Failure-Directed Program Trimming (Extended Version)

Kostas Ferles
The University of Texas at Austin
Austin, TX, USA
kferles@cs.utexas.edu

Valentin Wüstholz
The University of Texas at Austin
Austin, TX, USA
valentin@cs.utexas.edu

Maria Christakis
University of Kent
Canterbury, UK
M.Christakis@kent.ac.uk

Isil Dillig
The University of Texas at Austin
Austin, TX, USA
isil@cs.utexas.edu

ABSTRACT

This paper describes a new program simplification technique called *program trimming* that aims to improve the scalability and precision of safety checking tools. Given a program $\mathcal{P}$, program trimming generates a new program $\mathcal{P}'$ such that $\mathcal{P}$ and $\mathcal{P}'$ are *equi-safe* (i.e., $\mathcal{P}'$ has a bug if and only if $\mathcal{P}$ has a bug), but $\mathcal{P}'$ has fewer execution paths than $\mathcal{P}$. Since many program analyzers are sensitive to the number of execution paths, program trimming has the potential to improve the effectiveness of safety checking tools.

In addition to introducing the concept of program trimming, this paper also presents a lightweight static analysis that can be used as a pre-processing step to remove program paths while retaining equi-safety. We have implemented the proposed technique in a tool called Trimmer and evaluate it in the context of two program analysis techniques, namely abstract interpretation and dynamic symbolic execution. Our experiments show that program trimming significantly improves the effectiveness of both techniques.

KEYWORDS

Condition inference, abstract interpretation, dynamic symbolic execution

1 INTRODUCTION

Due to its potential to dramatically simplify programs with respect to a certain criterion (e.g., the value of a program variable at a given location), program slicing [85] has been the focus of decades of research in the program analysis community [84]. In addition to being useful for program understanding, slicing also has the potential to improve the scalability of bug-finding and verification tools by removing irrelevant code snippets with respect to some property of interest. Yet, despite this potential, relatively few bug-finding and verification tools use slicing as a pre-processing step.

In this paper, we argue that existing notions of a "program slice" do not adequately capture the kinds of program simplification that are beneficial to safety checking tools. Instead, we propose a new semantic program simplification technique called *program trimming*, which removes *program paths* that are irrelevant to the safety property of interest. Given a program $\mathcal{P}$, program trimming generates a simplified program $\mathcal{P}'$ such that $\mathcal{P}'$ violates a safety property if and only if the original program $\mathcal{P}$ does (i.e., $\mathcal{P}$ and $\mathcal{P}'$ are *equi-safe*). However, $\mathcal{P}'$ has the advantage of containing fewer execution paths than $\mathcal{P}$. Since the scalability and precision of many program analyzers depend on the number of program paths, program trimming can have a positive impact on many kinds of program analyses, particularly those that are *not* property directed.

To illustrate the difference between the standard notion of program slicing and our proposed notion of program trimming, consider the following very simple program, where ⋆ indicates a non-deterministic value (e.g., user input):

```
1 x := ⋆; y := ⋆;
2 if (y > 0) { while (x < 10) { x := x + y; } }
3 else { x := x - 1; }
4 assert x > 0;
```

Suppose that our goal is to prove the assertion; so, we are interested in the value of x at line 4. Now, every single statement in this program is relevant to determining the value of x; hence, there is nothing that can be removed using program slicing. However, observe that the then branch of the if statement is actually irrelevant to the assertion. Since this part of the program can never result in a program state where the value of x is less than 10, lines 2 and 3 can be simplified without affecting whether or not the assertion can fail. Hence, for the purposes of safety checking, the above program is equivalent to the following much simpler trimmed program $\mathcal{P}'$:

```
1 x := ⋆; y := ⋆;
2 assume y <= 0;
3 x := x - 1;
4 assert x > 0;
```

Observe that $\mathcal{P}'$ contains far fewer paths compared to the original program $\mathcal{P}$. In fact, while $\mathcal{P}$ contains infinitely many execution paths, the trimmed program $\mathcal{P}'$ contains only two, one through the successful and one through the failing branch of the assertion. Consequently, program analyzers that eagerly explore all program paths, such as bounded model checkers [14, 28] and symbolic execution engines [65], can greatly benefit from program trimming in terms of scalability. Furthermore, since many static analyzers (e.g., abstract interpreters [31]) typically lose precision at join points of the control flow graph, program trimming can improve their precision by removing paths that are irrelevant to a safety property.

Motivated by these observations, this paper introduces the notion of failure-directed program trimming and presents a lightweight algorithm to remove execution paths in a way that guarantees equi-safety. The key idea underlying our approach is to statically infer *safety conditions*, which are sufficient conditions for correctness and can be computed in a lightweight way. Our technique then negates these safety conditions to obtain *trimming conditions*, which are necessary conditions for the program to fail. The trimming conditions are used to instrument the program with assumptions such that program paths that violate an assumption are pruned.

Program trimming is meant as a lightweight but effective pre-processing step for program analyzers that check safety. We have implemented our proposed trimming algorithm in a tool called Trimmer and used it to pre-process hundreds of programs, most of which are taken from the software verification competition (SV-COMP) [11]. We have also evaluated the impact of trimming in the context of two widely-used program analysis techniques, namely abstract interpretation [31] and dynamic symbolic execution [18, 50]. Our experiments with Crab [48, 49] (an abstract interpreter) show that program trimming can considerably improve the precision of static analyzers. Furthermore, our experiments with Klee [17] (a dynamic symbolic execution tool) show that program trimming allows the dynamic symbolic execution engine to find more bugs and verify more programs within a given resource limit.

To summarize, this paper makes the following key contributions:

- We introduce the notion of program trimming as a new kind of program simplification technique.
- We propose an effective and lightweight inference engine for computing safety conditions.
- We describe a modular technique for instrumenting the program with trimming conditions.
- We demonstrate empirically that program trimming has a significant positive impact on the effectiveness of program analyzers. For instance, the cheapest configuration of Crab (an abstract interpreter) with trimming proves 21% more programs safe than the most expensive configuration of Crab without trimming in *less than 70% of the time*. In the context of a dynamic symbolic execution engine (Klee), trimming increases both the number of uncovered bugs by up to 30% and the number of verified programs by up to 18% while reducing the running time by up to 30%.

2 GUIDED TOUR

The running example, shown in Figure 1, is written in C extended with `assume` and `assert` statements. Note that the example is intentionally quite artificial to illustrate the main ideas behind our technique. Procedure `main` assigns a non-deterministic integer value to variable `m` and computes its factorial using the recursive `fact` procedure. The (light and dark) gray boxes are discussed below and *should be ignored for now*. We examine two variations of this example: one for dynamic symbolic execution (DSE) engines and another for abstract interpreters (AI).

Motivation #1: scalability. First, let us ignore the assertion on line 8 and only consider the one on line 16. Clearly, this assertion cannot fail unless `m` is equal to 123. Observe that procedure `main` contains infinitely many execution paths because the number of recursive calls to `fact` depends on the value of `m`, which is unconstrained. Consequently, a dynamic symbolic execution engine, like Klee, would have to explore (a number of) these paths until it finds the bug or exceeds its resource limit. However, there is only one buggy execution path in this program, meaning that the dynamic symbolic execution engine is wasting its resources exploring paths that cannot possibly fail.

```
 1 int fact(int n) {
 2   assume 0 <= n;
 3   assume n != 0;                // AI
 4   int r = 1;
 5   if (n != 0) {
 6     r = n * fact(n - 1);
 7   }
 8   assert n != 0 || r == 1;      // AI
 9   return r;
10 }
11
12 void main() {
13   int m = ★;
14   assume m == 123;              // DSE
15   int f = fact(m);
16   assert m != 123 || f == 0;    // DSE
17 }
```

Figure 1: Running example illustrating program trimming.

Our approach. Now, let us see how program trimming can help a symbolic execution tool in the context of this example. As mentioned in Section 1, our program trimming technique first computes *safety conditions*, which are sufficient conditions for the rest of the program to be correct. In this sense, standard *weakest preconditions* [37] are instances of safety conditions. However, automatically computing safety conditions precisely, for instance via weakest precondition calculi [37, 69], abstract interpretation [31], or predicate abstraction [4, 52], can become very expensive (especially in the presence of loops or recursion), making such an approach unsuitable as a pre-processing step for program analyzers that already check safety. Instead, we use lightweight techniques to infer safety conditions that describe a subset of the safe executions in the program. That is, the safety conditions inferred by our approach can be stronger than necessary, but they are still useful for ruling out many program paths that "obviously" cannot violate a safety property.

In contrast to a safety condition, a *trimming condition* at a given program point reflects a *necessary* condition for the rest of the program execution to fail. Since a necessary condition for a property $\neg Q$ can be obtained using the negation of a sufficient condition for Q, we can compute a valid trimming condition for a program point π as the negation of the safety condition at π. Thus, our approach trims the program by instrumenting it with assumptions of the form `assume` ϕ, where ϕ is the negation of the safety condition for that program point. Since condition ϕ is, by construction, necessary for the program to fail, the trimmed program preserves the safety of the original program. Moreover, since execution terminates as soon as we encounter an assumption violation, instrumenting the program with trimming conditions prunes program paths in a semantic way.

Program trimming on this example. Revisiting our running example from Figure 1, the safety condition right after line 15 is `m != 123 || f == 0`. Since procedure `fact` called at line 15 neither contains any assertions nor modifies the value of `m`, a valid safety condition right before line 15 is `m != 123`. Indeed, in executions that reach line 15 and satisfy this safety condition, the assertion does not fail. We can now obtain a sound trimming condition by negating the safety condition. This allows us to instrument the program with the `assume` statement shown in the dark gray box of line 14. Any execution that does not satisfy this condition is correct and is effectively removed by the `assume` statement in a way that preserves safety. As a result, a dynamic symbolic execution tool running on the instrumented program will only explore the single execution path containing the bug and will not waste any resources on provably correct paths. Observe that a bounded model checker would similarly benefit from this kind of instrumentation.

Motivation #2: precision. To see how our approach might improve the precision of program analysis, let us ignore the assertion on line 16 and only consider the one on line 8. Since `n = 0` implies `r = 1` on line 8, this assertion can clearly never fail. However, an abstract interpreter, like Crab, using intervals [31] cannot prove this assertion due to the inherent imprecision of the underlying abstract domain. In particular, the abstract interpreter knows that `n` is non-negative at the point of the assertion but has no information about `r` (i.e., its abstract state is $\top$). Hence, it does not have sufficient information to discharge the assertion at line 8.

Suppose, however, that our technique can infer the safety condition `n = 0` on line 3. Using this condition, we can now instrument this line with the trimming condition `n != 0`, which corresponds to the assumption in the light gray box. If we run the same abstract interpreter on the instrumented program, it now knows that `n` is strictly greater than 0 and can therefore prove the assertion even though it is using the same interval abstract domain. Hence, as this example illustrates, program trimming can also be useful for improving the precision of static analyzers in verification tasks.

3 PROGRAM TRIMMING

In this section, we formally present the key insight behind failure-directed program trimming using a simple imperative language in the style of IMP [86], augmented with `assert` and `assume` statements. This lays the foundation for understanding the safety condition inference, which is described in the next section and is defined for a more expressive language. Here, we present the semantics of the IMP language using big-step operational semantics, specifically using judgments of the form $\langle \sigma, s\rangle \Downarrow_{\varphi} \sigma'$ where:

- s is a program statement,
- σ, σ' are *valuations* mapping program variables to values,
- $\varphi \in \{\lightning, \diamond, \checkmark\}$ indicates whether an assertion violation occurred ($\lightning$), an assumption was violated ($\diamond$), or neither assertion nor assumption violations were encountered (denoted $\checkmark$).

We assume that the program terminates as soon as an assertion or assumption violation is encountered. We also ignore non-determinism to simplify the presentation.

Definition 3.1. **(Failing execution)** We say that an execution of s under σ is *failing* iff $\langle s, \sigma\rangle \Downarrow_{\lightning} \sigma'$, and *successful* otherwise.

In other words, a failing execution exhibits an assertion violation. Executions with *assumption* violations also terminate immediately but are not considered failing.

Definition 3.2. **(Equi-safety)** We say that two programs s, s' are *equi-safe* iff, for all valuations σ, we have:

$$\langle s, \sigma\rangle \Downarrow_{\lightning} \sigma' \iff \langle s', \sigma\rangle \Downarrow_{\lightning} \sigma'$$

In other words, two programs are equi-safe if they exhibit the same set of failing executions starting from the same state σ. Thus, program s' has a bug if and only if s has a bug.

As mentioned in Section 1, the goal of program trimming is to obtain a program s' that (a) is equi-safe to s and (b) can terminate early in successful executions of s:

Definition 3.3. **(Trimmed program)** A program s' is a *trimmed version* of s iff s, s' are equi-safe and

$$\begin{array}{ll} (1) & \langle s, \sigma\rangle \Downarrow_{\checkmark} \sigma' \implies \langle s', \sigma\rangle \Downarrow_{\checkmark} \sigma' \lor \langle s', \sigma\rangle \Downarrow_{\diamond} \sigma'' \\ (2) & \langle s, \sigma\rangle \Downarrow_{\diamond} \sigma' \implies \langle s', \sigma\rangle \Downarrow_{\diamond} \sigma'' \end{array}$$

Here, the first condition says that the trimmed program s' either exhibits the same successful execution as the original program or terminates early with an assumption violation. The second condition says that, if the original program terminates with an assumption violation, then the trimmed program also violates an assumption but can terminate in a different state σ''. In the latter case, we allow the trimmed program to end in a different state σ'' than the original program because the assumption violation could occur earlier in the trimmed program. Intuitively, from a program analysis perspective, we can think of trimming as a program simplification technique that prunes execution paths that are guaranteed not to result in an assertion violation.

Observe that program trimming preserves all terminating executions of program s. In other words, if s terminates under valuation σ, then the trimmed version s' is also guaranteed to terminate. However, program trimming does not give any guarantees about non-terminating executions. Hence, even though this technique is suitable as a pre-processing technique for safety checking, it does not necessarily need to preserve liveness properties. For example, non-terminating executions of s can become terminating in s'.

The definition of program trimming presented above does not impose any *syntactic* restrictions on the trimmed program. For instance, it allows program trimming to add and remove arbitrary statements as long as the resulting program satisfies the properties of Definition 3.3. However, in practice, it is desirable to make some syntactic restrictions on how trimming can be performed. In this paper, we perform program trimming by adding assumptions to the original program rather than removing statements. Even though this transformation does not "simplify" the program from a program understanding point of view, it is very useful to subsequent program analyzers because the introduction of `assume` statements prunes program paths in a *semantic* way.

4 STATIC ANALYSIS FOR TRIMMING

As mentioned in Section 1, our trimming algorithm consists of two phases, where we infer safety conditions using a lightweight static analysis in the first phase and instrument the program with trimming conditions in the next phase. In this section, we describe the safety condition inference.

4.1 Programming Language

In order to precisely describe our trimming algorithm, we first introduce a small, but realistic, call-by-value imperative language with pointers and procedure calls. As shown in Figure 2, a program in this language consists of one or more procedure definitions. Statements

$$
\begin{array}{llll}
\text{Program } \mathcal{P} & ::= & \overline{prc} \\
\text{Procedure } prc & ::= & \texttt{proc}\ \ prc(\overline{v_{in}}) : v_{out}\ \ \{s\} \\
\text{Statement } s & ::= & s_1; s_2 \mid v := e \mid v_1 := *v_2 \mid *v := e \\
& \mid & v := \texttt{malloc}(e) \mid v := \texttt{call}\ \ prc(\bar{v}) \\
& \mid & \texttt{assert}\ p \mid \texttt{assume}\ p \\
& \mid & \texttt{if}\ (\star)\ \{s_1\}\ \texttt{else}\ \{s_2\} \\
\text{Expression } e & ::= & v \mid c \mid e_1 \oplus e_2 \ \ (\oplus \in \{+, -, \times\}) \\
\text{Predicate } p & ::= & e_1 \oslash e_2 \ \ (\oslash \in \{<, >, =\}) \\
& \mid & p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p
\end{array}
$$

Figure 2: Programming language used for formalization. The notation $\bar{s}$ denotes a sequence $s_1, \ldots, s_n$.

include sequencing, assignments, heap reads and writes, memory allocation, procedure calls, assertions, assumptions, and conditionals. Since loops can be expressed as tail-recursive procedures, we do not introduce an additional loop construct. Also, observe that we only allow conditionals with non-deterministic predicates, denoted $\star$. However, a conditional of the form `if` (p) $\{s_1\}$ `else` $\{s_2\}$ can be expressed as follows in this language:

$$\texttt{if}\ (\star)\ \{\texttt{assume}\ p; s_1\}\ \texttt{else}\ \{\texttt{assume}\ \neg p; s_2\}$$

Since the language is quite standard, we do not present its operational semantics in detail. However, as explained in Section 3, we assume that the execution of a program terminates as soon as we encounter an assertion or assumption violation (i.e., the predicate evaluates to false). As in Section 3, we use the term *failing execution* to indicate a program run with an assertion violation.

4.2 Safety Condition Inference

Recall from Section 1, that a *safety condition* at a given program point π is a sufficient condition for any execution starting at π to be error free. More precisely, a safety condition for a (terminating) statement s is a formula φ such that $\varphi \Rightarrow wp(s, true)$, where $wp(s, \phi)$ denotes the *weakest precondition* of s with respect to postcondition ϕ [37]. While the most precise safety condition is $wp(s, true)$, our analysis intentionally infers stronger safety conditions so that trimming can be used as a pre-processing technique for safety checkers.

Our safety condition inference engine is formalized using the rules shown in Figure 3. Our formalization makes use of an "oracle" Λ for resolving queries about pointer aliasing and procedure side effects. For instance, this oracle can be implemented using a scalable pointer analysis, such as the Data Structure Analysis (DSA) method of Lattner et al. [68]. In the rest of this section, we assume that the oracle for resolving aliasing queries is *flow-insensitive.*

Figure 3 includes two types of inference rules, one for statements and one for procedures. Both forms of judgments utilize a *summary environment* Υ that maps each procedure *prc* to its corresponding safety condition (or "summary"). Since our programming language contains recursive procedures, we would, in general, need to perform a fixed-point computation to obtain sound and precise summaries. However, because our analysis initializes summaries conservatively, the analysis can terminate at any point to produce sound results.

$$(1)\quad \frac{\Lambda, \Upsilon, \Phi \vdash s_2 : \Phi_2 \qquad \Lambda, \Upsilon, \Phi_2 \vdash s_1 : \Phi_1}{\Lambda, \Upsilon, \Phi \vdash s_1; s_2 : \Phi_1}$$

$$(2)\quad \frac{\Phi' \equiv \Phi[e/v]}{\Lambda, \Upsilon, \Phi \vdash v := e : \Phi'}$$

$$(3)\quad \frac{\Phi' \equiv \Phi[drf(v_2)/v_1]}{\Lambda, \Upsilon, \Phi \vdash v_1 := *v_2 : \Phi'}$$

$$(4)\quad \frac{\Phi' \equiv store(drf(v), e, \Lambda, \Phi)}{\Lambda, \Upsilon, \Phi \vdash *v := e : \Phi'}$$

$$(5)\quad \frac{\Phi' \equiv \forall v. \Phi}{\Lambda, \Upsilon, \Phi \vdash v := \texttt{malloc}(e) : \Phi'}$$

$$(6)\quad \frac{\begin{array}{c}\overline{\alpha} \equiv modLocs(prc, \Lambda) \\ \Phi_s \equiv \forall v.\ havoc(\overline{\alpha}, \Lambda, \Phi) \\ \Phi' \equiv \Phi_s \wedge summary(prc, \Upsilon, \overline{v_{act}})\end{array}}{\Lambda, \Upsilon, \Phi \vdash v := \texttt{call}\ \ prc(\overline{v_{act}}) : \Phi'}$$

$$(7)\quad \frac{\Phi' \equiv p \wedge \Phi}{\Lambda, \Upsilon, \Phi \vdash \texttt{assert}\ \ p : \Phi'}$$

$$(8)\quad \frac{\Phi' \equiv p \Rightarrow \Phi}{\Lambda, \Upsilon, \Phi \vdash \texttt{assume}\ \ p : \Phi'}$$

$$(9)\quad \frac{\begin{array}{c}\Lambda, \Upsilon, \Phi \vdash s_1 : \Phi_1 \\ \Lambda, \Upsilon, \Phi \vdash s_2 : \Phi_2 \\ \Phi' \equiv \Phi_1 \wedge \Phi_2\end{array}}{\Lambda, \Upsilon, \Phi \vdash \texttt{if}\ (\star)\ \{s_1\}\ \texttt{else}\ \{s_2\} : \Phi'}$$

$$(10)\quad \frac{\begin{array}{c}\Lambda, \Upsilon, true \vdash s : \Phi \\ \Upsilon' \equiv \Upsilon[prc \mapsto \Phi]\end{array}}{\Lambda, \Upsilon \vdash \texttt{proc}\ \ prc(\overline{v_{in}}) : v_{out}\ \ \{s\} : \Upsilon'}$$

Figure 3: Inference rules for computing safety conditions.

With the exception of rule (10), all rules in Figure 3 derive judgments of the form $\Lambda, \Upsilon, \Phi \vdash s : \Phi'$. The meaning of this judgment is that, using environments Λ and Υ, it is provable that $\{\Phi'\}s\{\Phi\}$ is a valid Hoare triple (i.e., $\Phi' \Rightarrow wp(s, \Phi)$ if s terminates). Similarly to the computation of standard weakest preconditions [37], our analysis propagates safety conditions backward but sacrifices precision to improve scalability. In the following discussion, we only focus on those rules where our inference engine differs from standard precondition computation.

Heap reads and writes. An innovation underlying our safety condition inference is the handling of the heap. Given a store operation $*v := e$, this statement can modify the value of all expressions $*x$, where x is an alias of v. Hence, a sound way to model the heap is to rewrite $*v := e$ as

$$*v := e; \texttt{if}\ \ (v = v_1)\ *v_1 := e; \ldots; \texttt{if}\ \ (v = v_k)\ *v_k := e;$$

where $v_1, \ldots, v_k$ are potential aliases of v. Effectively, this strategy accounts for the "side effects" of statement $*v := e$ to other heap locations by explicitly introducing additional statements. These statements are of the form if $(v = v_i)\ *v_i := e$, i.e., if v and v_i are indeed aliases, then change the value of expression $*v_i$ to e.

While the strategy outlined above is sound, it unfortunately conflicts with our goal of computing safety conditions using lightweight analysis. In particular, since we use a coarse, but scalable alias analysis, most pointers have a large number of possible aliases in practice. Hence, introducing a linear number of conditionals causes a huge blow-up in the size of the safety conditions computed by our technique. To prevent this blow-up, our inference engine computes a safety precondition that is stronger than necessary by using the following conservative *store* operation.

Definition 4.1. **(Memory location)** We represent memory locations using terms that belong to the following grammar:

$$\textit{Memory location } \alpha := v \mid \textit{drf}(\alpha)$$

Here, v represents any program variable, and *drf* is an uninterpreted function representing the dereference of a memory location.

To define our conservative store operation, we make use of a function $\textit{aliases}(v, \Lambda)$ that uses oracle Λ to retrieve all memory locations α that may alias v.

Definition 4.2. **(Store operation)** Let $\textit{derefs}(\Phi)$ denote all α' for which a sub-term $\textit{drf}(\alpha')$ occurs in formula Φ. Then,

$$\textit{store}(\textit{drf}(\alpha), e, \Lambda, \Phi) := \Phi[e/\textit{drf}(\alpha)] \land \bigwedge_{\alpha_i \in A \setminus \{\alpha\}} \alpha_i \neq \alpha$$
$$\text{where } A \equiv \textit{aliases}(\alpha, \Lambda) \cap \textit{derefs}(\Phi)$$

In other words, we compute the precondition for statement $*v := e$ as though the store operation was a regular assignment, but we also "assert" that v is distinct from every memory location α_i that can potentially alias v. To see why this is correct, observe that $\Phi[e/\textit{drf}(v)]$ gives the weakest precondition of $*v := e$ when v does not have any aliases. If v does have aliases that are relevant to the safety condition, then the conjunct $\bigwedge_{\alpha_i \in A \setminus \{v\}} \alpha_i \neq v$ evaluates to *false*, meaning that we can never guarantee the safety of the program. Thus, $\textit{store}(\textit{drf}(v), e, \Lambda, \Phi)$ logically implies $\textit{wp}(*v := e, \Phi)$.

Example 4.3. Consider the following code snippet:

```
if (★) {assume $x = y$; $a := 3$; }
else  {assume $x \neq y$; $*y := 3$; }
$*x := a$; $t := *y$;
assert $t = 3$;
```

Right before the heap write $*x := a$, our analysis infers the safety condition $\textit{drf}(y) = 3 \land x \neq y$. Before the heap write $*y := 3$, the safety condition is $x \neq y$, which causes the condition before the assumption assume $x \neq y$ to be *true*. This means that executions through the else branch are verified and may be trimmed because x and y are not aliases for these executions.

Interprocedural analysis. We now turn our attention to the handling of procedure calls. As mentioned earlier, we perform interprocedural analysis in a modular way, computing summaries for each procedure. Specifically, a summary $\Upsilon(f)$ for procedure f is a sufficient condition for any execution of f to be error free.

With this intuition in mind, let us consider rule (6) for analyzing procedure calls of the form $v :=$ call $\textit{prc}(\bar{e})$. Suppose that $\bar{\alpha}$ is the set of memory locations modified by the callee *prc* but expressed in terms of the memory locations in the *caller*. Then, similarly to other modular interprocedural analyses [8, 9], we conservatively model the effect of the statement $v :=$ call $\textit{prc}(\overline{v_{act}})$ as follows:

```
assert $\textit{summary}(\textit{prc})$;
havoc $v$; havoc $\bar{\alpha}$;
```

Here, havoc α denotes a statement that assigns an unknown value to memory location α. Hence, our treatment of procedure calls asserts that the safety condition for *prc* holds before the call and that the values of all memory locations modified in *prc* are "destroyed".

While our general approach is similar to prior techniques on modular analysis [8, 9], there are some subtleties in our context to which we would like to draw the reader's attention. First, since our procedure summaries (i.e., safety conditions) are not provided by the user, but instead inferred by our algorithm (see rule (10)), we must be conservative about how summaries are "initialized". In particular, because our analysis aims to be lightweight, we do not want to perform an expensive fixed-point computation in the presence of recursive procedures. Therefore, we use the following *summary* function to yield a conservative summary for each procedure.

Definition 4.4. **(Procedure summary)** Let $\textit{hasAsrts}(f)$ be a predicate that yields *true* iff procedure f or any of its (transitive) callees contain an assertion. Then,

$$\textit{summary}(f, \Upsilon, \bar{v}) = \begin{cases} \Upsilon(f)[\bar{v}/\overline{v_{in}}] & \textit{if } f \in \textit{dom}(\Upsilon) \\ \textit{false} & \textit{if hasAsrts}(f) \\ \textit{true} & \textit{otherwise} \end{cases}$$

In other words, if procedure f is in the domain of Υ (meaning that it has previously been analyzed), we use the safety condition given by $\Upsilon(f)$, substituting formals by the actuals. However, if f has not yet been analyzed, we then use the conservative summary *false* if f or any of its callees have assertions, and *true* otherwise. Observe that, if f is not part of a strongly connected component (SCC) in the call graph, we can always obtain the precise summary for f by analyzing the program bottom-up. However, if f is part of an SCC, we can still soundly analyze the caller by using the conservative summaries given by $\textit{summary}(f, \Upsilon, \bar{v})$.

The other subtlety about our interprocedural analysis is the particular way in which havocking is performed. Since the callee may modify heap locations accessible in the caller, we define a *havoc* operation that uses the *store* function from earlier to conservatively deal with memory locations.

Definition 4.5. **(Havoc operation)**

$$\textit{havoc}(\textit{drf}(\alpha), \Lambda, \Phi) := \forall v_{new}.\ \textit{store}(\textit{drf}(\alpha), v_{new}, \Lambda, \Phi)$$
$$\text{where } v_{new} \notin \textit{freeVars}(\Phi)$$
$$\textit{havoc}(\overline{\alpha}, \Lambda, \Phi) := \textit{havoc}(\textit{tail}(\overline{\alpha}), \Lambda, \textit{havoc}(\textit{head}(\overline{\alpha}), \Lambda, \Phi))$$

Observe that the above definition differs from the standard way this operation is typically defined [8]. In particular, given a scalar variable v, the assignment $v := \star$, and its postcondition ϕ, the standard way to compute a conservative precondition for the assignment is $\forall v.\phi$ (i.e., ϕ must hold for *any* value of v). Note that an alternative way of computing the precondition is $\forall x.\phi[x/v]$, where x is not a free variable in ϕ. In the context of scalars, these two definitions are essentially identical, but the latter view allows us to naturally extend our definition to heap locations by using the previously defined *store* function. Specifically, given a heap location $drf(\alpha)$ modified by the callee, we model the effect of this modification as $\forall v_{new}.\ store(drf(\alpha), v_{new}, \Lambda, \Phi)$.

Theorem 4.6. *Suppose that* $\Lambda, \Upsilon, \Phi \vdash s : \Phi'$*, and assume that* Λ *provides sound information about aliasing and procedure side effects. Then, under the condition that* s *terminates and that the summaries provided by* Υ *are sound, we have* $\Phi' \implies wp(s, \Phi)$*.* [1]

5 PROGRAM INSTRUMENTATION

In the previous section, we discussed how to infer safety conditions for each program point. Recall that program trimming annotates the code with *trimming conditions*, which are necessary conditions for failure. Here, we describe how we instrument the program with suitable assumptions that preserve safety of the original program.

Intraprocedural instrumentation. First, let us ignore procedure calls and consider instrumenting a single procedure in isolation. Specifically, consider a procedure with body $s_1; \ldots; s_n$ and let:

$$\Lambda, \Upsilon, true \vdash s_i; \ldots; s_n : \Phi$$

We instrument the program with the statement `assume` $\neg\Phi$ right before statement s_i if s_i complies with the instrumentation strategy specified by the user (see Section 6). In general, note that we do not instrument at every single instruction because subsequent safety checkers must also analyze the assumptions, which adds overhead to their analysis.

Theorem 5.1. *Suppose that our technique adds a statement* `assume` Φ *before* $s_i; \ldots; s_n$*. Then,* Φ *is a necessary condition for* $s_i; \ldots; s_n$ *to have an assertion violation.*

Interprocedural instrumentation. One of the key challenges in performing program instrumentation is how to handle procedure calls. In particular, we cannot simply annotate a procedure f using the safety conditions computed for f. The following example illustrates why such a strategy would be unsound.

Example 5.2. Consider procedures `foo`, `bar`, and `baz`:

```
proc foo(x) {*x := 2; }
proc bar(a) {x := malloc(a); foo(x); assert a < 100; }
proc baz(b) {x := malloc(b); foo(x); assert b > 10; }
```

Here, the safety condition for procedure `foo` is just *true* since `foo` does not contain assertions or have callees with assertions. However, observe that we cannot simply instrument `foo` with `assume` *false* because there are assertions after the call to `foo` in `bar` and `baz`. One possible solution to this challenge is to only instrument the `main` method, which would be very ineffective. Another possible strategy might be to propagate safety conditions top-down from callers to callees in a separate pass. However, this latter strategy also has some drawbacks. For instance, in this example, variables `a` and `b` are not in scope in `foo`; hence, there is no meaningful instrumentation we could add to `foo` short of `assume` *true*, which is the same as having no instrumentation at all.

We solve this challenge by performing a program transformation inspired by previous work [53, 66]. The key idea underlying this program transformation is to create, for each procedure *prc*, a new procedure *prc'* that can never fail. In particular, we create *prc'* by (a) changing all assertions `assert` ϕ in *prc* to `assume` ϕ, and (b) replacing all calls to f (including recursive ones) with f'. Now, given a call site of *prc*, v `:= call` $prc(\bar{e})$, we replace it with the following conditional:

```
if (★) {v := call prc'(ē); }
else   {v := call prc(ē); assume false; }
```

This transformation is semantics preserving since it is merely a case analysis: Either *prc* succeeds, in which case it is safe to replace the call to *prc* with *prc'*, or it fails, in which case we can call original *prc* but add `assume` *false* afterward since *prc* has failed. The following example illustrates this transformation.

Example 5.3. Consider the following procedures:

```
proc foo(x, y)  {assert x > 0; bar(y); }
proc bar(z)     {assert z > 0; }
proc main(x, y) {foo(x, y); }
```

Our transformation yields the following new program:

```
proc foo'(x, y) {assume x > 0; bar'(y); }
proc foo(x, y)  {
  assert x > 0;
  if (★) {bar'(y); }
  else   {bar(y); assume false; }
}
proc bar'(z)    {assume z > 0; }
proc bar(z)     {assert z > 0; }
proc main(x, y) {
  if (★) {foo'(x, y); }
  else   {foo(x, y); assume false; }
}
```

The main advantage of this transformation is that it allows us to perform program instrumentation in a modular and conceptually simple way. In particular, we do not need to instrument the "safe" version *prc'* of a procedure *prc* since *prc'* never fails. On the other hand, it is safe to instrument *prc* with the negation of the local safety conditions since every call site of *prc* is followed by the statement `assume` *false* (i.e., execution terminates immediately after the call).

Example 5.4. Consider the following procedures `foo` and `bar`:

```
proc foo(x)    {assert x > 10; }
proc bar(a, x) {foo(x); assert a < 100; }
```

Our instrumentation yields the following new program:

```
proc foo'(x)  {assume x > 10; }
proc foo(x)   {assume x ≤ 10; assert x > 10; }
proc bar(a, x) {
  assume a ≥ 100 ∨ x ≤ 10;
  if (★) {foo'(x); }
  else   {foo(x); assume false; }
  assert a < 100;
}
```

Discussion. The reader may notice that our program transformation introduces additional branches that did not exist in the original program. Since the goal of program trimming is to reduce the number of execution paths while retaining equi-safety, this

[1]Proofs or proof sketches for all theorems can be found in the appendix.

transformation may seem counter-intuitive. However, because one of the branches is always followed by `assume` *false*, our transformation does not lead to a blow-up in the number of paths and allows us to perform the instrumentation modularly.

6 IMPLEMENTATION

We have implemented our program trimming algorithm as a tool called TRIMMER, meant as a lightweight pre-processor for program analyzers that check safety. Our implementation is based on the LLVM infrastructure [67] and performs instrumentation at the LLVM bit-code level. Hence, TRIMMER can be conveniently integrated into any safety checking tool that is built on top of the LLVM infrastructure and is capable of analyzing `assume` statements.

Recall from Section 4 that TRIMMER's safety inference engine requires alias and side effect information to soundly analyze heap stores and procedure calls. For this purpose, TRIMMER leverages LLVM's DSA pointer analysis [68], a highly-scalable, summary-based, flow-insensitive analysis.

Since TRIMMER can be useful to a variety of program analysis tools (including both static and dynamic analyzers), TRIMMER can be customized in different ways depending on the assumptions made by subsequent safety checkers. In what follows, we describe the different configurations that TRIMMER provides.

Reasoning about integer arithmetic. TRIMMER provides the option of treating integral-type expressions either as mathematical (unbounded) or fixed-width integers. Since some safety checkers ignore integer over- and under-flows but others do not, TRIMMER supports both encodings.[2] Analyzers treating values as mathematical integers can therefore use the configuration of TRIMMER that also makes this same unsound assumption.

Eliminating quantifiers. Recall from Section 4 that the safety conditions generated by our inference engine contain universal quantifiers. Hence, when negating the safety conditions, the resulting trimming conditions contain existentially-quantified variables. TRIMMER provides two alternatives for eliminating quantifiers. First, TRIMMER can remove quantifiers using Z3's quantifier elimination (QE) capabilities [36] after simplifying and pre-processing the formula. Second, TRIMMER also allows replacing quantified variables by calls to non-deterministic functions. Since quantified variables at the formula level correspond to program variables with unknown values, this strategy has the same effect as quantifier elimination.

Bounding the instrumentation. After TRIMMER instruments the program with trimming conditions, subsequent safety checkers need to analyze the assumptions. Hence, the number of additional `assume` statements as well as the *size* of the predicates can affect the running time of program analyzers. For this reason, TRIMMER allows users to customize where to add assumptions in the code. For example, sensible strategies include adding instrumentation right before loops and procedure calls, or before every conditional.

In a similar vein, TRIMMER also provides different options for bounding the size of the formulas used in `assume` statements. For example, the user can bound the number of conjuncts in the formula to be at most k, where k is a value chosen by the user. This strategy is sound because TRIMMER guarantees that the "simplified" formulas are weaker than the original trimming conditions.

[2]For the fixed-width integer encoding, TRIMMER strengthens safety conditions by requiring that there are no integer over- or under-flows. Specifically, TRIMMER utilizes arithmetic operations in the LLVM instruction set that return both the result of the operation and a flag indicating whether an over-flow occurred. Note that TRIMMER does not use bit-vectors for encoding fixed-width integers.

7 EXPERIMENTS

To evaluate the effectiveness of program trimming, we have used TRIMMER to pre-process hundreds of programs by instrumenting them with `assume` statements. Since these assumptions are not useful on their own, we evaluate the effect of program trimming in the context of two different LLVM-based program analyzers for safety checking. In particular, we use CRAB, an abstract interpreter that supports several abstract domains, and KLEE, a widely-used dynamic symbolic execution engine.

We ran our experiments on 439 programs[3], most of which (92%) are taken from the software verification competition (SV-COMP) benchmarks, which have clearly defined outcomes and are handled by numerous tools. Since the errors in many of the buggy programs in this benchmark set are very shallow[4], we also augment these benchmarks with additional buggy programs, either taken from other sources or obtained by injecting deeper bugs into safe SV-COMP benchmarks. The benchmarks taken from SV-COMP span a broad range of categories, including CONTROLFLOW, LOOPS, RECURSIVE, and ARRAYREACH, but exclude categories that are not handled by KLEE or CRAB, e.g., BITVECTORSREACH, CONCURRENCY.

In what follows, we describe the effects of program trimming on the results of CRAB and KLEE. We ran all of our experiments on an Intel Xeon CPU E5-2640 v3 @ 2.60GHz machine with 132 GB of memory running the Ubuntu 14.04.1 operating system. We used the latest available version of CRAB and the latest version of KLEE that was compatible with LLVM 3.6, which CRAB requires.

7.1 Impact of Program Trimming on CRAB

To demonstrate that program trimming increases precision across a range of abstract domains, we compare the performance of CRAB (with and without trimming) on three different domains with varying levels of precision:

- **Int** denotes the (non-relational) interval domain [31], which infers invariants of the form $c_1 \leq x \leq c_2$;
- **Zones** is the (relational) zones abstract domain [76], which infers difference constraints of the form $x - y \leq c$;
- **RTZ** is CRAB's most precise (native) abstract domain and corresponds to the reduced product of disjunctive intervals (i.e., disjunctions of constraints of the form $c_1 \leq x \leq c_2$) [45] and the zones abstract domains.

As mentioned in Section 6, TRIMMER can be customized using a variety of different configurations. To understand the precision vs. performance trade-off, we evaluate CRAB using the configurations of TRIMMER shown in Table 1. Here, the column labeled MC indicates the maximum number of conjuncts used in an `assume` statement. The third column labeled QE indicates whether we use quantifier elimination or whether we model quantified variables using calls to non-deterministic functions (recall Section 6). Finally, the columns labeled L/P and C denote the instrumentation strategy.

[3]Available at: https://mariachris.github.io/FSE2017/benchmarks.zip

[4]For example, in the existing SV-COMP benchmarks, KLEE can find the bug with a very low resource limit for 85% of the buggy programs.

Table 1: Overview of trimming configurations (incl. total number of added assume statements and time for pre-processing all benchmarks in the two right-most columns).

Configuration	MC	QE	L/P	C	A	Time (s)
Trim$_{L+B}$	4	✓	✓	✗	143	5.31
Trim$_{B}$	4	✓	✓	✓	1638	4.97
Trim$_{ND+B}$	4	✗	✓	✓	2801	7.34
Trim$_{L}$	∞	✓	✓	✗	156	6.05
Trim	∞	✓	✓	✓	1735	5.74
Trim$_{ND}$	∞	✗	✓	✓	2852	8.62

In configurations where there is a checkmark under L/P, we add `assume` statements right before loops (L) and before procedure (P) calls. In configurations where there is a checkmark under C, we also add instrumentation before every conditional. The two right-most columns show the total number of added `assume` statements (not trivially *true*) and the pre-processing time for all benchmarks. Since average trimming time is 11–20 milliseconds per benchmark, we see that program trimming is indeed very lightweight.

The results of our evaluation are summarized in Table 2. As we can see from this table, all configurations of program trimming improve the precision of Crab, and these improvements range from 23% to 54%. For instance, for the interval domain, the most precise configuration of Trimmer allows the verification of 68 benchmarks instead of only 49 when using Crab without trimming.

Another observation based on Table 2 is the precision vs. performance trade-offs between different configurations of Trimmer. Versions of Crab that use Trimmer with QE seem to be faster and more precise than those configurations of Trimmer without QE. In particular, the version of Trimmer with QE performs better because there are fewer variables for the abstract domain to track. We also conjecture that Trimmer using QE is more precise because the abstract domain can introduce imprecision when reasoning about logical connectives. For instance, consider the formula $\exists x.(x = 1 \land x \neq 1)$, which is logically equivalent to *false*, so Trimmer with QE would instrument the code with `assume` *false*. However, if we do not use QE, we would instrument the code as follows:

$$x := \texttt{nondet()}; \texttt{assume}\ \ x = 1 \land x \neq 1;$$

When reasoning about the `assume` statement, an abstract interpreter using the interval domain takes the meet of the intervals $[1, 1]$ and $\top$, which yields $[1, 1]$. Hence, using Trimmer without QE, Crab cannot prove that the subsequent code is unreachable.

Summary. Table 2 shows that trimming significantly improves the precision of an abstract interpreter with reasonable overhead. Our cheapest trimming configuration (**Trim**$_{L+B}$ **+ Int**) proves 21% more programs safe than the most expensive configuration of Crab without trimming (**RTZ**) in *less than 70% of the time.*

7.2 Impact of Program Trimming on Klee

In our second experiment, we evaluate the impact of program trimming on Klee, a state-of-the-art dynamic symbolic execution tool. We use a subset[5] of the variants of Trimmer (see Table 1) and evaluate trimming on Klee with three search strategies: breadth-first search (BFS), depth-first search (DFS), and random search (R).

[5]In particular, since Klee's analysis is already path-sensitive we do not consider variants that instrument before conditionals here.

Table 2: Increased precision of an abstract interpreter due to trimming. Since Crab treats integers as unbounded, our instrumentation also makes this assumption.

Configuration	Safe	Time (s)
Int	49 (+0%)	129 (+0%)
Trim$_{L+B}$ **+ Int**	63 (+29%)	149 (+16%)
Trim$_{B}$ **+ Int**	65 (+33%)	173 (+34%)
Trim$_{ND+B}$ **+ Int**	61 (+24%)	198 (+53%)
Trim$_{L}$ **+ Int**	64 (+31%)	151 (+17%)
Trim + Int	**68 (+39%)**	191 (+48%)
Trim$_{ND}$ **+ Int**	62 (+27%)	227 (+76%)
Zones	52 (+0%)	130 (+0%)
Trim$_{L+B}$ **+ Zones**	66 (+27%)	148 (+14%)
Trim$_{B}$ **+ Zones**	68 (+31%)	195 (+50%)
Trim$_{ND+B}$ **+ Zones**	64 (+23%)	222 (+71%)
Trim$_{L}$ **+ Zones**	67 (+29%)	150 (+15%)
Trim + Zones	**73 (+40%)**	281 (+116%)
Trim$_{ND}$ **+ Zones**	66 (+27%)	320 (+146%)
RTZ	52 (+0%)	215 (+0%)
Trim$_{L+B}$ **+ RTZ**	67 (+29%)	231 (+7%)
Trim$_{B}$ **+ RTZ**	76 (+46%)	535 (+149%)
Trim$_{ND+B}$ **+ RTZ**	66 (+27%)	582 (+171%)
Trim$_{L}$ **+ RTZ**	68 (+31%)	237 (+10%)
Trim + RTZ	**80 (+54%)**	1620 (+653%)
Trim$_{ND}$ **+ RTZ**	67 (+29%)	3330 (+1449%)

Since programs usually have infinitely many execution paths, it is necessary to enforce some resource bounds when running Klee. In particular, we run Klee with a timeout of 300 seconds and a limit of 64 on the number of forks (i.e., symbolic branches).

The results of our evaluation are presented in Table 3. Here, the column labeled Safe shows the number of programs for which Klee explores all execution paths without reporting any errors or warnings.[6] Hence, these programs can be considered verified. The second column, labeled Unsafe, shows the number of programs reported as buggy by each variant of Klee. In this context, a bug corresponds to an explicit assertion violation in the program. Next, the third column, labeled Paths, shows the number of program paths that Klee explored for each variant. Note that fewer paths is better—this means that Klee needs to explore fewer executions before it finds the bug or proves the absence of an assertion violation. The next two columns measure the number of programs for which each Klee variant reaches a resource limit. In particular, the column labeled Timeout shows the number of programs for which Klee fails to terminate within the 5-minute time limit. Similarly, the column Max-Forks indicates the number of programs for which each Klee variant reaches the limit that we impose on the number of forks. Finally, the last column, labeled Time, shows the total running time of each Klee variant on all benchmarks.

As shown in Table 3, program trimming increases the number of programs that can be proved safe by 16–18%. Furthermore, program trimming allows Klee to find up to 30% more bugs within the given resource limit. In addition, Klee with program trimming needs to explore significantly fewer paths (up to 38%) and reaches the resource bound on significantly fewer programs. Finally, observe that the overall running time of Klee decreases by up to 30%.

[6]By warning, we mean any internal Klee warning that designates an incompleteness in Klee's execution (e.g., solver timeouts and concretizing symbolic values).

Table 3: Summary of comparison with Klee. Since Klee treats integers in a sound way, we also use the variant of Trimmer that reasons about integer over- and under-flows.

Configuration	Safe	Unsafe	Paths	Timeout	Max-Forks	Time (s)
Klee$_{BFS}$	126 (+0%)	118 (+0%)	9231 (+0%)	73 (+0%)	73 (+0%)	21679
Trim$_{L+B}$ + Klee$_{BFS}$	146 (+16%)	145 (+23%)	5978 (-35%)	52 (-40%)	46 (-51%)	15558
Trim$_{L}$ + Klee$_{BFS}$	**146 (+16%)**	**153 (+30%)**	**5678 (-38%)**	**50 (-32%)**	**40 (-45%)**	**15264**
Klee$_{DFS}$	126 (+0%)	99 (+0%)	10024 (+0%)	91 (+0%)	75 (+0%)	26185
Trim$_{L+B}$ + Klee$_{DFS}$	146 (+16%)	124 (+25%)	6939 (-31%)	72 (-21%)	48 (-36%)	20797
Trim$_{L}$ + Klee$_{DFS}$	**146 (+16%)**	**129 (+30%)**	**6695 (-33%)**	**72 (-21%)**	**43 (-43%)**	**21164**
Klee$_{R}$	126 (+0%)	121 (+0%)	9227 (+0%)	71 (+0%)	72 (+0%)	21077
Trim$_{L+B}$ + Klee$_{R}$	149 (+18%)	146 (+21%)	5967 (-35%)	49 (-31%)	44 (-39%)	14844
Trim$_{L}$ + Klee$_{R}$	**149 (+18%)**	**152 (+26%)**	**5699 (-38%)**	**48 (-32%)**	**40 (-44%)**	**14850**

Figure 4 compares the number of benchmarks solved by the original version of Klee (using BFS) with its variants using program trimming. Specifically, the x-axis shows how many benchmarks were solved (i.e., identified as safe or unsafe) by each variant (sorted by running time), and the y-axis shows the corresponding running time per benchmark. For instance, we can see that **Trim**$_{L}$ + Klee$_{BFS}$ solves 246 benchmarks within less than one second each, whereas the original version of Klee only solves 203 benchmarks.

Summary. Overall, the results shown in Table 3 and Figure 4 demonstrate that program trimming significantly improves the effectiveness and performance of a mature, state-of-the-art symbolic execution tool. In particular, program trimming allows Klee to find more bugs and prove more programs correct within a given resource limit independently of its search strategy.

7.3 Threats to Validity

We identified these threats to the validity of our experiments:

- *Sample size*: We used 439 programs, most of which, however, are taken from the SV-COMP benchmarks, a widely-used and established set of verification tasks.
- *Safety checkers*: We evaluate our technique using two safety checkers, which, however, are mature and representative of two program analysis techniques.
- *Trimming configurations*: We only presented experiments using a selection of the different configurations that Trimmer provides (see Section 6). However, all of these configurations are orthogonal to each other, and we evaluated a large variety of them to demonstrate the benefits of our technique.

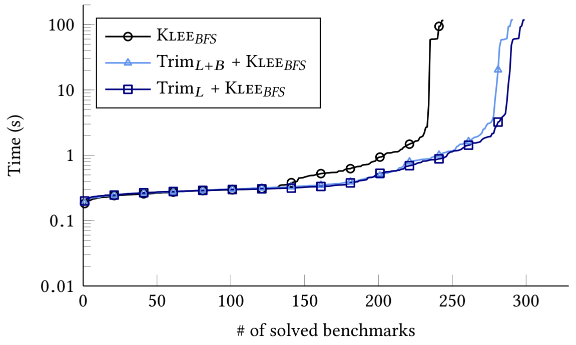

Figure 4: Quantile plot of time and solved benchmarks for selected Klee variants.

8 RELATED WORK

The ideas in this paper are related to a long line of previous work on program transformations and safety precondition inference.

Program slicing. One of the most well-known program simplification techniques is *program slicing*, which removes program statements that are not relevant to some criterion of interest (e.g., value of a variable at some program point) [1, 15, 84, 85]. A program slice can be computed either statically or dynamically and includes both forward and backward variants. Program trimming differs from traditional program slicing in two ways: first, trimming focuses on removing execution paths as opposed to statements; second, it is meant as a pre-processing technique for safety checkers rather than a transformation to aid program understanding. In particular, a typical slicing tool may not produce compilable and runnable code that could be consumed by subsequent safety checkers.

More semantic variants of program slicing have also been considered in later work [5, 20, 30, 46, 54, 64]. For instance, Jhala and Majumdar propose *path slicing* to improve the scalability of software model checkers [64]. In particular, path slicing eliminates all operations that are irrelevant toward the reachability of the target location in a given program path. Unlike program trimming, path slicing is not used as a pre-processing step and works on a single program path that corresponds to a counterexample trace.

Prior work has also considered how to slice the program with respect to a predicate [20, 30, 46, 54]. Such techniques can be useful for program understanding, for example, when the user only wants to see statements that affect a given condition (e.g., the predicate of a conditional). In contrast, program trimming is not meant as a program understanding technique and removes program paths that are irrelevant for a given safety property. Furthermore, the trimmed program is not meant for human consumption, as it semantically prunes program paths through the insertion of `assume` statements.

In general, slicing has been used before invoking a program analyzer [22, 23, 44, 55, 62, 63, 75]. A key difference with these approaches is that the result of trimming is valid code, which compiles and runs, instead of an abstract representation, such as a control flow graph or model.

Pre-processing for program analyzers. In the same spirit as this paper, prior work has also used program transformations to improve the precision or scalability of program analyzers [24, 26, 27, 53, 66, 83, 87]. For instance, a transformation for faster goal-directed search [66] moves all assertions to a single main procedure with the goal of speeding up analysis. Another program transformation called *loop splitting* aims to improve the precision of program analyzers by turning *multi-phase* loops into a sequence of *single-phase* loops [83]. However, neither of these techniques instrument the program with assumptions to guide safety checking tools.

Recent techniques rely on the verification results of a full-fledged analyzer, such as an abstract interpreter or a model checker, to guide automatic test case generation tools [24, 26, 34, 35] or other static analyzers [13, 25, 27, 87], some even using slicing as an intermediate step [34]. In contrast, program trimming is more lightweight by not relying on previous analyzers and, thus, can be used as a pre-processing step for any safety checker.

Precondition inference. The use of precondition inference dates back to the dawn of program verification [37]. Most verification techniques infer a sufficient condition for program safety and prove the correctness of the program by showing the validity of this condition [8, 10, 21, 37, 47, 58–60, 77]. In this work, we do not aim to infer the weakest possible safety precondition; instead, we use lightweight, modular static analysis to infer *a* sufficient condition for safety. Furthermore, we use safety conditions to prune program paths rather than to verify the program.

Program trimming hinges on the observation that the negation of a sufficient condition for property P yields a necessary condition for the negation of P. Prior program analysis techniques also exploit the same observation [39–41, 89]. For instance, this duality has been used to perform modular path-sensitive analysis [39] and strong updates on elements of unbounded data structures [40, 41].

While most program analysis techniques focus on the inference of sufficient preconditions to guarantee safety, some techniques also infer *necessary preconditions* [32, 33, 72, 73, 78]. For example, Verification Modulo Versions (VMV) infers both necessary and sufficient conditions and utilizes previous versions of the program to reduce the number of warnings reported by verifiers [73]. Similarly, necessary conditions are inferred to repair the program in such a way that the repair does not remove any "good" traces [72]. Finally, the techniques described by Cousot et al. infer necessary preconditions, which are used to improve the effectiveness of the Code Contracts abstract interpreter [32, 33, 45].

Abductive reasoning. There has been significant work on program analysis using abductive reasoning, which looks for a *sufficient* condition that implies a desired goal [3, 19, 38, 42, 43, 70, 90]. Our analysis for computing safety conditions can be viewed as a form of abductive reasoning in that we generate sufficient conditions that are stronger than necessary for ensuring safety. However, we perform this kind of reasoning in a very lightweight way without calling an SMT solver or invoking a logical decision procedure.

Modular interprocedural analysis. The safety condition inference we have proposed in this paper is modular in the sense that it analyzes each procedure independently of its callers. There are many previous techniques for performing modular (summary-based) analysis [2, 19, 39, 80, 88]. Our technique differs from these approaches in several ways: First, our procedure summaries only contain safety preconditions, but not post-conditions, as we handle procedure side effects in a very conservative way. Second, we do not perform fixed-point computations and achieve soundness by initializing summaries to *false*. Finally, we use summary-based analysis for program transformation rather than verification.

Property-directed program analysis. There is a significant body of work that aims to make program analyzers property directed. Many of these techniques, such as BLAST [12, 56, 57], SLAM [4, 6, 7], and YOGI [51, 79] rely on counterexample-guided abstraction refinement (CEGAR) [29] to iteratively refine an analysis based on counterexample traces. Another example of a property-directed analysis is the IC3/PDR algorithm [16, 61], which iteratively performs forward and backward analysis for bounded program executions to decide reachability queries. Although abstract interpretation is traditionally not property directed, there is recent work [81] on adapting and rephrasing IC3/PDR in the framework of abstract interpretation. In contrast, we propose a general pre-processing technique to make any eager program analysis property directed.

Path-exploration strategies. Most symbolic execution and testing techniques utilize different strategies to explore the possible execution paths of a program. For example, there are strategies that prioritize "deeper paths" (in depth-first search), "less-traveled paths" [71], "number of new instructions covered" (in breadth-first search), "distance from a target line" [74], or "paths specified by the programmer" [82]. In the context of symbolic execution, program trimming can be viewed as a search strategy that prunes safe paths and steers exploration toward paths that are more likely to contain bugs. However, as shown in our experiments, our technique is beneficial independently of a particular search strategy.

9 CONCLUSION

In this paper, we have proposed *program trimming*, a program simplification technique that aims to reduce the number of execution paths while preserving safety. Program trimming can allow any safety checker to be goal directed by pruning execution paths that cannot possibly result in an assertion violation. Furthermore, because our proposed trimming algorithm is very lightweight, it can be used as an effective pre-processing tool for many program analyzers. As shown by our evaluation, program trimming allows an abstract interpreter, namely Crab, to verify 21% more programs while cutting running time by 30%. Trimming also allows Klee, a dynamic symbolic execution engine, to find more bugs and verify more programs within a given resource limit.

In future work, we plan to investigate the impact of program trimming on other kinds of program analyzers, such as bounded model checkers. We also plan to investigate alternative program trimming algorithms and strategies.

ACKNOWLEDGMENTS

We would like to thank Cristian Cadar and Martin Nowack for their help with Klee. We would also like to thank the main developer of Crab, Jorge Navas, for his help with Crab and for his positive feedback on the usefulness of program trimming. We are grateful to Microsoft for partly funding the first three authors and to the anonymous reviewers for their constructive feedback. This work is supported by AFRL Award FA8750-15-2-0096 and NSF Award #1453386. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] Hiralal Agrawal and Joseph Robert Horgan. 1990. Dynamic Program Slicing. In *PLDI*. ACM, 246–256.

[2] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. 2007. An Overview of the Saturn Project. In *PASTE*. ACM, 43–48.

[3] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. In *POPL*. ACM, 789–801.

[4] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *PLDI*. ACM, 203–213.

[5] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *POPL*. ACM, 97–105.

[6] Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *CAV (LNCS)*, Vol. 2102. Springer, 260–264.

[7] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *POPL*. ACM, 1–3.

[8] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (LNCS)*, Vol. 4111. Springer, 364–387.

[9] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: The Spec# Experience. *CACM* 54 (2011), 81–91. Issue 6.

[10] Mike Barnett and K. Rustan M. Leino. 2005. Weakest-Precondition of Unstructured Programs. In *PASTE*. ACM, 82–87.

[11] Dirk Beyer. 2017. Competition on Software Verification (SV-COMP). (2017). https://sv-comp.sosy-lab.org.

[12] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The Software Model Checker BLAST: Applications to Software Engineering. *STTT* 9 (2007), 505–525. Issue 5.

[13] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. 2012. Conditional Model Checking: A Technique to Pass Information between Verifiers. In *FSE*. ACM, 57–67.

[14] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS (LNCS)*, Vol. 1579. Springer, 193–207.

[15] David Binkley and Keith Brian Gallagher. 1996. Program Slicing. *Advances in Computers* 43 (1996), 1–50.

[16] Aaron R. Bradley. 2011. SAT-Based Model Checking Without Unrolling. In *VMCAI (LNCS)*, Vol. 6538. Springer, 70–87.

[17] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX, 209–224.

[18] Cristian Cadar and Dawson R. Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN (LNCS)*, Vol. 3639. Springer, 2–23.

[19] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction. In *POPL*. ACM, 289–300.

[20] Gerardo Canfora, Aniello Cimitile, and Andrea de Lucia. 1998. Conditioned Program Slicing. *IST* 40 (1998), 595–607. Issue 11–12.

[21] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A Powerful Approach to Weakest Preconditions. In *PLDI*. ACM, 363–374.

[22] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. 2012. Program Slicing Enhances a Verification Technique Combining Static and Dynamic Analysis. In *SAC*. ACM, 1284–1291.

[23] Yunja Choi, Mingyu Park, Taejoon Byun, and Dongwoo Kim. 2015. Efficient Safety Checking for Automotive Operating Systems Using Property-Based Slicing and Constraint-Based Environment Generation. *Sci. Comput. Program.* 103 (2015), 51–70. Issue 1.

[24] Maria Christakis. 2015. *Narrowing the Gap between Verification and Systematic Testing*. Ph.D. Dissertation. ETH Zurich, Switzerland.

[25] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2012. Collaborative Verification and Testing with Explicit Assumptions. In *FM (LNCS)*, Vol. 7436. Springer, 132–146.

[26] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *ICSE*. ACM, 144–155.

[27] Maria Christakis and Valentin Wüstholz. 2016. Bounded Abstract Interpretation. In *SAS (LNCS)*, Vol. 9837. Springer, 105–125.

[28] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *FMSD* 19 (2001), 7–34. Issue 1.

[29] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *CAV (LNCS)*, Vol. 1855. Springer, 154–169.

[30] Joseph J. Comuzzi and Johnson M. Hart. 1996. Program Slicing Using Weakest Preconditions. In *FME (LNCS)*, Vol. 1051. Springer, 557–575.

[31] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

[32] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *VMCAI (LNCS)*, Vol. 7737. Springer, 128–148.

[33] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *VMCAI (LNCS)*, Vol. 6538. Springer, 150–168.

[34] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. 2015. Just Test What You Cannot Verify!. In *FASE (LNCS)*, Vol. 9033. Springer, 100–114.

[35] Przemyslaw Daca, Ashutosh Gupta, Henzinger, and Thomas A. 2016. Abstraction-Driven Concolic Testing. In *VMCAI (LNCS)*, Vol. 9583. Springer, 328–347.

[36] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.

[37] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *CACM* 18 (1975), 453–457. Issue 8.

[38] Isil Dillig and Thomas Dillig. 2013. Explain: A Tool for Performing Abductive Inference. In *CAV (LNCS)*, Vol. 8044. Springer, 684–689.

[39] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, Complete and Scalable Path-Sensitive Analysis. In *PLDI*. ACM, 270–280.

[40] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Fluid Updates: Beyond Strong vs. Weak Updates. In *ESOP (LNCS)*, Vol. 6012. Springer, 246–266.

[41] Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise Reasoning for Programs Using Containers. In *POPL*. ACM, 187–200.

[42] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *OOPSLA*. ACM, 443–456.

[43] Thomas Dillig, Isil Dillig, and Swarat Chaudhuri. 2014. Optimal Guard Synthesis for Memory Safety. In *CAV (LNCS)*, Vol. 8559. Springer, 491–507.

[44] Julian Dolby, Mandana Vaziri, and Frank Tip. 2007. Finding Bugs Efficiently with a SAT Solver. In *ESEC/FSE*. ACM, 195–204.

[45] Manuel Fähndrich and Francesco Logozzo. 2010. Static Contract Checking with Abstract Interpretation. In *FoVeOOS (LNCS)*, Vol. 6528. Springer, 10–30.

[46] John Field, Ganesan Ramalingam, and Frank Tip. 1995. Parametric Program Slicing. In *POPL*. ACM, 379–392.

[47] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *PLDI*. ACM, 234–245.

[48] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2016. An Abstract Domain of Uninterpreted Functions. In *VMCAI (LNCS)*, Vol. 9583. Springer, 85–103.

[49] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2016. Exploiting Sparsity in Difference-Bound Matrices. In *SAS (LNCS)*, Vol. 9837. Springer, 189–211.

[50] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI*. ACM, 213–223.

[51] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. 2010. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *POPL*. ACM, 43–56.

[52] Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *CAV (LNCS)*, Vol. 1254. Springer, 72–83.

[53] Arie Gurfinkel, Ou Wei, and Marsha Chechik. 2008. Model Checking Recursive Programs with Exact Predicate Abstraction. In *ATVA (LNCS)*, Vol. 5311. Springer, 95–110.

[54] Mark Harman, Robert M. Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. 2001. Pre/Post Conditioned Slicing. In *ICSM*. IEEE Computer Society, 138–147.

[55] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. 2000. Slicing Software for Model Construction. *Higher-Order and Symbolic Computation* 13 (2000), 315–353. Issue 4.

[56] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from proofs. In *POPL*. ACM, 232–244.

[57] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *POPL*. ACM, 58–70.

[58] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *CACM* 12 (1969), 576–580. Issue 10.

[59] C. A. R. Hoare. 1971. Procedures and Parameters: An Axiomatic Approach. In *Symposium on Semantics of Algorithmic Languages*. Lecture Notes in Mathematics, Vol. 188. Springer, 102–116.

[60] C. A. R. Hoare and Jifeng He. 1987. The Weakest Prespecification. *Inf. Process. Lett.* 24 (1987), 127–132. Issue 2.

[61] Krystof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *SAT (LNCS)*, Vol. 7317. Springer, 157–171.

[62] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. 2005. F-Soft: Software Verification Platform. In *CAV (LNCS)*, Vol. 3576. Springer, 301–306.

[63] Joxan Jaffar and Vijayaraghavan Murali. 2014. A Path-Sensitively Sliced Control Flow Graph. In *FSE*. ACM, 133–143.

[64] Ranjit Jhala and Rupak Majumdar. 2005. Path Slicing. In *PLDI*. ACM, 38–47.

[65] James C. King. 1976. Symbolic Execution and Program Testing. *CACM* 19 (1976), 385–394. Issue 7.
[66] Akash Lal and Shaz Qadeer. 2014. A Program Transformation for Faster Goal-Directed Search. In *FMCAD*. IEEE Computer Society, 147–154.
[67] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Computer Society, 75–88.
[68] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *PLDI*. ACM, 278–289.
[69] K. Rustan M. Leino. 2005. Efficient Weakest Preconditions. *IPL* 93 (2005), 281–288. Issue 6.
[70] Boyang Li, Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Mooly Sagiv. 2013. Synthesis of Circular Compositional Program Proofs via Abduction. In *TACAS (LNCS)*, Vol. 7795. Springer, 370–384.
[71] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *OOPSLA*. ACM, 19–32.
[72] Francesco Logozzo and Thomas Ball. 2012. Modular and Verified Automatic Program Repair. In *OOPSLA*. ACM, 133–146.
[73] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. 2014. Verification Modulo Versions: Towards Usable Verification. In *PLDI*. ACM, 294–304.
[74] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *SAS (LNCS)*, Vol. 6887. Springer, 95–111.
[75] Lynette I. Millett and Tim Teitelbaum. 2000. Issues in Slicing PROMELA and Its Applications to Model Checking, Protocol Understanding, and Simulation. *STTT* 2 (2000), 343–349. Issue 4.
[76] Antoine Miné. 2004. *Weakly Relational Numerical Abstract Domains. (Domaines Numériques Abstraits Faiblement Relationnels)*. Ph.D. Dissertation. École Polytechnique, Palaiseau, France.
[77] Yannick Moy. 2008. Sufficient Preconditions for Modular Assertion Checking. In *VMCAI (LNCS)*, Vol. 4905. Springer, 188–202.
[78] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. 2012. Abstractions from Tests. In *POPL*. ACM, 373–386.
[79] Aditya V. Nori, Sriram K. Rajamani, Saideep Tetali, and Aditya V. Thakur. 2009. The YOGI Project: Software Property Checking via Static Analysis and Testing. In *TACAS (LNCS)*, Vol. 5505. Springer, 178–181.
[80] Amir Pnueli and Micha Sharir. 1981. Two Approaches to Interprocedural Data Flow Analysis. *Program Flow Analysis: Theory and Applications* (1981), 189–234.
[81] Noam Rinetzky and Sharon Shoham. 2016. Property Directed Abstract Interpretation. In *VMCAI (LNCS)*, Vol. 9583. Springer, 104–123.
[82] Koushik Sen, Haruto Tanno, Xiaojing Zhang, and Takashi Hoshino. 2015. GuideSE: Annotations for Guiding Concolic Testing. In *AST*. IEEE Computer Society, 23–27.
[83] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *CAV (LNCS)*, Vol. 6806. Springer, 703–719.
[84] Frank Tip. 1995. A Survey of Program Slicing Techniques. *J. Prog. Lang.* 3 (1995), 121–189. Issue 3.
[85] Mark Weiser. 1981. Program Slicing. In *ICSE*. IEEE Computer Society, 439–449.
[86] Glynn Winskel. 2012. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.
[87] Valentin Wüstholz. 2015. *Partial Verification Results*. Ph.D. Dissertation. ETH Zurich, Switzerland.
[88] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating Precise and Concise Procedure Summaries. In *POPL*. ACM, 221–234.
[89] Xin Zhang, Mayur Naik, and Hongseok Yang. 2013. Finding Optimum Abstractions in Parametric Dataflow Analysis. In *PLDI*. ACM, 365–376.
[90] Haiyan Zhu, Thomas Dillig, and Isil Dillig. 2013. Automated Inference of Library Specifications for Source-Sink Property Verification. In *APLAS (LNCS)*, Vol. 8301. Springer, 290–306.

A PROOF OF THEOREM 4.6

Proof sketch. For most statements (e.g., assignment, assumption, assertion), Φ' is just the standard weakest precondition of s with respect to Φ.

For heap reads and writes, we already argued why $\Phi' \Rightarrow wp(s, \Phi)$. The heap allocation rule is also correct since it "havocs" the allocated pointer.

The correctness of the procedure call rule follows from the following two facts: First, $summary(prc, \Upsilon, \bar{v})$ is a conservative safety condition for the call to f. In particular, if $f \in dom(\Upsilon)$, this follows from the soundness of Υ. If $f \notin dom(\Upsilon)$, *false* (resp. *true*) is a sufficient condition for the safety of any procedure that does (resp. does not) contain an assertion. Second, we "havoc" the value of any memory location modified in f. The correctness of our *havoc* operation follows from (a) the correctness of the *store* function, and (b) $\forall v.\phi \Rightarrow wp(v := e, \phi)$ for any expression e. □

B PROOF OF THEOREM 5.1

Proof. The proof is by induction on the number of statements (i.e., $n - i$).

Suppose $i = n$. If s_n is not an assertion, then the safety condition is *true*, so we add `assume` *false*. Since s_n can never fail, *false* is indeed necessary for failure. If s_n is `assert` ϕ, then the necessary condition for failure is $\neg\phi$. Since the safety condition for s_n is ϕ, our technique instruments the code with `assume` $\neg\phi$.

For the inductive step, suppose $i < n$ and let:

$$\Lambda, \Upsilon, true \vdash s_{i+1}; \ldots; s_n : \Phi$$

By the inductive hypothesis, $\neg\Phi$ is a necessary condition for the failure of $s_{i+1}, \ldots, s_n$. We consider three cases: (1) s_i is an assertion `assert` ϕ. Then, the necessary condition for the failure of $s_i; \ldots; s_n$ is $\neg\phi \vee \neg\Phi$. Since the safety condition for $s_i; \ldots; s_n$ is $\phi \wedge \Phi$, our technique instruments the code with `assume` $\neg\phi \vee \neg\Phi$. (2) If s_i is an assumption `assume` ϕ, the necessary condition for failure is $\phi \wedge \neg\Phi$, which is exactly the trimming condition computed by our technique. (3) Otherwise, the necessary condition for failure is $wp(s_i, \neg\Phi)$. Suppose $\Lambda, \Upsilon, \Phi \vdash s_i : \Phi'$. By soundness of the safety condition inference, we have $\Phi' \Rightarrow wp(s_i, \Phi)$, and we instrument the code with `assume` $\neg\Phi'$. Since s_i is neither an assertion nor an assumption, we have $wp(s_i, \neg\Phi) \equiv \neg wp(s_i, \Phi)$; thus, $wp(s_i, \neg\Phi) \Rightarrow \neg\Phi'$. □