equiv \mathsf{x} := \mathsf{e} \text{ where } x_k \text{ is fresh and } \overline{e} \text{ is e with proper renaming} \end{cases} \quad (1)$$

Note that while adding the constraint to $\mathcal{C}$, we rename each variable in the constraint to its latest version in $\mathcal{C}$. For assignments, we create a fresh variable on the left hand side. This intuitively mimics a Static Single Assignment (SSA) based symbolic execution of the transition $\ell \xrightarrow{\mathsf{op}} \ell'$.

Given a symbolic state $\sigma \equiv \langle \ell, \mathcal{C} \rangle$ we define the *evaluation* of σ, represented as $[\![\sigma]\!] : \mathit{SymStates} \rightarrow \mathit{FOL}$ as the *projection* of the constraints in $\mathcal{C}$ onto the set of program variables *Vars*. The projection is performed by eliminating existentially all auxiliary variables that are not in *Vars*. Intuitively $[\![\sigma]\!]$ is an FOL formula only on the latest versions of variables in $\mathcal{C}$, and is equisatisfiable with $\mathcal{C}$.

A *symbolic path* $\pi \equiv \sigma_0 \cdot \sigma_1 \cdot \ldots \cdot \sigma_n$ is a sequence of symbolic states such that $\forall i \bullet 1 \leq i \leq n$ the state σ_i is a *successor* of σ_{i-1}. A symbolic state $\sigma' \equiv \langle \ell', \cdot \rangle$ is a successor of another $\sigma \equiv \langle \ell, \cdot \rangle$ if there exists a transition relation $\ell \xrightarrow{\mathsf{op}} \ell'$. A path $\pi \equiv \sigma_0 \cdot \sigma_1 \cdot \ldots \cdot \sigma_n$ is *feasible* if $\sigma_n \equiv \langle \ell, \mathcal{C} \rangle$ such that $[\![\sigma_n]\!]$ is satisfiable. Note that this needs a query to a *theorem prover* for satisfiability checking on the resulting formula. We assume the theorem prover is sound but not necessarily complete. That is, the theorem prover must say a formula is unsatisfiable only if it is indeed so. If $[\![\sigma_n]\!]$ is unsatisfiable the path is called *infeasible* and σ_n is called an *infeasible state*.

A *symbolic execution tree* contains all the execution paths explored during the symbolic execution of a transition system by triggering Eq. (1). The nodes represent symbolic states and the edges represent transitions between states.

Trace. A *trace* $\mathcal{T}$ is a sequence of transitions $\ell_{\mathsf{start}} \xrightarrow{\mathsf{op}_1} \ell_1 \cdot \ell_1 \xrightarrow{\mathsf{op}_2} \ell_2 \cdots \ell_{n-1} \xrightarrow{\mathsf{op}_n} \ell_{\mathsf{end}}$ such that each ℓ_{i+1} is a successor of ℓ_i, and is obtained by executing the program $\mathcal{P}$ with certain inputs. W.l.o.g, we assume that the inputs have been encoded as assignments in the trace itself. Abusing notation, we say "σ is the symbolic state at ℓ along the trace" if symbolic execution, starting with the state $\sigma_{\mathsf{start}} \equiv \langle \ell_{\mathsf{start}}, \emptyset \rangle$ at the beginning of the trace, results in the state σ at ℓ (if ℓ is within a loop, there may be multiple symbolic states at ℓ, in which case we would make unambiguous the state that we are referring to). Again w.l.o.g, we assume that a *target property* (e.g., an assertion) is provided at ℓ_{end} that is implied by $\mathcal{T}$. We interpret this property as a FOL formula ϕ on the program variables. Formally, if σ_{end} is the symbolic state at ℓ_{end} along the trace $\mathcal{T}$, then $[\![\sigma_{\mathsf{end}}]\!] \models \phi$.

A *compressed trace* $\mathcal{T}_c$ is a sequence of transitions where some programs points are annotated with loop invariants. Formally, $\mathcal{T}_c$ is a sequence $\ell_{\mathsf{start}} \xrightarrow{\mathsf{op}_1} \ell_1 \cdot \ell_1 \xrightarrow{\mathsf{op}_2} \ell_2 \cdots \ell_{m-1} \xrightarrow{\mathsf{op}_m} \ell_{\mathsf{end}}$ where there may exist transitions $\ell_{\mathsf{loop}} \xrightarrow{\mathcal{I},\mathcal{S}} \ell_{\mathsf{loop}}$ where ℓ_{loop} is a looping point, $\mathcal{I}$ is a FOL formula representing the loop invariant, and $\mathcal{S}$ is the symbolic execution tree which stands as a proof of invariance of $\mathcal{I}$, should the programmer wish to check that $\mathcal{I}$ is indeed an invariant. An important property of $\mathcal{T}_c$ is that the invariants are safe. That is, if σ'_{end} is the symbolic state at ℓ_{end} along $\mathcal{T}_c$, then $[\![\sigma_{\mathsf{end}}]\!] \models [\![\sigma'_{\mathsf{end}}]\!] \models \phi$. Moreover, the length of $\mathcal{T}_c$ is at most the length of $\mathcal{T}$ (i.e., $m \leq n$) as a result of looping points being compressed using invariants.

```
COMPRESSTRACE(𝒯, 𝒫, φ)
1:    σ := ⟨ℓ_start, ∅⟩ and t := 1
2:    while t ≠ t_end do
3:        let 𝒯[t] := ℓ --op--> ℓ'
4:        if ℓ is a loop from 𝒯 [t] to 𝒯 [t_exit] then
5:            ⟨σ', 𝒮⟩ := LOOPINV_BIID(σ, 𝒫)
6:            if (CHECKSAFEINV(σ', 𝒯, t_exit, φ)) then
7:                t := t_exit + 1
8:                𝒯_c := 𝒯_c · ℓ --[[σ']],𝒮--> ℓ
9:                σ := ⟨ℓ_exit, 𝒞'⟩ where 𝒞' is the constraint list
                       of σ' and ℓ_exit is the loop exit point
10:               continue
11:           endif
12:       endif
13:       𝒯_c := 𝒯_c · ℓ --op--> ℓ'
14:       σ:= SYMSTEP(σ, ℓ --op--> ℓ') and t := t + 1
15:   done
16:   return GENERALIZE(𝒯_c, 𝒫, φ)

CHECKSAFEINV(σ, 𝒯, t_exit, φ)
17:   for t = t_exit to t_end do
18:       let 𝒯 [t] ≡ ℓ --op--> ℓ'
19:       σ := SYMSTEP(σ, ℓ --op--> ℓ')
20:   done
21:   if [[σ]] ⊨ φ then return true else return false
```

Fig. 3. Loop Compression with Invariants

V. ALGORITHM

We now present our algorithm in two phases. In the first phase, we perform forward symbolic execution along the trace to compute inductive invariants to compress the loops. Here we attempt to discover the *strongest* possible invariants and also build the SE tree for each invariant discovered. In the second phase, we generalize the invariants in the compressed trace using backward weakest-precondition computation.

A. Loop compression with invariants

Our main algorithm consists of the procedures shown in Fig. 3. The main procedure, COMPRESSTRACE, takes as inputs the trace $\mathcal{T}$, program $\mathcal{P}$ and the target ϕ, and returns a compressed trace $\mathcal{T}_c$ as defined in Section IV. It models the trace $\mathcal{T}$ as an array using the variable t as the index variable, and t_{end} its length. It starts by initializing σ, representing the current symbolic state, to $\langle \ell_{\mathsf{start}}, \emptyset \rangle$ and t to 1. In line 2, a loop runs till the end of the trace is reached (i.e., till t becomes t_{end}), doing the following in each iteration. Assuming that the current transition along the trace is from ℓ to ℓ', it checks if ℓ is the starting point of a loop. If so, then let the loop's iterations in $\mathcal{T}$ run from the current trace index t to, say, t_{exit}. That is, $\mathcal{T}$ $[t]$ is the transition from ℓ to the loop body and $\mathcal{T}$ $[t_{\mathsf{exit}}]$ is the transition from ℓ to the loop's exit.

Now the algorithm attempts to discover an invariant for this loop in the program by calling the procedure LOOPINV_BIID

with the current symbolic state σ and the program $\mathcal{P}$ (line 5). In principle, this procedure can implement any algorithm that generates a loop invariant, for example [3], [10], [26]. We only require the procedure to return a tuple $\langle \sigma', \mathcal{S} \rangle$ where σ' is an invariant state at ℓ (i.e., $[\![\sigma]\!] \models [\![\sigma']\!]$ and $[\![\sigma']\!]$ is an invariant) and $\mathcal{S}$ is the SE tree, the proof that $[\![\sigma']\!]$ is indeed invariant through the loop at ℓ. In this paper, this procedure will implement the BIID technique, which is one particular way of discovering invariants using symbolic execution (SE) and using the SE tree as the proof tree for invariance.

Now that σ' is an invariant at ℓ, the algorithm then checks whether it is a *safe invariant*. The idea is to symbolically execute the trace after the loop with the discovered invariant as the state and to check if the target is implied at the end. This is done by calling CHECKSAFEINV at line 6 with the invariant state σ', the trace $\mathcal{T}$ and t_{exit}, the index of the loop's exit transition along $\mathcal{T}$, and the target ϕ. CHECKSAFEINV basically implements the symbolic execution along the trace starting at t_{exit} till t_{end} (lines 17-20), and checks whether the symbolic state at t_{end} implies ϕ (line 21).

If the check passed, then σ' represents a safe invariant, meaning the loop has been compressed. Therefore we can continue with our method along the rest of the trace after the loop. Recall that t_{exit} was the index in $\mathcal{T}$ for the loop's exit transition. Hence, in line 7, the algorithm assigns the trace index variable t to t_{exit}+1. In the next line, it records the safe invariant in the compressed trace $\mathcal{T}_c$ by adding to it the (looping) transition $\ell \xrightarrow{[\![\sigma']\!], \mathcal{S}} \ell$, where $[\![\sigma']\!]$ is the invariant, and $\mathcal{S}$ is the proof for its invariance. Finally, the state σ is updated to $\langle \ell_{\mathsf{exit}}, \mathcal{C}' \rangle$ to signify that symbolic execution should continue from the loop's exit point ℓ_{exit} with the constraint list $\mathcal{C}'$ that carries the invariant's (σ') state.

If the check at line 6 failed, it means the invariant turned out to be unsafe, in which case our algorithm discards the work done, and unrolls the loop by simply following the trace $\mathcal{T}$. It adds the current transition from ℓ to ℓ' to the compressed trace $\mathcal{T}_c$ (line 13), and symbolically executes the current state σ with the current transition to get the next state (line 14). These steps simulate unrolling the loop along the trace $\mathcal{T}$, until a loop header is reached again at line 4. The entire process continues until the end of the trace $\mathcal{T}$ is reached, at which point it calls GENERALIZE to generalize the invariants in $\mathcal{T}_c$.

Basic Individually Invariant Discovery (BIID). We now present our particular method to discover invariants and their respective proofs when calling LOOPINV_BIID. At a high level, our method follows paths in the transition system $\mathcal{P}$ starting at the given loop header, and at the end of each path π that reaches the looping point again (making a cycle), it deletes individual constraints at the loop header that do not still hold at the end of π. The process is repeated until no deletions are made, i.e., until a *fix-point* is reached. This entails that the constraints left undeleted at the loop header still hold at the end of every path through the loop, in other words, being invariant through the loop. The SE tree generated at fix-point provides the proof of invariance. This method provides a fine balance between getting the strongest invariants and efficiency.

```
LOOPINV_BIID(σ, P)
1:  do
2:      σ'' := σ  and  π := ∅  and  S := ∅
3:      ⟨σ, S⟩ := LOOPINV (σ, P, π, S)
4:  until σ'' == σ
5:  return ⟨σ, S⟩

LOOPINV(σ≡ ⟨ℓ, C⟩, P, π, S)
6:  if ∃ σ_h ≡ ⟨ℓ, ·⟩ ∈ π then
7:      REMOVENONINV(σ_h, σ)
8:      S := S ∪ {π}  and  return ⟨σ, S⟩
9:  foreach ℓ --op--> ℓ' ∈ P do
10:     σ' := SYMSTEP(σ, ℓ --op--> ℓ')
11:     if [[σ']] is unsat then continue
12:     ⟨·, S'⟩ := LOOPINV (σ', P, π · σ, S)
13:     S := S ∪ S'
14: done
15: return ⟨σ, S⟩

REMOVENONINV (σ_h ≡ ⟨ℓ, C⟩, σ)
16: let [[σ_h]] be c_1 ∧ c_2 ∧ ... ∧ c_n
17: foreach c_i in [[σ_h]] do
18:     if [[σ]] ⊭ c_i then C := C \{c_i}
19: done
```

Fig. 4. Basic Individually Invariant Discovery

LOOPINV_BIID (Fig. 4) is a wrapper procedure that implements the fix-point computation (lines 1-4) on the symbolic state σ. It initializes π to $\varnothing$ (the empty sequence) and $\mathcal{S}$ to $\emptyset$, where π will be used by another procedure to represent the current symbolic path, and $\mathcal{S}$ will eventually represent the SE tree from which the proof that $[\![\sigma]\!]$ is an invariant is extracted. It then calls the procedure LOOPINV passing the state σ and these initialized variables. This process is repeated until fix-point is reached (line 4).

LOOPINV is a recursive procedure that explores all paths in the loop from the current state σ. In lines 9-13, for each transition from ℓ in the program $\mathcal{P}$, it first obtains the next state σ' by performing a symbolic step from σ. Then, if σ' is not an infeasible state, it recursively calls itself with σ', appending σ to the current path π to signify that it has been reached. The symbolic tree returned, corresponding to the execution of σ', is stored in $\mathcal{S}'$ which is then combined together with $\mathcal{S}$ (line 13). In lines 6-8, it checks if a cyclic looping point has been reached (i.e., the current program point ℓ has already been visited along the path π). If so, it removes constraints from the loop header that were not invariant through π, by calling REMOVENONINV with σ_h, the symbolic state at the loop header. Then, since the end of the path has been reached, it adds π to the symbolic tree $\mathcal{S}$ and returns (line 8).

REMOVENONINV is a straightforward procedure. It first obtains the list of constraints at σ_h in evaluated form by applying $[\![\sigma_h]\!]$ (line 16). Then, for each constraint c_i it checks if c_i still holds at the end of the path by checking if $[\![\sigma]\!]$

entails c_i. If not, c_i is deleted from the list of constraints at σ_h (lines 17-19). At the highest level LOOPINV_BIID repeatedly calls LOOPINV until no more deletions are made at the loop header. Once fix-point is reached (line 4), σ becomes an invariant state at the loop header. It is then returned along with $\mathcal{S}$ which serves as the proof that σ is indeed invariant.

There are two important remarks about our algorithm:

(A) Path-explosion. In general, there can be an exponential number of paths within a loop, making BIID intractable. This "path-explosion" problem in symbolic execution is tackled using *interpolation* [22], [28], [23]. The basic idea is to avoid the redundant exploration of a state σ at a program point ℓ if it is *equivalent* to another state σ' at ℓ ($[\![\sigma]\!] = [\![\sigma']\!]$). Interpolation increases the likelihood of this by discarding irrelevant information when comparing σ and σ'. That is, when σ' was explored, interpolation would remove certain constraints from it such that even if $[\![\sigma]\!] \models [\![\sigma']\!]$ (a weaker condition), σ can be considered equivalent to σ', and need not be explored. Although interpolation itself is orthogonal to this paper, it is an important optimization without which BIID cannot scale. We tacitly assume that if interpolation is used to build the symbolic tree, there is an "oracle" that may remove constraints from symbolic states and instructs our algorithm when a symbolic state need not be redundantly explored.

(B) Proof of invariance. For each invariant that we discovered, we can also produce a proof of invariance in the form of a Hoare-proof [19] extracted from the loop's SE tree $\mathcal{S}$. Roughly, during symbolic execution, if a state with constraints C_1 leads to a state with constraints C_2 executing a statement op, we can form the Hoare-triple $\{C_1\}\ \mathsf{op}\ \{C_2\}$. Axiomatic triples can then be composed using standard Hoare-logic rules [19] to form the entire proof. The process is straightforward but tedious, and is not in scope of this paper. The purpose of this proof is it can be provided to an automatic proof assistant such as Coq [2] to ensure the validity of our invariants, if required.

B. Invariant Generalization

The invariants discovered in the previous phase are guaranteed to be safe, but may be logically too strong. The idea in this phase is to perform a (backward) *weakest precondition* computation along the compressed trace starting from the target, and to weaken the invariants as long as the weakest precondition is implied. This algorithm is shown in Fig. 5.

The GENERALIZE procedure takes as input the compressed trace $\mathcal{T}_c$ along with the target ϕ and the program. It begins by initializing the "post-condition" variable Ψ to ϕ at line 1. Then, starting at $t_{c_{\mathsf{end}}}$ (the end of $\mathcal{T}_c$) and going backwards, it does the following at each step. If the current transition is not an annotated loop invariant, it computes the weakest liberal precondition ($\widehat{wlp}$) of Ψ along the transition op (lines 3-4). The wlp is defined as the weakest formula on the pre-state such that if the execution of op terminates, it results in the post-state Ψ. In practice, it can be approximated by making a linear number of calls to a theorem prover, as shown in [22].

GENERALIZE($\mathcal{T}_c$, $\mathcal{P}$, ϕ)
1: $\Psi := \phi$
2: **for** $t_c := t_{c_{\mathsf{end}}}$ to 1 **do**
3: **if** $\mathcal{T}_c\ [t_c] \equiv \ell_1 \xrightarrow{\mathsf{op}} \ell_2$ **then**
4: $\Psi := \widehat{wlp}\ (\Psi, \mathsf{op})$
5: **else if** $\mathcal{T}_c\ [t_c] \equiv \ell \xrightarrow{\mathcal{I},\mathcal{S}} \ell$ **then**
6: $\mathcal{I}' := \nabla\ (\mathcal{I}, \Psi)$
7: $\sigma' := \langle \ell, \mathcal{I}' \rangle$
8: **if** LOOPINV_BIID (σ', $\mathcal{P}$) returns $\langle \sigma'', \mathcal{S}' \rangle$ s.t. $\sigma'' = \sigma'$ **then**
9: $\mathcal{T}_c\ [t_c] := \ell \xrightarrow{\mathcal{I}',\mathcal{S}'} \ell$
10: $\Psi := \mathcal{I}'$
11: **else** $\Psi := \mathcal{I}$
done
12: **return** $\mathcal{T}_c$

Fig. 5. Invariant Generalization using Weakest Precondition

If the current transition is a loop invariant (line 5) $\mathcal{I}, \mathcal{S}$ at ℓ, it first computes a *widening* of $\mathcal{I}$ w.r.t. the post-condition Ψ. The widening operator ∇ returns a formula $\mathcal{I}'$ such that $\mathcal{I} \models \mathcal{I}' \models \Psi$, i.e., $\mathcal{I}'$ is weaker than $\mathcal{I}$ but still strong enough to imply the post-condition. However, we need to check if $\mathcal{I}'$ is still invariant through the loop. This is done in lines 7-8 by calling LOOPINV_BIID and checking whether it returns the same invariant state σ' at the loop header. If so, then line 9 replaces the current invariant in $\mathcal{T}_c$ with the new annotation $\mathcal{I}', \mathcal{S}'$ where $\mathcal{S}'$ is the symbolic tree generated for $\mathcal{I}'$ by LOOPINV_BIID (note that $\mathcal{S}'$ can be different from $\mathcal{S}$ because it is the SE tree for a different, weaker, invariant). Finally, line 10 sets $\mathcal{I}'$ to be the post-condition to be propagated back.

If the weakened formula $\mathcal{I}'$ is not invariant, the algorithm makes no change to the existing annotation and just propagates $\mathcal{I}$ backward (line 11). Once the beginning of the trace is reached, the algorithm returns $\mathcal{T}_c$ – the compressed trace containing now generalized invariants with SE trees as proofs.

THEOREM 1 [TERMINATION]. *Given an execution trace $\mathcal{T} \equiv t_1 \cdot t_2 \cdots t_n$ where each t_i is a transition $\ell_i \xrightarrow{\mathsf{op}} \ell_j$,* COMPRESSTRACE *will terminate and return a compressed trace $\mathcal{T}_c \equiv t_1 \cdot t_2 \cdots t_m$ where each t_i is either a transition or an invariant annotation $\ell_{loop} \xrightarrow{\mathcal{I},\mathcal{S}} \ell_{loop}$, such that $m \leq n$.*

This follows directly from the fact that we replace loop iterations with invariants, thus reducing the number of transitions. Termination is guaranteed since our unrolling is bounded by the (finite) number of loop iterations in the execution trace.

VI. EXPERIMENTAL EVALUATION

We implemented our algorithm on the TRACER [21] framework for symbolic execution and evaluated it on several medium-sized benchmarks from the software verification competition SV-COMP'13 [4]. The programs are all unsafe, and in order to work with a meaningful trace, we invoked Directed Automated Random Testing (DART) [15], [5], commonly

Benchmark	Trace length		%C	#U	#HT	Time
	Orig.	Com.				
SSH client	462	95	80%	6	47	289s
SSH server	346	13	96%	0	91	94s
tokenring	885	218	75%	2(2)	66	161s
cdaudio	1434	121	92%	0	15	128s
floppy	398	83	80%	1	4	2s

TABLE I

TRACE STATISTICS FOR OUR EXPERIMENTS. %C: PERCENTAGE COMPRESSION, #U: NUMBER OF UNROLLS UNTIL COMPRESSION WAS ACHIEVED (INNER LOOP UNROLLS, IF ANY), #HT: NUMBER OF HOARE-TRIPLES FOR THE PROOF OF INVARIANCE

known as *concolic testing*, to obtain inputs that violate the safety property. Typically, the inputs control the outcome of non-deterministic branch statements in the program. In all cases, we encoded the inputs into the program, and set our target for the resulting "error trace" to be the negation of the violated property. To make the experimentation easier we applied static slicing provided by Frama-C [1] to remove statically irrelevant statements. All experiments were run on an Intel 2.3Ghz system with 2GB memory. We first tabulate the results in Table I, and explain each benchmark in detail.

A. SSH Client (`s3_clnt_1_false.cil.c`)

Our first example is a buggy SSH client program from the ssh-simplified suite. We provide an abstract overview of the benchmark. It consists of a big loop that reads the current state variable `s` and performs some action, then setting `s` to the next state. In certain states, a `flag` variable is checked to be of some value, and if so, is incremented. In a particular state S, if `flag` was found to be 4, the error variable `err` is set to 1, and `flag` is not modified thereafter. The safety property is that `err` should be 0 in the end (i.e., `flag` should not be 4 when `s` is S), which of course is violated by the given input. We set the target to the negation of the safety property, namely, $\{err = 1\}$. On running the code with buggy inputs, the trace iterated through the loop **40** times, executing a total of **462** transitions, and implied the target.

It is noteworthy that dynamic slicing was unable to remove any iteration as a whole, as the state `s` changes in each iteration and the loop's exit is control-dependent on `s`. To compare with other invariant discovery methods, we computed invariants using (1) the polyhedra abstract domain of APRON [24] and (2) the INVGEN [18] tool. Both of them returned *unsafe* invariants such as $\{$**flag**$\geq$**0**$\}$. This shows that safe invariants cannot often be computed easily, and there is a need for our unrolling mechanism to compute safe invariants in practice.

Our algorithm initially only discovered the unsafe invariant *true*. Upto five unrolls, it only kept discovering unsafe invariants. However, after the **6**th unroll, the trace reached the state S, checked if `flag` was 4, and set `err` to 1. At this point, our algorithm discovered the *safe* invariant $\{$**err=1, flag=4**$\}$, that implies the target $\{err = 1\}$. Note that $flag = 4$ is needed to preserve the invariance of $err = 1$. This is a practical example of a loop converging towards stronger invariants as it iterates, culminating in a safe invariant.

Ultimately, the trace was reduced from **462** transitions to **95** transitions (**80%** compression). Our algorithm also discovered 3 invariants on a few other variables in the program, but the generalization phase deleted them as they were irrelevant to the target. The proof for the invariant contained **47** Hoare-triples. The process took **289s** to complete.

B. SSH Server (`s3_srvr_6_false.cil.c`)

Our next benchmark is a buggy SSH server program from the ssh-simplified suite. The structure of this program is similar to SSH Client, in that a main loop iterates using a state variable `s` until it reaches a particular value. Here, two safety properties are present: one outside (before) the loop and one inside. As before the target is $\{err = 1\}$. On running the buggy inputs, the trace executed the loop **14** times before implying the target, for a total of **346** transitions.

On invoking our algorithm, we reached the loop header with the symbolic state that included $\{err = 1\}$ because in this program, the safety property that was before the loop was itself violated. On symbolically exploring the loop, we concluded that even though the other safety property modifies `err` in the loop, this constraint on `err` remains invariant. Thus, we were able to compress the loop even without any unrolling. The resulting compressed trace contained only **13** transitions, and **91** triples were generated as proof of invariance. We also discovered 20 other invariants which were removed by the generalization phase. The entire process took **94s** to complete.

C. Tokenring (`token_ring01_unsafe.c`)

Our third benchmark is a buggy token ring algorithm from the loops suite. It consists of two functions `master` and `transmit`, and in the main function, a loop calls these two functions in each iteration non-deterministically. This loop is nested within another loop that keeps running the inner loop a given number of times. In a certain instance of the inner loop, when the two functions are called in a particular sequence, an error is triggered in `master`. The inputs controlled the number of iterations of both loops, and determined the specific combination of calls that triggers the error. The inputs directed the trace to execute **5** iterations of the outer loop, and for each of those, **6** iterations of the inner loop – a total of **30** iterations with **885** transitions. We were unable to invoke APRON and INVGEN for this and the following two benchmarks, as their C front-end did not support programs with function calls.

Our algorithm was unable to find a safe invariant for the outer loop immediately. It had to unroll the outer loop twice – however, in the first iteration, it managed to compress the inner loop immediately, and in the second iteration, it unrolled the inner loop twice and compressed it. This benchmark exhibited a remarkable feature of our method in practice – even if we are unable to compress an outer loop iteration, we can unroll it and still compress the inner loop in that iteration.

Ultimately, the trace was reduced to just **2** outer loop iterations, each containing a compression of the inner loop, and finally a compression of the outer loop. The number of transitions was reduced from **885** to **218** (**75%** compression). Importantly, this compressed trace shows that the particular sequence of calls that triggers the error happened only in the

Bench -mark	Bound	Trace length		%C	#U	Time
		Orig.	Compr.			
cdaudio	4	342	121	65%	0	97s
	8	498	121	75%	0	99s
	16	810	121	85%	0	106s
	32	1434	121	92%	0	128s
floppy	4	146	83	43%	1	2s
	8	182	83	54%	1	2s
	16	254	83	67%	1	2s
	32	398	83	80%	1	2s

TABLE II
TREND WITH VARYING LOOP BOUNDS FOR cdaudio AND floppy

second outer loop iteration. This is quite valuable information to a programmer as they can quickly focus debugging efforts on that part, rather than checking which iteration caused the error. Finally, **66** triples were generated for the proof of invariance. The entire process took **161s** to complete.

D. cdaudio (`cdaudio_simpl1_unsafe.cil.c`*)*

Our fourth benchmark is the buggy version of the driver "cdaudio" from the ntdrivers-simplified suite. It consists of 15 safety properties of which exactly 1 is violated. The generated inputs violate this property after going through the (only) loop in the program. We noticed that the loop has an *arbitrary* bound, i.e., the bound is simply the number of attempts made to start the CD-device, set to 4 by default. Once the device is started, signified by a `status` variable, the loop exits.

Within the loop, there exist statements modifying variables that appear in other safety properties in the program. However, the validity of other properties is not affected because of two reasons: (1) the properties capture the *relationship* between the variables and not the actual value (e.g., $\{s = NP\}$, where `s` is set to `NP` in the loop) and (2) given the incoming context to the loop, many statements that modify these variables are along *infeasible paths*. Executing the program with the default bound of 4, the trace ran through **342** statements. On invoking our algorithm, we were able to compress all 4 iterations of the loop with safe invariants that either captured sufficient exact relationships between the variables (e.g., {**s=NP**}) or their values which are sufficient to establish the relationship (e.g., {**s=1,NP=1**}). The trace was reduced to **121** statements, and **15** triples were produced for the proof of invariance.

Since the bound is arbitrary, we tried to increase it in order to see a *trend* of our compression method. In Table II, the row cdaudio shows the statistics for bounds 4, 8, 16 and 32 for the loop. We used these bounds to observe the trend for an *exponentially* increasing loop bound. In all cases our compression resulted in the trace being **121** transitions long. That is, even if the loop bound is increased, we were able to compress it using the same invariant without additional unrolling. In all cases, **15** triples were generated as before. As it can be seen, the amount of compression approaches more than **90%** as the number of iterations increases. Moreover, the timing is not affected drastically, as we were able to finish in about **2 mins** in all cases. The slight increase in timing is due to the CHECKSAFEINV procedure that has to run along the now longer trace to check if a discovered invariant is safe.

E. floppy (`floppy_simpl3_unsafe.cil.c`*)*

Our final benchmark is another buggy driver "floppy" from the ntdrivers-simplified suite. This program has 20 safety properties out of which exactly 1 is violated. A loop in the program, among other things, assigns either 0 or a negative value, say N, to a variable `ntStatus` non-deterministically. If `ntStatus` is assigned 0, or the loop's bound is exhausted, the loop exits. If it is never assigned 0, its value N is passed across many functions to a variable `status`, which is checked to be equal to 259. If the check fails, the error is triggered. The generated buggy inputs execute a trace that always assigns `ntStatus` to N, making the check fail.

We set our target to be $\{status = N\}$. We again noted that the loop bound is arbitrary, and in fact, not even specified in the program. Due to lack of a "default" bound, we simply used the loop bounds from cdaudio, i.e., 4, 8, 16 and 32. When our algorithm was invoked on the trace, we were unable to discover a safe invariant right away, as the loop destroys the initial value of `ntStatus` (i.e., 0) by setting it to N thereby preventing our invariant to capture any constraint on `ntStatus`. After one unroll however, we were able to capture the constraint $\{ntStatus = N\}$ which is now invariant through the loop. This turned out to be a safe invariant, compressing the remaining iterations of the loop.

As before, we show in Table II the compression trend for this benchmark. We obtained traces of sizes **146** to **398** by varying the bounds. In all cases however, we were able to compress the trace to **83** transitions after **1** unroll. The amount of compression varies between **43%** to **80%** depending on the bound. In all cases, **4** triples were generated for the proof of invariance. We were able to compress the trace quickly within **2 secs**, as the invariant discovered caused many paths to become infeasible, making SE terminate fast.

In summary, we have shown that our algorithm can achieve significant compression (75-96%) of execution traces in practice. We conclude this section by pointing out an important observation – for all our benchmarks, the discovered invariants *did not bound* the loop iterations. For example, in "cdaudio", the invariant {**s=1,NP=1**} does not bound any variable controlling the number of loop iterations. This means that the compression resulted in the error trace being generalized to a potentially *infinite* number of traces that cause the error in a similar way. An important benefit of this generalization is that a DSE search procedure can be instructed to, in one go, avoid exploring all these similar buggy paths and instead be directed towards other paths, thus increasing its path coverage.

While this paper focused on producing the actual generalized trace, its many such applications are left as our future work. We believe that in general, the kind of generalization achieved by loop invariants—and we emphasize that ours are *safe invariants obtained after sufficient unrolling*—does provide a very powerful form of "interpolation" for third-party applications, most of which are based on aggregating information from individual traces.

REFERENCES

[1] Frama-C Software Analyzers. http://frama-c.com/.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. M. Noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq proof assistant reference manual—version v6.1. Technical Report 0203, INRIA, 1997.

[3] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *PLDI'07*.

[4] Dirk Beyer. Second competition on software verification. In *TACAS*, 2013.

[5] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. In *ASE*, pages 443–446, 2008.

[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, 2008.

[7] Jurgen Christ, Ermis Ermis, Martin Schaf, and Thomas Wies. Flow-sensitive fault localization. *VMCAI*, 2013.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterExample-Guided Abstraction Refinement. In *CAV'00*.

[9] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, 2005.

[10] Michael Colon, Sriram Sankaranarayanan, and Henry Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.

[11] Ermis Ermis, Martin Schaf, and Thomas Wies. Error invariants. *FM*, 2012.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 27:213–224, 2001.

[13] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. ICSE 2000, 2000.

[14] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, 2010.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.

[16] A. Groce. Error Explanation with Distance Metrics. In *TACAS*, pages 108–122, 2004.

[17] A. Groce and W. Visser. What Went Wrong: Explaining Counterexamples. In *In SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[18] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An Efficient Invariant Generator. In *CAV*, 2009.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 1969.

[20] J. Jaffar, , J.A. Navas, and A. Santosa. Unbounded Symbolic Execution for Program Verification. In *RV 2011*, pages 396–411, 2011.

[21] J. Jaffar, V. Murali, J.A. Navas, and A. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *CAV 2012*, pages 758–766, 2012.

[22] J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *CP*, 09.

[23] Joxan Jaffar, Vijayaraghavan Murali, and Jorge Navas. Boosting Concolic Testing via Interpolation. In *FSE*, 2013.

[24] Bertrand Jeannet and Antoine Mine. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, 2009.

[25] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[26] G. Lalire, M. Argoud, and B. Jeannet. The Interproc Analyzer. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc, 2009.

[27] Mengjun Li. A practical loop invariant generation approach based on random testing, constraint solving and verification. ICFEM'12, 2012.

[28] K. L. McMillan. Lazy annotation for program testing and verification. In T. Touili, B. Cook, and P. Jackson, editors, *22nd CAV*, volume 6174 of *LNCS*, pages 104–118. Springer, 2010.

[29] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. DARWIN: An Approach for Debugging Evolving Programs. ESEC/FSE, 2009.

[30] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. DARWIN: An Approach for Debugging Evolving Programs. *ACM Trans. Softw. Eng. Methodol.*, 2012.

[31] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. POPL '04, 2004.

[32] Peter H. Schmitt and Benjamin Weiß. Inferring invariants by symbolic execution. In *Proceedings, 4th International Verification Workshop (VERIFY'07)*, 2007.

[33] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, pages 263–272, 2005.

[34] Natasha Sharygina and Doron Peled. A combined testing and verification approach for software reliability. FM, 2001.

[35] Jamie Stark and Andrew Ireland. Invariant discovery via failed proof attempts. In *Logic-Based Program Synthesis and Transformation*, 1999.

[36] Andreas Zeller. Isolating cause-effect chains from computer programs. In *FSE '02*, pages 1–10, 2002.

[37] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 2002.

[38] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. PLDI, 2006.

[39] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. PLDI, 2004.

[40] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. ICSE, 2003.