Rinser: Deriving Succinct Evidence for Automated Code Reviews

Thanh-Toan Nguyen[1⋆], Pavle Subotić[2], and Bor-Yuh Evan Chang[2,3]

[1] National University of Singapore
[2] Amazon, UK
[3] University of Colorado Boulder

Abstract. Automated static analyzers are increasingly part of the code review process. And just like with human code reviewers, a potential issue reported by the analyzer may be important or unimportant (depending on any number of things like context, environment, or abstraction). But unlike with human reviewers, it is often difficult to obtain further clarity on the reasoning behind the warning.

In this industrial case study, we present Rinser, a tool developed at Amazon to ease user interaction with static analysis results presented in code reviews. Our technique is an on-demand algorithm that produces succinct evidence for possible error states. We have integrated Rinser as a post-analysis pass for alarms given by the Infer static analyzer and are able to produce traces on average 10x shorter than Infer traces in under 6 seconds on average due the the removal of irrelevant information.

Keywords: Static Analysis · Debugging · Witnesses

1 Introduction

We consider the problem of producing efficiently-actionable evidence from a static analyzer. That is, we seek to close the gap between static analyzers and their human users by providing precisely the right evidence that enables users to accept or dismiss potential issues reported by the analyzer.

Our efforts have been pivotal in gaining user trust when integrating static analysis into the code review process. The code review process is a traditionally manual inspection of source code by developers other than the author. The code review process is paramount to ensuring quality in software projects at Amazon and beyond [36, 35, 18]. Code reviews provide another pair of eyes to check the correctness, coherence, maintainability of the code, as well as being a medium for team alignment, discussion, and learning. But on the down side, code reviews require significant resources and may result in delays in development. Therefore,

⋆ Work done while a PhD intern at Amazon, UK the use of static analyzers has been suggested as a mechanism to add *automation* to the code review process [2, 18].

The strength of automated code reviews, backed by static analyzers, is in finding cumbersome bugs that could be missed by human reviewers due to *developer fatigue* [37]. However, typical static analyzers are limited in a key dimension. They either provide no explanation to the origin of static analysis alarms or their explanations are often too verbose and obscure for developers to use effectively. Similar conclusions have been observed in [24, 4]. Various studies have conjectured what the ideal static explanation should be [5]. In this work we take a pragmatic approach that has been shown to be successful in an industrial setting. We argue that providing a succinct explanation of relevant code is key to productive developer static analysis alarm tiraging. However, in this regard current state-of-the-art static analyzers fall short. For example, Facebook's Infer provides trace information for each alarm—built alongside the analysis recording its derivations. Our experience has been that these traces are too imprecise and verbose for Amazon developers to effectively triage and action on the alarms. In the case of our industrial use case, namely, the Prime Video application, the average Infer trace provided to developers was 44.5 statements long. However, we found that, on average, only 10% of these statements were relevant to the user—in the sense that they contributed to the state changes to realize the reported bug.

It is important to note that verbose traces have repercussions beyond mere inconvenience or productivity. In working with developers, we have found that even in the case of a true bug (unknown to the developer), if the developer is not provided with a quickly understandable explanation, they are likely to dismiss it as static analyser noise. Thus any effort in increasing precision to reduce actual noise from analysis abstraction can easily be zeroed out if the explanations are not clear to the developer.

In this paper, we present Rinser, a tool developed at Amazon to improve the evidence developers receive from static analyzers. Specifically, Rinser provides an *on-demand* method to compute explanation traces comprised of *relevant* statements, that is, statements that can affect the potential error under investigation and avoids searching paths that are infeasible. The Rinser algorithm builds explanation traces by performing a subsequent backwards, goal-directed static analysis using a symbolic domain comprised of intuitionistic separation logic [32] formulae and path constraints [10]. Specifically, this analysis computes a backwards over-approximation of the error condition, meaning it can soundly refute potential error conditions. Unlike previous work employing similar domains [10], Rinser leverages the backward analysis to explicitly construct concise explanation traces for an error condition reported by a forwards static analysis.

As shown in Fig. 1, we have implemented Rinser as a backwards analysis tool executed after a traditional forwards analysis, such as the Infer static analyzer.

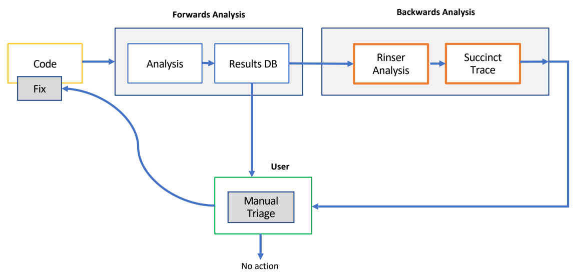

Fig. 1: Rinser *Setup.* Rinser *is an on-demand analysis given an error-condition query. To derive error queries, an initial static analysis is performed, for example, using a forwards abstract interpretation on a code base. The results consisting of potential errors along with auxiliary computed data (e.g., a call graph) is produced by the forwards analysis, which is given to* Rinser *to construct an evidence trace for each reported error. The explanation trace can be used by the developer to either produce a fix (or ignore). In the case that* Rinser *fails to produce an evidence trace, it has instead derived a sound refutation for the given error query (i.e., proven the potential error reported by the forwards analysis to be a false positive).*

When evaluating Rinser on several open source and an industrial code base, our tool is able to surface possible bugs to developers with explanation traces that on average provably remove 85% of statements compared to state-of-the-art tools in under 6 seconds.

Foremost, our tool has eased the adoption of static analyzers at Amazon and driven additional dialogue between engineers on the quality of the code. As a bonus, it has also provided a means to soundly remove many trivial false-positive warnings, thereby also reducing the subsequent manual triage effort.

Overall, our contributions are summarised below.

- Present the Rinser algorithm and tool design
- Demonstrate the utility of Rinser at an industrial scale by an experimental evaluation on 7 popular open source projects and the Amazon Prime Video application
- Report our experience using Rinser in the Amazon code review process

2 Motivation

In this section, we motivate the need for a more precise approach to static analysis evidence construction and give an overview of our proposed approach.

When a list of alarms is presented to a developer by a static analyzer, it is desirable that evidence is provided alongside the results to help the developer in their triaging effort. Commonly, evidence is provided in the form of an *explanation trace*, i.e., a sequence of execution steps that lead to the reported bug. This allows a developer to quickly decide whether to fix or dismiss the static analysis alarm. However, in the current debugging mode of industrial strength tools such as Infer, explanation traces contain excessive irrelevant execution steps. In our evaluation on a broad set of benchmarks (Section 6) we have found that on average 85% of statements in Infer traces are irrelevant i.e., do not contribute to the issue.

As we discuss in Section 7, this can be detrimental to usability, productivity and erode user trust.

```
137. char ∗∗rawHistory = malloc(sizeof(char∗) ∗ historyState → length);
...
162. if(historyList[i] → line && strlen(historyList[i] → line) > itemOffset){
163.     line = historyList[i] → line + itemOffset;
164. else {
165.     line = historyList[i] → line;
166. }
167. rawHistory[rawOffset] = line;
```

Fig. 2: Null Dereference Example in Hstr[21]

For instance, Fig. 2 is a code fragment of Hstr [21], an open source benchmark from Section 6. Infer reports that `rawHistory` may be `null` since `malloc` is used at line 137, and it is then dereferenced by calling `rawHistory[rawOffset]`. It is clear that the conditional statement (from line 162 to line 165) is not related to whether `rawHistory` is `null` or dereferenced. However, it is included in Infer trace, as demonstrated in Fig. 3.

Given we demonstrate this on a small illustrative example, it may not be obvious that irrelevant information impacts developer productivity. However consider our real world industrial scale code base, namely, the Amazon Prime Video application. Here the average explanation trace provided by Infer consists of 44.5 instructions and can be as large as 94 execution steps. When a large portion of these instructions have no relation to the reported bug the trace can be deemed as unintelligible and negatively impact on the actionability of static analysis results.

To produce more succinct explanation traces, we use a backwards analysis, which starts from the suspected bug location and attempts to construct an explanation trace that can lead to the suspected bug. We convert the bug into a *query*, i.e., a logical formula that describes the state at the location the suspected bug occurred. For our example, we convert the dereference to a logical assumption: $\texttt{emp} \wedge \texttt{rawHistory} = \texttt{null}$ before line 167, following the algorithm described in

```
src/hstr_history.c : 137 : 9 :
137. char **rawHistory = malloc(sizeof(char*) * historyState → length);
...
src/hstr_history.c : 162 : 16 :  Taking true branch
162. if(historyList[i] → line && strlen(historyList[i] → line) > itemOffset){
        ↑
src/hstr_history.c : 162 : 40 :  Taking true branch
162. if(historyList[i] → line && strlen(historyList[i] → line) > itemOffset){
                                ↑
src/hstr_history.c : 167 : 13 :
167. rawHistory[rawOffset] = line;
```

Fig. 3: Infer *Trace Segment of Null Dereference Alarm of Fig.* 2

Section 6.1, this assertion flows backwards, to the conditional statement at line 162.

Since both branches of this statement *do not modify* our initial assumption, the statement is considered *irrelevant* and not included in our explanation trace. Hence, our explanation trace only contains information related to whether the bug happens or not, filtering out all irrelevant information.

Another dimension where Rinser removes irrelevant execution steps, is in the pruning of infeasible traces. Rinser prunes explanation traces that lead to a contradictory state and thus avoids further unneeded and irrelevant explanation tarce construction. For instance, Fig. 4 shows an example taken from the benchmark OpenSSL [31] evaluated in Section 6.

```
static int test_bio_callback(void){
    bio = BIO_new(BIO_s_mem());
    if (bio == NULL)
        goto err;
    ...
    err :
        BIO_free(bio);
}
int BIO_free(BIO *a){
    if (a == NULL)                          //The case a is null is handled
        return 0;
    if(CRYPTO_DOWN_REF(&a → references, &ret, a → lock) <= 0)
    ...
}
```

Fig. 4: A Null Dereference Example in OpenSSL [31]

The debugging information shows that `bio` could be `null` and it is dereferenced by calling `bio_free`. However, the case input parameter `a` of `bio_free` is `null` is handled in `bio_free` by the conditional statement (line 10 and 11). Hence, it is impossible that `a` could be `null` and dereferenced in `bio_free`. Using our approach, an assertion $\texttt{emp} \wedge a = \texttt{null}$ is inserted at line 12 in which the alarm reports that `a` is dereferenced by executing $\texttt{a} \rightarrow \texttt{references}$. Going backward, it meets the condition $a \neq \texttt{null}$ which makes the new assertion $\texttt{emp} \wedge a = \texttt{null} \wedge a \neq \texttt{null}$. This assertion is proved to be unsatisfiable using the SMT solver Z3 [28]. In this particular case, since no explanation trace exist and the reported bug can be soundly proven as noise and dismissed.

3 Evidence Construction Algorithm

In this section, we describe our algorithm for computing concise explanation traces as evidence for static analysis alarms.

We define a set of predicates, **Pred**. Each predicate is defined by a combination of finite intuitionistic separation logic and path constraints formulae ordered by implication (cf. Section 3.1 [10]). A predicate permits us to formulate a query $\mathcal{Q} \in \textbf{Pred}$ that describes an error condition or a state that leads to an error condition.

We assume an *unstructured* control-flow graph (CFG) as a programming model to describe our algorithm. A CFG has a set of locations **Loc** and transitions **Trans**. A transitions denoted $trans := loc \rightarrow_{cmd} loc'$ where each transition has a source location $loc \in \textbf{Loc}$ and destination location $loc' \in \textbf{Loc}$ and an associated atomic command $cmd \in \textbf{Cmd}$. A location may also be be a entry or exit location. An *alarm* is a query $\mathcal{Q} \in \textbf{Pred}$ and location. We define a generic monotonic backward predicate transformer Pre such that $Pre : \textbf{Cmd} \times \textbf{Pred} \rightarrow \textbf{Pred}$. Given a command $cmd \in \textbf{Cmd}$ and initial predicate formula $\mathcal{Q} \in \textbf{Pred}$, Pre produces a new $\mathcal{Q}' \in \textbf{Pred}$, i.e., $Pre(cmd, \mathcal{Q}) = \mathcal{Q}'$. Since we assume atomic commands in a CFG, the predicate transformer is *exact*. For simplicity we don't specify all the transformers for each possible statement and leave the language unspecified. We refer the reader to [10] for standard transformer semantics for an imperative language. However, we highlight since we are operating on a CFG, structured statements (e.g., while loops) are not defined in the backward transformer.

Definition 1 (Statement Relevance). *An atomic command $cmd \in$ **Cmd** is relevant to a trace $\mathcal{T}$ iff for a query $\mathcal{Q}$ and a monotonic backward transformation $Pre(cmd, \mathcal{Q})$, $\mathcal{Q} \neq Pre(cmd, \mathcal{Q})$*

Definition 2 (Succinct Trace Explanations). *Assume CFG path $cmd_1 \ldots cmd_n$, a feasible* trace explanation *$\mathcal{T}$ is a sequence of commands that can be defined inductively from the original query $\mathcal{Q}_1$. Thus,*

$$cmd_1 \in \mathcal{T} \text{ iff } Pre(cmd_1, \mathcal{Q}_1) = \mathcal{Q}_2 \text{ s.t } \mathcal{Q}_2 \neq \bot \quad (1)$$

$$cmd_n \in \mathcal{T} \text{ iff } cmd_{n-1} \in \mathcal{T} \wedge Pre(cmd_n, \mathcal{Q}_n) = \mathcal{Q}_{n+1} \text{ s.t. } \mathcal{Q}_{n+1} \neq \quad (2)$$

In other words, Definition 2 states an backward execution of feasible explanation trace does not lead to the state $\perp$ and its intermediate states cannot equal $\perp$.

Moreover, we say if a trace explanation only contains statements that are relevant *(Definition 1), i.e., each state in the trace execution changes, then the trace is a* succinct explanation trace. *The set of succinct explanation traces is called a succinct explanation trace set, or just trace set, defined by $\overline{\mathcal{T}}$.*

3.1 Algorithm Description

3.1.1 Algorithm MkSuccinctTraces: We describe our approach via the Algorithm `MkSuccinctTraces`. To ease the presentation we make some simplifying assumptions to aid presentation. We assume only one entry location and exit location per function, and one caller pre function and do not go to the level of detail of describing work list fixpoint algorithms to deal with loops. We implement standard algorithms for fixpoint computations. Moreover, we omit several optimizations which we describe at separately.

The input to Algorithm **MkSuccinctTraces** is a specific alarm, i.e, a query, location pair, a CFG of a given function $\mathcal{P}$ obtained from a call graph `call_graph`. `MkSuccinctTraces` returns a set of succinct traces. We note in our implementation we can vary between finding all traces and finding a single trace. The the latter case the essence of the approach does not change.

The inputs apart from our trace come from a prior forward static analysis (see Fig. 1). Initially our trace set $\overline{\mathcal{T}}$ is an empty set. We process the initial function via a call to the `ProcFunc` algorithm which given a predate, location pair, a procedure with a CFG, a call graph and a trace set, augments the trace set and returns the state predicate at its entry location.

If the processed function $\mathcal{P}$ has any caller function $\mathcal{P}'$, the caller function is processed with the exit location (call to `GetExitLoc`) and state produced by $\mathcal{P}$ as input. The predicate state $\mathcal{Q}$ and trace set is updated.

When we reach the entry function and no more functions need to be processed we terminate and return our trace set.

3.1.2 Algorithm ProcFunc: The Algorithm `ProcFunc` traverses all paths starting from the initial query location to the entry location of a function. This is done by calling for each command `ProcCmd`. Since we operate on a CFG, we may encounter loops in the CFG. For readability, we omit details of our worklist algorithms and the fixpoint computation, however we specify `widen` as a widening operator to highlight the fact that when a loop is encountered a widening operator is used to accelerate termination with the cost of over-approximating the state and hence the trace set, i.e., as a consequence some traces in our trace set may be false explanations.

Moreover, if a path leads to an infeasible state, i.e., $\mathcal{Q}_1 = \bot$, we ensure its trace is not included in our trace set. Otherwise the trace (or trace set if there is a call command) is added to the set of traces and the states are merged via the disjunctive operator. Any equivalent states will result in the predicate being simplified using standard laws of logic.

3.1.3 Algorithm ProcCmd:

The `ProcCmd` algorithm distinguishes between two cases:

(1) The case that a function is called, in which case we check if the function is relevant to the query. If so, we call the `ProcFunc` procedure. To reduce the computation effort, we only go into checking a called function $\mathcal{F}$ when its parameters $\vec{n}$ or the variable its returned value is assigned to, e.g. `y`, relate to the current query. We check that in the procedure $\texttt{Related}(\mathcal{Q}, y, \vec{n})$. This procedure returns `true` when `y` or one of the parameters $\vec{n}$ appears in the heap part of the query $\mathcal{Q}$. Intuitively, this means the function $\mathcal{F}$ may alter the query $\mathcal{Q}$. Note because of this case, a command may return a trace set. Because of this, the input trace set and output trace set of the called procedure need to me merged where the `merge` operator appends all traces in one trace set to all traces in the other trace set. For example if trace sets $\overline{\mathcal{T}}_1$ = { [1, 2, 3], [4, 5, 6] } and $\overline{\mathcal{T}}_2$ = { [5, 7], [9, 11], [12] } then $\texttt{merge}(\overline{\mathcal{T}}_1, \overline{\mathcal{T}}_2)$ = { [1, 2, 3, 5, 7], [1, 2, 3, 9, 11], [1, 2, 3, 12] [4, 5, 6, 5, 7], [4, 5, 6, 9, 11], [4, 5, 6, 12] }.

(2) The default case handles an atomic command. To update the value of a query $\mathcal{Q}$, we process statements in a backward manner, using `Pre` procedure. For instance, in the assignment $\mathcal{Q}' = \texttt{Pre}(\mathcal{Q}, \texttt{cmd})$ $\mathcal{Q}'$ is the query before executing the statement `cmd` and $\mathcal{Q}$ is the query after the execution. Hence, the tripe $\{\mathcal{Q}'\}$ `cmd` $\{\mathcal{Q}\}$ is like a Hoare triple in program verification but since `Pre` is defined on atomic commands i.e., no loops etc. the triple is *exact*. In this case, a trace set with a single trace is returned.

3.2 Key Properties

Provable Relevance Checking: In all cases, Definition 1 is enforced by only adding commands to a trace that change the predicate formula. To add an execution point into our trace $\mathcal{T}$, we compare values of the old query $\mathcal{Q}$ and the new one $\mathcal{Q}'$. If they are different, we add the execution to the current trace $\mathcal{T}$ to make a new trace $\mathcal{T}'$. Moreover, if we prove a trace produces an infeasible state i.e., equal to formula $\bot$, it can be soundly pruned. This is done, by the use of an SMT solver (e.g., Z3 [28]). If all traces lead to a $\bot$ state we say the query is *refuted*.

Over-Approximation: If a command `cmd` is in a loop we perform extrapolate the inductive invariant by weakening the formula to insure we over-approximate and ensure termination. Weakening a *Pre* means that it discharges an assumption and conceptually assumes that the loop can satisfy that condition. If it doesn't, then it's a false explanation, but we only get sound refutations. A simple instance of this strategy is to removing the path conditions (setting them to true). However, other mechanisms can be implemented [14]. Note: the need for widening results in over-approximation thus in the presence of loops our trace may indeed be spurious. We point the reader to Section 7 for a discussion in the consequences of this decision.

Trace Sets as Succinct CFG: We have described evidence as a set of partial succinct traces to the potential error location (with the initial error-condition query). They are succinct in that they leave out or slice out the commands that do not change the error-state query. They are partial in that the head of the trace is a residual error-state query at the head location that if unsatisfiable guarantees that the initial error-condition query at the potential error location is unsatisfiable. Alternatively, we can see this set of partial succinct traces as a sub-graph of the control-flow graph that slices out the irrelevant commands with error-state queries at a (backward) frontier from the potential error location.

3.3 Optimisations

We describe several optimizations that can improve the performance of the algorithm but are not detailed in Fig 5.

Forward memory information: From the forward analysis we can use over-approximating points to information or function summaries in separation logic to quickly determine if processing a function will lead to all infeasible traces. In [10] this points to information is used to speed up refutations, and avoid path explosions. In the case of analyzers such as Infer we can use the function summaries provided.

Stop at true states: During trace exploration if a state $\mathcal{Q} = \top$ or `true`, due to the monotonic properties of Pre, we can correctly assume that all proceeding commands will result in a true state. Thus we can stop the trace construction at that point.

Intersection with forward trace: Given an over approximating forward trace is computed the intersection of our over approximating backward trace can improve precision and lead to less false positives. The resulting trace set will in this case be a subset of both the forward and backward trace sets.

4 Integration into the Infer Static Analyzer

4.1 Implementation Details

We implement Rinser as a backward analysis *checker* on top of the Infer **explore** mode[4] in 2.5K lines of OCaml code. Rinser uses the explore mode to

[4] https://fbinfer.com/static/man/infer-explore.1.html

MkSuccinctTraces(alarm = ($\mathcal{Q}$, loc), ($\mathcal{P}$, `cfg`), call_graph)

Input: An alarm: query $\mathcal{Q}$, location `loc` pair in the CFG `cfg`, of a current function $\mathcal{P}$, in the call graph `call_graph`
Output: Return a set of explanation traces $\overline{\mathcal{T}}$.

```
1: $\overline{\mathcal{T}}$, $\mathcal{Q}$:= ProcFunc(alarm, ($\mathcal{P}$, cfg), call_graph, $\emptyset$)
2: while $\mathcal{P}$ has caller $\mathcal{P}'$ with cfg$'$ do
3:     $\overline{\mathcal{T}'}$, $\mathcal{Q}'$:= ProcFunc(($\mathcal{Q}$, GetExitLoc($\mathcal{P}'$)), ($\mathcal{P}'$, cfg$'$), call_graph, $\overline{\mathcal{T}}$)
4:     $\mathcal{Q}$:= $\mathcal{Q}'$; $\overline{\mathcal{T}}$:= $\overline{\mathcal{T}'}$; $\mathcal{P}$:= $\mathcal{P}'$
5: return $\overline{\mathcal{T}}$
```

ProcFunc(($\mathcal{Q}$, loc), ($\mathcal{P}$, `cfg`, `call_graph` $\mathcal{T}$))

Input: predicate $\mathcal{Q}$, location `loc` pair in the CFG `cfg`, of a current function $\mathcal{P}$, in the call graph `call_graph`,
Output: Return a set of explanation traces $\overline{\mathcal{T}'}$and predicate $\mathcal{Q}'$.

```
1: $\overline{\mathcal{T}'}$:= $\emptyset$
2: for ($loc'$ | $loc \rightarrow_{cmd} loc' \in$ cfg) do
3:     $\overline{\mathcal{T}}_1$, $\mathcal{Q}_1$ := ProcCmd( $loc \rightarrow_{cmd} loc'$, ($\mathcal{P}$, cfg), call_graph, $\overline{\mathcal{T}}$)
4:     if InLoop(loc$'$) then
5:         $\mathcal{Q}_1$ := Widen($\mathcal{Q}_1$, cmd)
6:     if $\mathcal{Q}_1 = \perp$ then
7:         $\overline{\mathcal{T}}_2 = \emptyset$; $\mathcal{Q}_2 = \mathcal{Q}_1$
8:     else
9:         $\overline{\mathcal{T}}$2, $\mathcal{Q}$2 := ProcFunc(($\mathcal{Q}_1$, l'), ($\mathcal{P}$, cfg), call_graph, $\overline{\mathcal{T}}_1$)
10:    $\overline{\mathcal{T}'}$:= $\overline{\mathcal{T}'}\cup \overline{\mathcal{T}}_2$; $\mathcal{Q}'$:= $\mathcal{Q}'\vee$ $\mathcal{Q}_2$
11: return $\overline{\mathcal{T}}$, $\mathcal{Q}'$
```

ProcCmd($\mathcal{Q}$, $loc \rightarrow_{cmd} loc'$, ($\mathcal{P}$, `cfg`), call_graph, $\mathcal{T}$)

Input: predicate $\mathcal{Q}$, location `loc` pair in the CFG `cfg`, of a current function $\mathcal{P}$, in the call graph `call_graph`, a trace set $\overline{\mathcal{T}}$of accumulated traces
Output: Return a set of explanation traces $\overline{\mathcal{T}'}$and predicate $\mathcal{Q}'$.

```
1: switch cmd do
2:     case y = $\mathcal{F}(\vec{n})$:                         // Call function F with parameter $\vec{n}$
3:         if Related($\mathcal{Q}$, y, $\vec{n}$) then
4:             loc = GetExitLoc($\mathcal{F}$)
5:             cfg$'$ = GetCfg($\mathcal{F}$)
6:             ($\mathcal{Q}'$, $\overline{\mathcal{T}}_1$) = ProcFunc(($\mathcal{Q}$, loc) $\mathcal{F}$, cfg$'$, call_graph, {cmd})
7:             $\overline{\mathcal{T}'}$:= merge($\overline{\mathcal{T}}$, $\overline{\mathcal{T}}_1$)
8:         else ($\mathcal{Q}'$, $\overline{\mathcal{T}'}$) = ($\mathcal{Q}$, $\overline{\mathcal{T}}$)
9:     case Default:
10:        $\mathcal{Q}'$:= Pre(cmd, $\mathcal{Q}$)
11:        if $\mathcal{Q}' \neq \mathcal{Q}$ then $\overline{\mathcal{T}} \neq \emptyset$ ? $\overline{\mathcal{T}'}$:= { cmd::t : t $\in \overline{\mathcal{T}}$} : $\overline{\mathcal{T}'}$:= { cmd }
12:        else $\mathcal{T}'$:= $\mathcal{T}$
13: return $\mathcal{Q}'$, $\overline{\mathcal{T}'}$
```

Fig. 5: Trace Explanation Construction

access to information computed during the forwards analysis. Rinser operates on the Smallfoot Intermediate Language (SIL) [7] which is one of the main intermediate representations inside Infer. The current implementation uses a syntactic call graph provided by Infer, however an external call-graph based on a points-to analysis can be used in principle to provide increased precision. Due to no explicit points-to analysis being available in Infer function summaries and the existing Infer trace can be used to implement the optimizations outlined in Section 6.1. In the implementation we may each SIL instruction to the C++ code location, so the trace contains pointers to the original code. We leverage much of the Infer checker infrastructure to construct our abstract domain and perform our backwards analysis including fixpoint computations.

4.2 Usage

When integrated into the code review process, Rinser can be run in batch right after the static analysis run finishes or on-demand, when selected by the user. Typically, a piece of code is flagged by a code review bot that executes Infer under the hood. The bot allows the user to obtain additional information by way of a witness trace. If Rinser has not been run before in batch mode, the code review system will run Rinser using the command: `Infer explore --find-witness --select ID` where ID is the alarm ID, otherwise it will obtain cached trace information. The current service level agreement (SLA) states that the tool has 30 seconds to obtain a trace. If no trace can be produced (but some may exist), e.g., due to a timeout, the pre-computed Infer trace is used. Rinser tarces are shown using the command: `Infer explore --show-witness`. We currently use the Infer call-graph which is syntactic as well as the existing trace to optimize the algorithm.

5 Related Work

Static Analyzer Explanations: Static analysis has been described in the industrial code review process in [2, 13, 35, 18]. In this paper, we focus on one element of integrating static analysis into the code review process, namely, providing users with alarm explanations. Trace explanations or witness computations are described for industrial static analysers such as Astrée in [33] and Soufflé [41, 38]. The explanation mechnism also used a backward goal orientated analysis, however their domain is a abstract (approximate) trace domain. The work in Soufflé focuses on Datalog-based static analyzers and users a proof annotations computed in the forward (bottom-up) evaluation to provide minimal proof trees in a top down (backward) evaluation. Brauer and Simon propose [11] a technique which performs a backwards analysis using abduction of propositional Boolean logic to generate legitimate traces. In [8], a forward fixpoint algorithm is proposed along with a general theory showing how invariants, issued as post-fixpoints of abstract interpretations, can be compressed to provide witnesses of particular program properties. In [12], concise traces are learnt from model checking counter-examples and Beer et al. [6] uses the notion of causality to explain counter-examples. Lahiri et al. [25] present a technique in which root causes of verification failures are found using MaxSat. In [19] a method for providing better explanations to typing errors is described. **Program Slicing:** Our technique has some similarities to slicing in that we search for feasible program paths from an error location when we build our trace containing relevant commands. Compared to traditional program slicing [40], our technique is *semantic* and is closer to semantic slicing techniques e.g., [23, 9]. Our techniques has seemingly orthogonal aims to slicing: we are primarily concerned with building a succinct explanation trace, pruning infeasible paths is a bi-product of our approach. On the other hand, our trace set computation can be seen as a semantically *sliced* CFG that can be used to highlight relevant code or generate traces as explanations as realized in Rinser. **Backward Analysis:** Our approach is inspired by the Thresher algorithm [10]. We use a backwards exploration that is over-approximating and refutably sound. Unlike Thresher, however, Rinser explicitly builds a trace witness and adds another dimension of pruning: it uses semantic information obtained from the backward analysis to build traces that only contains relevant commands. At a more practical level, Rinser targets C/C++ code and is implemented in the Infer static analysis framework. Moreover, Our approach shares similar theoretical similarities with other backward static analyses [30, 3, 1, 15].

6 Evaluation

In this section, we evaluate Rinser on popular open source libraries and an industrially sourced code base from Amazon Prime Video. These code bases are written in C/C++. The outcome of our evaluation is to validate the following claims.

Claim-I: Reduction in Irrelevant Commands in Trace. Compared to Infer, we are able to produce more succinct explanation traces by removing irrelevant trace information.

Claim-II: Succinct Explanation Construction within SLA. We claim that for a very short SLA of 30 seconds, we are able to produce a significant reduction in explanation trace sizes.

Experimental Setup. Our experiments were conducted on an Intel® Core™ i7-6700 (3.4GHz) CPU, 8GB RAM, running Ubuntu 16.04 LTS. We first use Infer to analyze the code bases and get a list of alarms. Our experiments focus on null dereferences but in principle we can support other memory related properties.

6.1 Open Source Code Bases

We run Rinser on a number of popular open source code bases in Fig. 6 and present results in Fig. 7. We report the size of each code base in the **KLOC**

Benchmark	KLOC
Hstr [21]	19
Htop [22]	20
Craft [16]	55
Tmux [39]	65
Hashcat [20]	141
OpenSSL [31]	300
NeoMutt [29]	484
Prime Video	283

Fig. 6: Benchmark Characteristics

Benchmark	Tot	#Pru	#Tr	#TO	IR(%)	Len(avg)	ILen(avg)	Ratio	Exec(s)
Hstr [21]	27	2	25	0	56	3.44	7.8	2.3	0.33
Htop [22]	44	0	44	0	78	2.6	12.2	4.62	0.34
Craft [16]	53	2	51	0	79	4.8	23.2	4.9	2.18
Tmux [39]	34	6	28	0	83	2.9	16.7	5.8	17.92
Hashcat [20]	50	3	47	0	81	4.3	28	5.3	1.94
OpenSSL [31]	91	30	57	4	77	5.1	21.8	4.3	3.22
NeoMutt [29]	83	7	74	2	74	5.9	22.6	3.8	2.2
Prime Video	93	30	61	2	88	5.5	44.5	8.2	9.6

Fig. 7: Results of Running RINSER *on C/C++ Benchmarks*

column. The column **Tot** is the number of potential alarms produced by INFER. **#Pru** shows the number of potential alarms that our tool could prune. **#Tr** denotes the number of evidence traces produced. **#TO** expresses the number of times our tool was not able to finish within the 30 second SLA. **IR** shows the percentage of irrelevant statements in the original INFER trace, **Len** and **ILen** are the number of commands on average in traces of our tool and INFER, respectively. Consequently, **Ratio** shows the length reduction ratio. **Exec(s)** computes the average execution time of pruning and/or computing traces when it did not timeout.

Our results demonstrate the utility of our approach on 7 open source benchmarks. ***Claim-I*** is supported by the fact that we were able to prove 85% of commands on average from the INFER trace were irrelevant, this resulted in their removal and our traces being on average 5.1x shorter, while performing full refutations on 13% of issues. This is because our evidence construction algorithm in Section only add a command into an evidence trace when it is verified to be relevant with the current queries. Besides, there are only 6 issues timed out in these open source benchmarks since our algorithm `ProcCmd` in Section only processes related called functions. Hence, the running time of most issues is well bellow the SLA time.

Claim-II is supported by the fact that we were able to produce explanation traces for 85% of potential alarms produced by INFER. Hence, for 98% of bugs we were able to either refute or produce improved explanations in less than 6 seconds on average, well below the SLA of 30 seconds. The benchmarks show a trend towards more timeouts with larger code sizes. At the same time, opportunities for refutations appear to increase.

6.2 Amazon Prime Video

We evaluate our tool on a Prime Video application implemented in C/C++. We cannot provide a detailed design of the service, but we highlight the key features. The service runs as part of the Prime Video application on the actual devices, e.g., televisions, Roku devices, Amazon Fire TV devices, etc. and consists of 283 KLOC.

We report the evaluation results of the Amazon Prime Video code base in the last row in Fig. 7. ***Claim-I*** is supported by the fact that the explanation traces computed were on average 8.2x shorter than Infer's. 32% of the time no traces were produced due to a refutation, and 66% of the time a trace witness was found. In the rare event that there is a time out (2% of the time) we fall back on Infer's traces. ***Claim-II*** is supported by the fact that when we do not time out, a refutation or explanation is computed in less than 10 seconds on average, again, well below the SLA of 30 seconds.

7 Experiences and Lessons Learnt

Rinser is used at Amazon to help clarify static analysis results to developers. We were initially motivated by low action rates on static analysis results. As similarly concluded in several studies, e.g., [36, 24, 4, 34], we found the main reason for developer disengagement with static analyzers was poor *result understandability*, i.e., a lack of understanding of analyzer output, followed by a tedious debugging effort and a subsequent loss of trust in the tools. Moreover, our experience working with Amazon development teams to integrating static analysis in their code review process has led to some surprising observations which have further motivated the design of our tool.

Observation-I: Concise Evidence is At Least as Important as Precision. If the developer cannot quickly understand the reason for an alarm, then almost always they take no further action. That is, determining whether the alarm actually corresponds to a bug or not is postponed indefinitely—in effect dismissing the alarm as "false" regardless of whether or not there is a true bug lurking behind the alarm. Thus, even high precision analysis results may be deemed ineffective, if the root cause of the alarm is obscure and not quickly resolved by the developer.

Observation-II: Not All Code is Equal. During on-boarding meetings with developers, we learnt that the time developers were willing to invest in alarms depends on the code location. Developers ignore alarms in some files (e.g., test code, libraries) and prioritize investigations in others. This need to prioritize precious triage time motivates Rinser's on-demand-first design.

Observation-III: Not All Results are Equal. Minimizing false positives is indeed important—"obvious false alarms" undoubtedly negatively impact user trust. But at an industrial scale, it is not the only metric of importance for an analyzer. For example, industrial-scale analyzers need to meet service-level agreements (SLAs) for developer wait time [26] to mitigate the pitfalls of context switching [27]. We have found some false positives are "forgivable" and in fact have some utility—provided that they can be efficiently understood. We found that concise evidence for non-trivial alarms (that perhaps turn out not to be actual bugs) facilitated discussion among developers and has led to code improvements that, for example, avoid potential future bugs. Similar observations are reported in [17] and can be compared to scenarios in the traditional manual code review process where no concrete bug is found but a code improvement is desirable nonetheless. In essence, if a false positive is hard to refute algorithmically, it may also be hard for a developer to manually understand the code. On the other hand, true positives may have no action associated with them as the application context (e.g., how it is used, other underlying assumptions about external libraries) are not explicit in code and thus cannot be incorporated into the analysis. While such cases are technically true positives, they are treated as false positives and not auctioned upon in reality. Thus we recommend the metric of fix rate as an indicator of the effectiveness of a tool. Rinser thus not only attempts to improve triage productivity but also improved fix rates.

8 Future Work

User Counter Arguments: Rinser is the first increment in our overall goal of narrowing the gap between developers and static analyzers. In our current implementation, once a trace is received, a user can ultimately reject or accept the trace. In future iterations of our tool we want to enable an interactive REPL like environment where a developer could provide Rinser with a counter argument (e.g., where the tool has missed some context, or over approximated due to widening). Rinser can then rewind its abstract execution, incorporate the counter argument and produce a new witness.

Repair Suggestions as Explanations: A trace explanation is a set of commands that evaluate to true and thus explain why the analyzer believes the error to be real. Along side the trace, potential mutations in the code can be computed which falsify the trace explanations and thus could provide additional utility to the user by providing a *suggested* fix that soundly refutes an alarm.

Incorporating Other Abstract Domains: The current Rinser implementation only supports null dereference queries. We plan to extend Rinser to support other memory related queries such as memory leaks. Moreover, in future work we plan on extending our approach to other abstract domains such as numerical domains.

9 Conclusion

We have presented Rinser, a tool developed for computing explanations for static analysis alarms. Rinser performs a backwards analysis to build concise witness traces that have been pivotal for user acceptance of static analyzers at Amazon. We have demonstrated the utility of Rinser on both popular open source code bases and an industrially sourced Amazon code base, namely, the Amazon Prime Video application. We show that our tool is able to meet the industrial SLA of 30 seconds 98% of the time and to remove irrelevant statements in traces thus reducing the size of the trace by 6.3x on open source benchmarks and 10x on our industrial use case.

References

[1] Aws Albarghouthi, Josh Berdine, Byron Cook, and Zachary Kincaid. “Spatial Interpolants”. In: *European Symposium on Programming (ESOP).* 2015.

[2] Vipin Balachandran. “Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation”. In: *International Conference on Software Engineering (ICSE).* 2013, pp. 931–940.

[3] Thomas Ball, Orna Kupferman, and Greta Yorsh. “Abstraction for Falsification”. In: *International Conference on Computer Aided Verification (CAV).* 2005.

[4] Sebastian Baltes, Oliver Moseler, Fabian Beck, and Stephan Diehl. “Navigate, Understand, Communicate: How Developers Locate Performance Bugs”. In: *ESEM.* 2015.

[5] Titus Barik. “How should static analysis tools explain anomalies to developers?” In: *FSE.* 2016.

[6] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. “Explaining counterexamples using causality”. In: *Formal Methods in System Design* (2012).

[7] Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. “Smallfoot: Modular Automatic Assertion Checking with Separation Logic”. In: *FMCO.* 2005.

[8] Frédéric Besson, Thomas P. Jensen, and Tiphaine Turpin. “Small Witnesses for Abstract Interpretation-Based Proofs”. In: *European Symposium on Programming (ESOP).* 2007.

[9] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. “Path Invariants”. In: *SIGPLAN Not.* 42.6 (June 2007).

[10] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. “Thresher: Precise Refutations for Heap Reachability”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 2013, pp. 275–286.

[11] Jörg Brauer and Axel Simon. "Inferring Definite Counterexamples through Under-Approximation". In: *NASA International Symposium on Formal Methods (NFM)*. 2012.

[12] Martin Chapman, Hana Chockler, Pascal Kesseli, Daniel Kroening, Ofer Strichman, and Michael Tautschnig. "Learning the Language of Error". In: *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. 2015, pp. 114–130.

[13] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. "Continuous Formal Verification of Amazon s2n". In: *International Conference on Computer Aided Verification (CAV)*. 2018.

[14] A. Cortesi. "Widening Operators for Abstract Interpretation". In: *IEEE International Conference on Software Engineering and Formal Methods*. 2008, pp. 31–40.

[15] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. "Precondition Inference from Intermittent Assertions and Application to Contracts on Collections". In: *Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2011, pp. 150–168.

[16] *Craft Source Code*. 2020 (accessed Jan 21, 2020). URL: `https://github.com/fogleman/Craft`.

[17] Carlo Dimastrogiovanni and Nuno Laranjeiro. "Towards Understanding the Value of False Positives in Static Code Analysis". In: *2016 Seventh Latin-American Symposium on Dependable Computing, LADC 2016, Cali, Colombia, October 19-21, 2016*. 2016, pp. 119–122.

[18] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. "Scaling Static Analyses at Facebook". In: *Communications of the ACM* 62.8 (2019), pp. 62–70.

[19] Nabil El Boustani and Jurriaan Hage. "Corrective Hints for Type Incorrect Generic Java Programs". In: *PEPM*. 2010.

[20] *Hashcat Source Code*. 2020 (accessed Jan 21, 2020). URL: `https://github.com/hashcat/hashcat`.

[21] *Hstr Source Code*. 2020 (accessed Jan 21, 2020). URL: `https://github.com/dvorka/hstr`.

[22] *Htop Source Code*. 2020 (accessed Jan 21, 2020). URL: `https://github.com/hishamhm/htop`.

[23] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. "Path-Sensitive Backward Slicing". In: *SAS*. 2012, pp. 231–247.

[24] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In: *International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681.

[25] Shuvendu K. Lahiri, Rohit Sinha, and Chris Hawblitzel. "Automatic Root-causing for Program Equivalence Failures in Binaries". In: *International Conference on Computer Aided Verification (CAV)*. 2015, pp. 362–379.

[26] Thomas D. LaToza, Gina Venolia, and Robert DeLine. "Maintaining mental models: a study of developer work habits". In: *International Conference on Software Engineering (ICSE)*. 2006, pp. 492–501.

[27] Lucas Layman, Laurie Williams, and Robert St. Amant. "Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools". In: *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, September 20-21, 2007, Madrid, Spain*. 2007, pp. 176–185.

[28] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. 2008.

[29] *NeoMutt Source Code*. 2020 (accessed Jan 21, 2020). URL: https://github.com/neomutt/neomutt.

[30] Peter W. O'Hearn. "Incorrectness Logic". In: *PACMPL* 4.POPL (2020).

[31] *OpenSSL Source Code*. 2020 (accessed Jan 21, 2020). URL: https://github.com/openssl/openssl.

[32] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *LICS*. USA, 2002.

[33] Xavier Rival. "Understanding the Origin of Alarms in Astrée". In: *International Static Analysis Symposium (SAS)*. 2005.

[34] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "Lessons from Building Static Analysis Tools at Google". In: *Communications of the ACM* 61.4 (2018), pp. 58–66.

[35] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. "Modern Code Review: A Case Study at Google". In: *ICSE - SEIP*. 2018.

[36] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. "Tricorder: Building a Program Analysis Ecosystem". In: *International Conference on Software Engineering (ICSE)*. 2015, pp. 598–608.

[37] Saurabh Sarkar and Chris Parnin. "Characterizing and Predicting Mental Fatigue during Programming Tasks". In: *2nd IEEE/ACM International Workshop on Emotion Awareness in Software Engineering, SEmotion@ICSE 2017, Buenos Aires, Argentina, May 21, 2017*. 2017, pp. 32–37.

[38] Jixiang Shen, Xi Wu, Neville Grech, Bernhard Scholz, and Yannis Smaragdakis. "Explaining Bug Provenance with Trace Witnesses". In: *SOAP*. 2020.

[39] *TMux Source Code*. 2020 (accessed Jan 21, 2020). URL: https://github.com/tmux/tmux.

[40] Mark Weiser. "Program Slicing". In: *ICSE*. ICSE '81. 1981.

[41] David Zhao, Pavle Subotic, and Bernhard Scholz. "Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy". In: *ACM Trans. Program. Lang. Syst.* (2020).