# Casper: An Efficient Approach to Call Trace Collection

Rongxin Wu[†] Xiao Xiao[†] Shing-Chi Cheung [†] Hongyu Zhang[§] Charles Zhang[†]
[†]The Hong Kong University of Science and Technology [§]Microsoft Research
[†]{wurongxin, richardxx, scc, charlesz}@cse.ust.hk [§]honzhang@microsoft.com

## Abstract

Call traces, i.e., sequences of function calls and returns, are fundamental to a wide range of program analyses such as bug reproduction, fault diagnosis, performance analysis, and many others. The conventional approach to collect call traces that instruments each function call and return site incurs large space and time overhead. Our approach aims at reducing the recording overheads by instrumenting only a small amount of call sites while keeping the capability of recovering the full trace. We propose a call trace model and a logged call trace model based on an LL(1) grammar, which enables us to define the criteria of a feasible solution to call trace collection. Based on the two models, we prove that to collect call traces with minimal instrumentation is an NP-hard problem. We then propose an efficient approach to obtaining a suboptimal solution. We implemented our approach as a tool Casper and evaluated it using the DaCapo benchmark suite. The experiment results show that our approach causes significantly lower runtime (and space) overhead than two state-of-the-arts approaches.

***Categories and Subject Descriptors*** F.3.2 [*Semantics of Programming Languages*]: Program Analysis; D.2.5 [*Testing and Debugging*]: Tracing

***General Terms*** Theory, Algorithms, Performance

***Keywords*** Call Trace, Instrumentation, Overhead

## 1. Introduction

A call trace [19], recording a sequence of function calls and returns, captures the calling behaviour of a program including the temporal execution order of function calls and their calling contexts. Call traces are widely used, for instance, in bug reproduction [21], fault diagnosis [22, 36, 45], performance analysis [17, 31, 44], program comprehension [15, 37, 38], anomaly detection [9, 18, 26, 39], and many other analyses.

The current practice of call trace collection by instrumenting *each* call and return site usually incurs large runtime and space overhead. This overhead is further exacerbated by the prevalent use of "small" functions (or methods) so as to follow the good object-oriented programming style [7, 14]. For example, we observed 213.9% runtime overhead for the Java programs in the Dacapo benchmark suite (See Section 6). At the same time, the volume of such traces logged by real production systems can grow by 50 gigabytes (around 120-200 million lines) each hour [31]. Obviously, greatly improving the time and space efficiency of call trace collection can tremendously benefit the related program analysis techniques in practice.

Existing program tracing and profiling techniques can be classified into two categories. The first focuses on collecting the intraprocedural control flow path with minimal overhead, by either selectively instrumenting a small subset of program points [4] or applying certain encoding scheme for acyclic paths free of any call sites [5, 25]. Since call traces are inherently interprocedural, one needs to extend these methods by logging the function ID before each call site [25] to connect the intraprocedural pieces, essentially just as costly as the conventional approach. The second category [7, 8, 33, 41] mainly addresses online program analysis and, thus, focuses on encoding the latest calling contexts and maintaining a summary of historic calling contexts for profiling purpose (e.g., calling context tree). Call trace collection is more general than profiling as it often needs to faithfully encode the full history of calling contexts.

Our key insight for efficiently and compactly collecting call trace is that the execution of certain call sites can be implied by the execution of some others. Therefore, the basic idea of our technique is to collect a partial call trace at runtime by selectively instrumenting a small subset of the call and return sites. This partial trace is then used *offline* to infer and to recover the full call trace. The key to such lossless recovery is a criteria that guides the selection of the instrumentation sites. This criteria is derived from our trace models based on an LL(1) grammar and a theoretical framework for finding the optimal instrumentation. We show that minimizing the instrumentation sites to satisfy the LL(1) grammar is NP-hard. We propose a $H_n$-approximation (harmonic function) [42] instrumentation approach that works very well in practice. The framework is also general enough to handle callbacks, virtual calls, JVM implicit calls, and exceptions. This generality allows us to efficiently collect call traces for modern software that heavily uses dynamic features.

In summary, the main contributions of this paper are:

- An LL(1) grammar based framework to theoretically study the inference based call trace collection problem. The theory framework defines the criteria of deducible partial call trace and opens a door to further study of inference based algorithms.
- The NP-hardness proof of minimizing the instrumentation sites and an $H_n$-approximation approach for call trace collection.
- Design and implement a tool Casper that can collect complete call trace even in presence of callbacks, virtual calls, JVM implicit calls, and exceptions.



*POPL'16*, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837619

- A thorough experimental study of Casper with the DaCapo benchmark suite. Our experiments show that Casper only incurs 68.0% runtime overhead on average, which is 31.8% of the time of instrumenting all the call sites to collect full call trace. Meanwhile, the size of the partial call trace is 36.1% of the size of the full call trace.

The remainder of this paper is organized as follows. We first briefly introduce call traces and define the terms used by the discussion in Section 2. In Section 3, we propose a theoretical framework to study the call trace inference problem. Section 4 describes our suboptimal instrumentation algorithm and Section 5 discusses the implementation details including how to handle dynamic features of Java. We discuss the experiment results in Section 6. Finally, Section 7 surveys related work and Section 8 concludes this paper.

## 2. Background

Call traces [19], composed of sequences of function calls and returns, are essential for many program analysis techniques. The temporal order of calls captured in call traces are essential for anomaly detection [9, 18, 26, 39], performance analysis [17, 31, 44] and fault diagnosis [22, 36, 45]. It also helps program comprehension and testing such as mining program specifications from call traces [15, 37, 38]. Besides, call traces subsume the calling context information [40]. Therefore, all the applications of calling context [2, 7, 8, 33, 34, 48] naturally can work with call traces.

Similar to the *tracing problem* [4], the *call trace collection problem* is concerned with recording the sufficient information about a program's execution in order to reproduce a full call trace. Call traces are conventionally collected via program instrumentation to record the execution of every call and return site, which is very costly in both time and space. An alternative collection approach is to use control flow profiling techniques [4, 5, 24, 25] to the call site control flow graph (*CSCFG*), a reduced version of control flow graph (*CFG*) that only preserves the control flow dependency for call sites and return sites. Formally, *CSCFG* is defined as follows.

**Definition 1. Call Site Control Flow Graph (*CSCFG*)**: A *CSCFG* is a directed graph defined by a 4-tuple $<P, E, H, T>$. $P$ is a set of nodes representing the call and the return sites. To ease the presentation, they are all referred to as call sites unless specially distinguished. $E$ is a set of directed edges, where an edge $<n, m>$ represents a path on the control flow graph from $n$ to $m$ without going through another call site. $H$ represents the set of entry call sites. A call site is an entry call site if there is a path from the program start point to that call site without passing other call sites. Correspondingly, $T$ represents the set of exit call sites, which reach the program exit point without passing other call sites.

**Definition 2. Predecessor Call Site** and **Successor Call Site**: Given an edge from $n$ to $m$ in a *CSCFG*, $n$ denotes a predecessor of $m$ and $m$ a successor of $n$.

Figure 1 exemplifies the *CSCFG* construction where $S_0, \cdots, S_3$ represent non-call statements and $c_1$ and $c_2$ represent call sites. The *CSCFG* contains an edge $<c_1, c_2>$since a path goes from $c_1$ to $c_2$ without going through other call sites. Moreover, $c_1$ is the entry call site and *return* is the exit call site of this *CSCFG*. $c_1$ is a predecessor of $c_2$ and $c_2$ is a successor of $c_1$.

To profile interprocedurally, Melski et al. [30] conceptually proposed an approach to encoding the interprocedural acyclic path, which extends the Ball & Larus encoding algorithm [5]. The approach labels the edges in an interprocedural super graph by functions in order to capture the calling context of procedures during execution. However, as Larus [25] pointed out, there are two major limitations that may make it impractical: (1) the large number of potential paths may make path ID unbounded; (2) difficulties in deriving static call graphs that can precisely model reflective calls. Larus [25] proposed a technique to profile the interprocedural control flow path. It encodes the acyclic intraprocedural control flow path that do not include any call site, and logs the corresponding function ID before each call site so that interprocedural path can be recorded. Therefore, adapting the technique [25] to collect call traces is the same as instrumenting at each call site and return site. Larus [24] proposed to instrument at the targets of each conditional branch and regenerate a full trace from the log data. We adapt this technique [24] by instrumenting the call sites that are the targets of the branch nodes for call trace collection. However, our experiments show that using this technique still incurs a significant performance overhead. And this motivates us to study how to solve the call trace collection problem with minimal instrumentation.

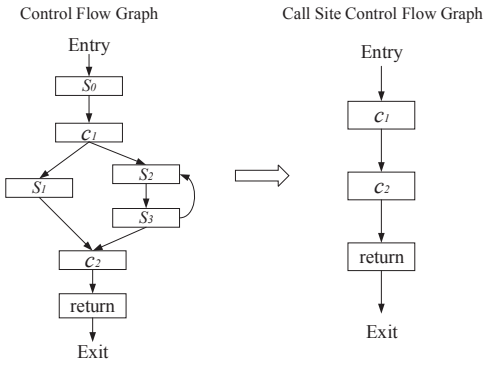



Figure 1: Call Site Control Flow Graph.

## 3. Call Trace Inference

In a nutshell, our approach first performs static analysis, instruments the selected call sites, and collects a *partial call trace* at runtime. In the offline stage, from the partial trace, our technique recovers the full call trace that could have been captured by instrumenting every call sites. The call trace inference is based on a grammar-based theoretical framework that models both the call traces in general and their reduced representation. The two models enable us to define the criteria of the deductibility of partial call trace and prove the correctness of our selection algorithm for instrumenting call sites.

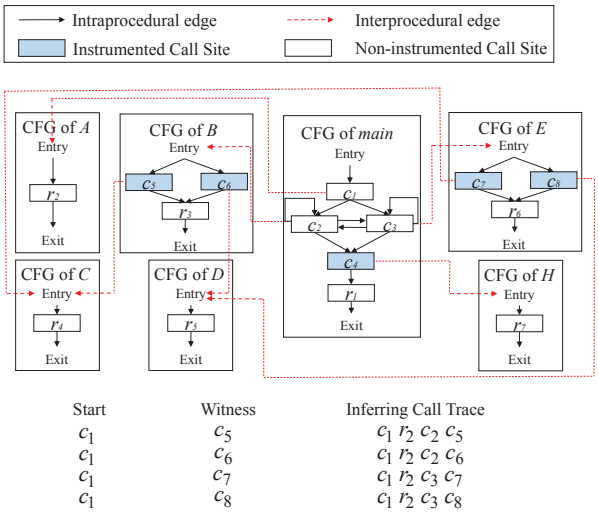
### 3.1 Intuition of Call Trace Inference

We illustrate the intuition of the call trace inference using the example in Figure 2. Suppose we only instrument call sites $\{c_4, c_5, c_6, c_7, c_8\}$ and the program execution follows the interprocedural edges denoted by the dotted arrows in Figure 3. When an instrumented call site is executed, its "witness", a unique ID of the call site, is logged. For our example, the witness log, ($c_5$, $c_5$, $c_7$), is a partial trace describing an instance of program execution.

We now explain how the full call trace ($c_1$, $r_2$, $c_2$, $c_5$, $r_4$, $r_3$, $c_2$, $c_5$, $r_4$, $r_3$, $c_3$, $c_7$, $r_4$, $r_6$, $c_4$, $r_7$, $r_1$) of the program execution can be inferred from the partial call trace ($c_5$, $c_5$, $c_7$). To infer the full trace, we first construct the *CSCFG*s as shown in Figure 3. The full call trace must start with the call site $c_1$ which is part of the entry function $main$. The call site following $c_1$ in the call trace is $r_2$ since $r_2$ is the unique successor of $c_1$ in the interprocedural control flow graph. After the return $r_2$ of function $A$, one of the successor call sites of $c_1$, i.e., $c_2$ or $c_3$, is to be generated in the call trace. To decide the successor of $c_1$, we read the witness $c_5$ from the partial call trace. As shown in Figure 3, $c_2$ invokes function $B$, while $c_3$ invokes function $E$. Therefore, the witnessed call site $c_5$ implies the execution of $c_2$ because executing function $B$ generates witness

```
1  void main(String[] args){
2    A();  // c1
3    do{
4      if (p1)
5        B();  // c2
6      else
7        E();  // c3
8    }while(p2);
9    H();  // c4
10   return;  // r1
11 }
12
13 void A(){
14   ...//no call sites
15   return;  // r2
16 }
17
18 void B(){
19   if (p3)
20     C();  // c5
21   else
22     D();  // c6
23   return;  // r3
24 }
25 void C(){
26   ...//no call sites
27   return;  // r4
28 }
29
30 void D(){
31   ...//no call sites
32   return;  // r5
33 }
34
35 void E(){
36   if (p3)
37     C();  // c7
38   else
39     D();  // c8
40   return;  // r6
41 }
42
43 void H(){
44   ...//no call sites
45   return;  // r7
46 }
```

Figure 2: A running example. $c$ for call site, $r$ for return site.




| Start | Witness | Inferring Call Trace |
|---|---|---|
| $c_1$ | $c_5$ | $c_1\ r_2\ c_2\ c_5$ |
| $c_1$ | $c_6$ | $c_1\ r_2\ c_2\ c_6$ |
| $c_1$ | $c_7$ | $c_1\ r_2\ c_3\ c_7$ |
| $c_1$ | $c_8$ | $c_1\ r_2\ c_3\ c_8$ |

Figure 3: An intuitive example of call trace inference.

$c_5$ or $c_6$ and function $E$ $c_7$ or $c_8$. As a result, we can infer that the prefix of the full call trace is ($c_1$, $r_2$, $c_2$, $c_5$), the only possible execution to generate the partial trace starting with $c_5$. Following the same deduction method, the full call trace can be reconstructed by extending the prefix of the call trace from the witnesses call sites.

We make two inspiring observations from the example above. First, if a call site $c$ has a unique successor $c_i$ in terms of control flow dependency, the execution of $c_i$ is implied by the execution of $c$. Second, if $c$ has multiple successors, we can leverage the witness of instrumented call sites to determine the executed successor. The call site inference relation is constructed to ensure we can uniquely generate the full call trace. Note that the call site inference relation in our work is different from the well-known domination relation on the Control Flow Graph [1]. The domination relation only guarantees that the execution of $u$ can be inferred by the execution of $v$, if call site $u$ dominates call site $v$. However, since there may be multiple call traces from $u$ to $v$, using domination relation alone does not guarantee the uniqueness in generating the original call trace.

### 3.2 The Construction of Call Trace Model

Call traces are dynamic in nature and determined by the execution path that a program takes. And, yet, we need to reason about them in a general way in order to develop inference methods. Our approach uses an abstract trace model to encode all possible traces incurred by programs.

Our call trace model is analogous to dynamic programming [11] as the representation of full traces is decomposed into a sequence of abstract representations of partial traces. That is, a (partial) call trace that starts with a call site $c_i$ in a function $f$ is comprised of three sequential parts: (1) $c_i$; (2) a call trace that is generated by executing the callee function $f_{c_i}$ called at $c_i$, denoted as $Func_{f_{c_i}}$ (if $c_i$ is a return site, $Func_{f_{c_i}}$ is empty); (3) a call trace that is generated by executing the call sites from the successor call sites of $c_i$ to the exit of $f$, denoted as $Suc_{c_i}$ (if $c_i$ is an exit call site, $Suc_{c_i}$ is empty). Except the first part, the other two parts are abstract representations of sub-traces. $Func_{f_{c_i}}$ can be represented as a call trace that starts with one of the entry call sites in $f_{c_i}$. Similarly, $Suc_{c_i}$ can be represented as a call trace that starts with one of the successor call sites of $c_i$. Thus, the above representation of partial call traces can be recursively defined. Let *entry* denote the entry function of the given program. A full call trace can be formally represented by $Func_{entry}$, which can be recursively derived from the representation of partial call traces. Based on the above intuition, we propose to model a program's call traces with a context-free grammar $G$ as follows.

**Definition 3.** *Call Trace Model* $G = \{V, \Sigma, S, R\}$ where:

1. $V$ is a finite set of non-terminals, each represents any call trace generated by executing from a call site (including the entry call site) of a function in the given program to the function's exit call site. $Func_f \in V$ and $Suc_{c_i} \in V$ for any function $f$ and any call site $c_i$ (excluding return sites) in the program. $Func_f$ denotes any possible call trace that can be generated by executing function $f$. $Suc_{c_i}$ denotes any possible call trace that can be generated by executing from a successor call site of $c_i$ to the exit call site of the function in which $c_i$ resides. $Func_f$ and $Suc_{c_i}$ are partial call traces of the whole program. The call trace of the whole program is composed of these two kinds of partial call traces.
2. $\Sigma$ is a finite set of terminals representing the possible call sites and return sites of which a call trace can be composed.
3. $S$ is a start symbol that denotes the abstraction of any call trace generated by executing the entry function.
4. $R$ is a set of production rules. The production rules in $R$ are defined in two forms.

**Production Rule Form 1 (Function Trace Rule):**
$\forall\ Func_f \in V, \forall\ c_i \in H_f$ and $c_i$ is a call site, the production rule is defined as:
$\boldsymbol{Func_f \rightarrow c_i Func_{f_{c_i}} Suc_{c_i}}$ .
$\forall\ Func_f \in V, \forall\ c_i \in H_f$ and $c_i$ is a return site, the production rule is defined as:
$\boldsymbol{Func_f \rightarrow c_i}$.
$Func_f$ denotes any possible call trace that can be generated by executing function $f$, $H_f$ denotes the entry call sites in the *CSCFG* of $f$, $f_{c_i}$ denotes the function called at call site $c_i$, and $Suc_{c_i}$ denotes any call trace that can be generated by executing from a successor call site of $c_i$ to the exit call site of the function in which $c_i$ resides. $Func_f$ and $Suc_{c_i}$ are partial call traces of the whole program. Note that, if $c_i$ which is a return site, we do not define a non-terminal $Suc_{c_i}$ ($Suc_{c_i} \notin V$) as well the production rules for $Suc_{c_i}$.
To facilitate the explanation, sometimes we use $\gamma_{c_i} (\gamma_{c_i} \in V^*)$ as the grammar symbol immediately following $c_i$ in the production rules. If $c_i$ is a return site, $\gamma_{c_i}$ is empty. If $c_i$ is a call site, $\gamma_{c_i}$ is $Func_{f_{c_i}} Suc_{c_i}$. Then, no matter $c_i$ is a return site or a call site, the production rule is represented as $\boldsymbol{Func_f \rightarrow c_i\ \gamma_{c_i}}$.

**Production Rule Form 2 (Call Site Successor Trace Rule):**
$\forall c_i \in \Sigma$ and $c_i$ is a call site, $\forall <c_i, c_j> \in E_f$ and $c_j$ is a call site, the production rule is defined as:
$\boldsymbol{Suc_{c_i} \rightarrow c_j Func_{f_{c_j}} Suc_{c_j}}$.
$\forall c_i \in \Sigma$ and $c_i$ is a call site, $\forall <c_i, c_j> \in E_f$ and $c_j$ is a return site, the production rule is defined as:
$\boldsymbol{Suc_{c_i} \rightarrow c_j}$.
$Suc_{c_i}$ represents any possible call trace that can be generated by executing the call sites from $c_j$(i.e. $c_j$ is a successor call site of $c_i$ in function $f$), and $E_f$ denotes the edges in the *CSCFG* of $f$. Other notations follow those in the Function Trace Rule above. $f_{c_j}$ is the callee function in the call site $c_j$, $Suc_{c_j}$ represents any possible call trace that can be generated by executing the call sites from the successor call site of $c_j$ to the exit call site of the function where $c_j$ is.
Similar to Production Rule Form 1, $\gamma_{c_i}(\gamma_{c_i} \in V^*)$ is also used to simplify the production rules. Then, no matter $c_i$ is a return site or a call site, the production rule is represented as $\boldsymbol{Suc_{c_i} \rightarrow c_j \gamma_{c_j}}$.

Based on the production rule form 1, we can generate production rules for each function shown in Figure 2. For example, for the $main$ function, there is a production rule $Func_{main} \rightarrow c_1 Func_A Suc_{c_1}$. More production rules are given in Figure 4 (a). Similarly, we apply the production rule form 2 to generate the production rules for each call site. Take the call site $c_1$ in Figure 2 as an example. There are two production rules for $Suc_{c_1}$: $Suc_{c_1} \rightarrow c_2 Func_B Suc_{c_2} | c_3 Func_E Suc_{c_3}$. More production rules are given in Figure 4 (a). Based on the production rule form 1 and 2, we can construct the call trace model $G$ for a given program. For example, Figure 4 (a) shows the production rules for the program in Figure 2.

### 3.3 LL(1) Grammar and Call Trace Model

An LL(1) grammar [11] is a formal language that can be parsed by the LL(1) parser. If the proposed call trace model $G$ is an LL(1) grammar, we can leverage the LL(1) parser to verify whether an arbitrary call trace is valid or invalid. In this paper, we treat a call trace as a valid call trace if it can be generated from a program execution.

To verify an LL(1) grammar, it relies on the two functions, *FIRST* [11] and *FOLLOW* [11]. $FIRST(\alpha)$, where $\alpha$ is any string of grammar symbols, denotes the set of terminals that appear as the first symbol of any string derived from $\alpha$. Note that, if $\alpha$ can derive an empty string, $\epsilon \in FIRST(\alpha)$. Given a non-terminal $A$, $FOLLOW(A)$ denotes the set of terminals that can appear immediately to the right of $A$ in some sentential form; that is, the set of terminals $T$ such that for each $\tau \in T$ there exists a derivation of the form $S \xrightarrow{*} \alpha A \tau \beta$, for some $\alpha$ and $\beta$.

Using the above two functions, an LL(1) grammar can be defined as follows.

**Definition 4.** A grammar $G$ is LL(1) if and only if (1) $G$ does not contain left-recursion rules($A \rightarrow A\alpha$). Whenever $A \rightarrow \alpha | \beta$ are two distinct production rules of $G$, the following constraints hold: (2) $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$. (3) At most one of $\alpha$ and $\beta$ can derive the empty string $\epsilon$. (4) If $\beta \xrightarrow{*} \epsilon$, then $\alpha$ does not derive any string beginning with a terminal in $FOLLOW(A)$. Likewise, if $\alpha \xrightarrow{*} \epsilon$, then $\beta$ does not derive any string beginning with a terminal in $FOLLOW(A)$.

With the definition above, we now verify if the proposed call trace model $G$ is an LL(1) grammar. Since every production rule in $G$ begins with a terminal symbol in the derivation, $G$ does not contain any left-recursion. In the call trace model $G$, whenever $N \rightarrow c_i \gamma_{c_i} | c_j \gamma_{c_j}$ are two distinct production rules ($N$ is a non-terminal symbol defined in either Production Rule Form 1 or Production Rule Form 2; $c_i$ and $c_j$ are two different call sites.), then we can get that, $FIRST(c_i \gamma_{c_i}) = \{c_i\}$ and $FIRST(c_j \gamma_{c_j}) = \{c_j\}$. Since both $FIRST(c_i \gamma_{c_i})$ and $FIRST(c_j \gamma_{c_j})$ are not empty and $FIRST(c_i \gamma_{c_i}) \cap FIRST(c_j \gamma_{c_j}) = \emptyset$, all the constraints of LL(1) grammar in Definition 4 are satisfied. Thus, the call trace model $G$ is an LL(1) grammar.

Being LL(1), the call trace model can be used by an LL(1) parser to parse any call trace of a program. The LL(1) parser leverages a top down and leftmost derivation strategy to parse a possible call trace generated by program execution. The selection of production rules plays a key role in each step of parsing call traces. The LL(1) parser uses a lookahead symbol from the leftmost and unparsed input to unambiguously determine which production rule is selected for derivation. For example, in Figure 4 (b), when the call trace is derived by $Func_{main} \rightarrow c_1 r_2 Suc_{c_1}$, there are two possible production rules to derive $Suc_{c_1}$ (Rules 10 and 11). Then, a lookahead symbol in terms of a call site $c_2$ (the leftmost and unparsed input symbol) is used to select one of the two rules. Once a production rule is chosen, we match the input symbols with the terminal symbols on the right of the production rule. Since the LL(1) parser is a predictive parser which is a recursive-descent parser with no backtracking, call traces can be parsed by the LL(1) parser deterministically.

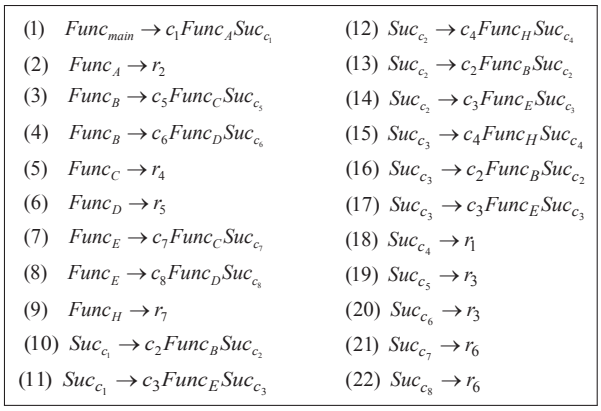
A *valid* trace is a trace that can be generated by a program execution. There are two types of valid call traces: ones generated by the normal termination of the program execution (e.g., Figure 4 (b)) and the ones by abnormal terminations (e.g., Figure 4 (c)). The trace ($c_1$, $r_2$) in Figure 4 (d) is an example of invalid call traces since it cannot be generated by any execution of the program in Figure 2. Figure 4(b)(c)(d) shows how valid and invalid call traces are parsed using the production rules in Figure 4 (a) and the LL(1) parser. Valid call traces in case (1) can be accepted by an LL(1) grammar in conventional ways when the string derived from the start symbol based on the LL(1) parser is the same as the input string. However, for any valid call trace in case (2), the LL(1) parser can only derive a sentential form which contains both terminals and non-terminals, and the input call trace matches the prefix of the derivation. To handle this, we slightly relax the validity criterion of a call trace in $G$ as follows: the input call trace is considered as *valid* if it matches the prefix of the derivation of $Func_{main}$. Under the relaxed criterion, the trace ($c_1$, $r_2$, $c_2$, $c_5$) in Figure 4 (c) is valid. The trace ($c_1$, $r_1$) in Figure 4 (d) is invalid because it cannot match any prefix of a derivation of $Func_{main}$ based on the LL(1) parser.

As a valid call trace can be accepted by the call trace model, we can leverage the model to directly deduce a set of call sites solving the call trace collection problem instead of enumerating all valid traces and verifying if a given set of call sites is able to collect them all.

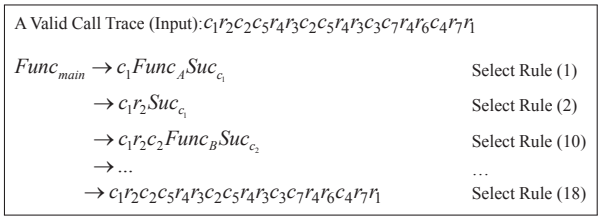
### 3.4 The Construction of Logged Call Trace Model

An instrumentation $\mathbb{I}$ is a set of instrumented call sites of a given program. Each logged call trace is a sequence of witnesses of the elements in $\mathbb{I}$. In this section, we show how a logged (partial) call trace is represented using the call trace model $G$.

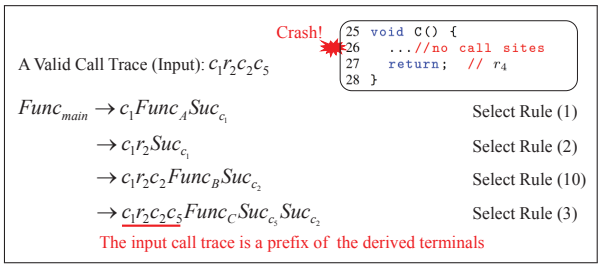
For any non-terminal $N$ in $G$ which represents a valid partial call trace, its corresponding logged call trace is denoted by $N'$. Given any call trace $N$ derived by a production rule $N \rightarrow c_i \gamma_{c_i}$, two situations arise with $c_i$. If $c_i \in \mathbb{I}$, we can leverage a similar production rule to derive the corresponding logged call trace of $N$: $N' \rightarrow c_i \gamma'_{c_i}$, where $\gamma'_{c_i}$ represents the corresponding logged call trace of $\gamma_{c_i}$. If $c_i \notin \mathbb{I}$, its execution is not logged. We can leverage a production rule to derive the corresponding logged call trace of $N$: $N' \rightarrow \gamma'_{c_i}$, where $\gamma'_{c_i}$ represents the corresponding logged call trace of $\gamma_{c_i}$. As such, we define a new grammar $G'$ to represent any valid logged call trace (a valid logged call trace is a logged call trace that can be generated from a program execution).



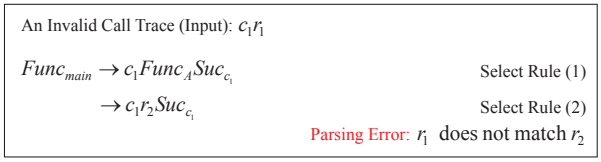

(a)




(b)




(c)




(d)

Figure 4: Examples of representing and parsing call traces based on production rules. (a) Production rules in the call trace model; (b) An example of parsing a valid call trace generated by a normal execution; (c) An example of parsing a valid call trace generated by an abnormal (crash) execution; (d) An example of parsing an invalid call trace.

**Definition 5.** Assume an instrumentation $\mathbb{I}$ and a call trace model $G$= $\{V, \Sigma, S, R\}$, the grammar $G'$= $\{V', \Sigma', S', R'\}$ for logged call traces is given as follows:

1. $V'$ is a finite set of non-terminals, each derives from $V$ and represents a logged call trace generated by executing from a call site (including the entry call site) of a function in the given program to the function's exit call site. $Func'_f \in V'$ and $Suc'_{c_i} \in V'$ for any function $f$ and any call site $c_i$ (excluding return sites) in the program. $Func'_f$ derives from $Func_f$ and represents the logged call trace of $Func_f$. $Suc'_{c_i}$ derives from $Suc_{c_i}$ and represents the logged call trace of $Suc_{c_i}$. The above derivation relation from $V$ to $V'$ is a bijection from $V$ to $V'$, since every element in $V$ is exactly paired with one element in $V'$, and vice versa.

2. $\Sigma' = \mathbb{I} \cup \{\epsilon\}$, where $\epsilon$ is an empty string.

3. $S'$ derives from $S$ and is a start symbol that denotes the abstraction of a logged call trace generated by executing the entry function.

4. $R'$ is a set of production rules deriving from $R$. For any production rule $N \rightarrow c_i \gamma_{c_i}$ $(N \in V)$ in $R$, if $c_i \in \mathbb{I}$, $N' \rightarrow c_i \gamma'_{c_i} (N' \in V')$ is derived as a production rule in $R'$, where $N'$ is the corresponding logged call trace of $N$, $\gamma'_{c_i}$ represents the corresponding logged call trace of $\gamma_{c_i}$; if $c_i \notin \mathbb{I}$, $N' \rightarrow \gamma'_{c_i}$ $(N' \in V')$ is derived as a production rule in $R'$, where $\gamma'_{c_i}$ represents the corresponding logged call trace of $\gamma_{c_i}$. Similar to $\gamma_{c_i}$, we use $\gamma'_{c_i}$ to simplify the explanation. If $c_i$ is a return site, $\gamma'_{c_i}$ is empty; if $c_i$ is a call site, $\gamma'_{c_i}$ is $Func'_{f_{c_i}} Suc'_{c_i}$ which is derived from $Func_{f_{c_i}} Suc_{c_i}$. Note that, if $c_i \notin \mathbb{I}$ and $\gamma'_{c_i}$ is empty, $N'$ does not derive anything, the production rule is defined as $N' \rightarrow \epsilon$. To ease the subsequent explanation of logged call trace model, let $N' \rightarrow \alpha_{c_i}$ denote the derived production rule of $N \rightarrow c_i \gamma_{c_i}$, no matter $c_i$ is in $\mathbb{I}$ or not. The derivation above relation from $R$ to $R'$ is a surjective function from $R$ to $R'$, since every production rule in $R$ is mapped to only one production rule in $R'$ and every production rule in $R'$ is derived from a production rule in $R$. Note that, two different production rules in $R$ may derive the same production rule in $R'$ (as discussed in Section 3.5), so the relation is not necessarily an injective function.

In Figure 5 (a), we give an example of the production rules in $R'$, which are derived from the production rules in Figure 4 (a) with a given instrumentation $\mathbb{I}$.

By Definition 5, each non-terminal $N$ in $V$ is mapped to only one derived non-terminal $N'$ in $V'$, and each production rule $R_n$ for $N$ in $R$ to only one derived production rule $R'_n$ for $N'$ in $R'$. These mapping relations of non-terminals and production rules enable us to precisely synchronize the steps of parsing a call trace to that of generating the corresponding logged call trace. Figure 5 (b) gives an example of how a logged call trace can be generated when parsing a full call trace using the LL(1) parser. The parsing of a call trace starts with $Func_{entry}$, while the generation of its logged call trace starts with $Func'_{entry}$. In each parsing step, we pick a non-terminal $N$ to continuously derive $Func_{entry}$ and a production rule $R_i$ to rewrite $N$. At the same time, we derive $Func'_{entry}$ by picking the $N'$ to which $N$ is mapped and applying the corresponding derived production rule of $R_i$ to rewrite $N'$.

Intuitively, by reversing this process of generating the logged call trace (i.e., to parse a logged call trace and generate its call trace simultaneously), we can recover the call trace and solve the collection problem. Motivated by this intuition, we study in the following the criteria of a feasible solution (i.e., instrumentation) to the call trace collection problem.

### 3.5 Criteria of Feasible Solution to Call Trace Collection Problem

An arbitrary instrumentation $\mathbb{I}$ may not guarantee that any call trace can be inferred from its logged call trace. For example, suppose $\mathbb{I}$ is $\{c_8\}$ for the program in Figure 2, $(c_1, r_2, c_2, c_5, r_4, r_3, c_4, r_7, r_1)$ and $(c_1, r_2, c_2, c_6, r_5, r_3, c_4, r_7, r_1)$ are two possible call traces that can be generated by executing this program. These two call traces fail to generate any useful witness information based on the call sites in $\mathbb{I}$. It is infeasible to infer the genuine call trace when we obtain insufficient witness information after executing the program. Thus, $\mathbb{I}$=$\{c_8\}$ is not a feasible solution to the call trace collection problem. This motivates us to define the criteria of a feasible solution to the call trace collection problem.

In Section 3.4, we have discussed how a logged call trace can be generated along with parsing its call trace. Inspired by that, we study whether it is feasible to apply the similar intuition to infer

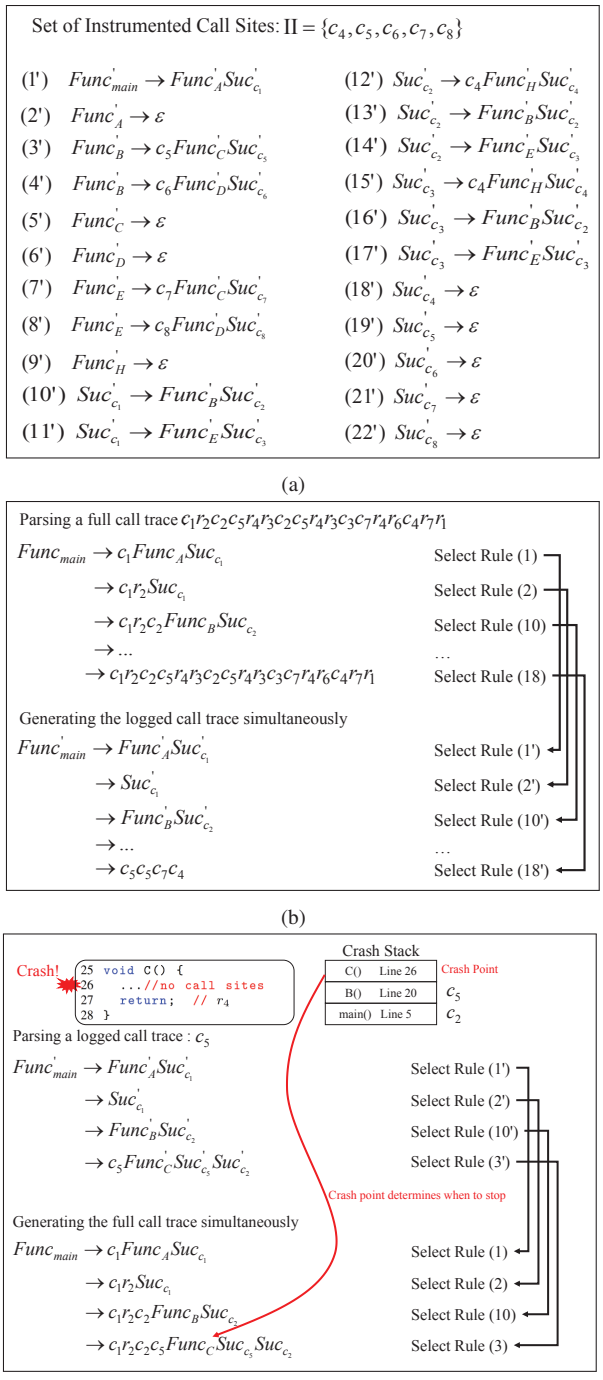



Figure 5: (a) The derived production rules in $R'$; (b) An example of generating a logged call trace when parsing its full call trace; (c) An example of generating a full call trace when parsing its logged call trace.

any full call trace from its logged call trace. Thus, we need to check whether any non-terminal $N'$ in $V'$ is mapped to only one original non-terminal $N$ in $V$, any production rule $R'_i$ in $R'$ is mapped to only one original production rule $R_i$ in $R$, and any step of parsing a logged call trace is mapped to only one step of generating original call trace.

By Definition 5, the derivation relation from $V$ to $V'$ is a bijection from $V$ to $V'$. Thus, any non-terminal $N'$ in $V'$ can be mapped to one and only one non-terminal $N$ in $V$ where $N'$ is derived from $N$.

Although any production rule $R_i$ in $R$ is mapped to only one derived production rule $R'_i$ in $R'$, $R'_i$ can be mapped from multiple production rules in $R$. In the following, we discuss all possible cases on how $R_i$ and $R_j$ derive the production rules in $R'$. (1) If $R_i$ and $R_j$ are used to derive different non-terminals, whatever $\mathbb{I}$ is, the deriving production rules $R'_i$ and $R'_j$ are different. This is because the non-terminals in the left side of $R'_i$ and $R'_j$ are different. (2) If $R_i$ and $R_j$ are used to derive the same non-terminal (i.e. $N \rightarrow c_i\gamma_{c_i}(R_i)\,|c_j\gamma_{c_j}(R_j)$ ) and $\gamma_{c_i}$ and $\gamma_{c_j}$ are not empty at the same time, whatever $\mathbb{I}$ is, $R'_i$ and $R'_j$ are different. This is because, no matter $c_i$ and $c_j$ are in $\mathbb{I}$ or not, $\gamma'_{c_i}$ in $R'_i$ and $\gamma'_{c_j}$ in $R'_j$ are different. (3) With the same condition as (2) except both $\gamma_{c_i}$ and $\gamma_{c_j}$ are empty, if at least one of $c_i$ and $c_j$ are in $\mathbb{I}$, $R'_i$ and $R'_j$ are different. This is because, $c_i$ and $c_j$ are different, $c_i$ and $\gamma'_{c_j}$ are different, and $\gamma'_{c_i}$ and $c_j$ are different. (4) With the same condition as (3) except both $c_i$ and $c_j$ are not in $\mathbb{I}$, $R_i$ and $R_j$ will derive the same production rule in $R'$, i.e. $N' \rightarrow \epsilon$. Therefore, if a production rule $R'_i$ in $R'$ is generated as case (4) based on an instrumentation $\mathbb{I}$, we cannot determine the original production rule of $R_i$ and generating the original call trace simultaneous with parsing the logged call trace is infeasible.

If a given $\mathbb{I}$ can satisfy the condition that each production rule in $R'_i$ in $R'$ is mapped to exactly one production rule $R_i$ in $R$, it enables us to map each step of parsing a logged call trace to one step of generating its original full call trace. Moreover, if any logged call trace can be parsed by a predictive parser (e.g., the LL(1) parser), the call trace that is simultaneously generated with the mapped steps is guaranteed to be the unique and genuine one. Therefore, how a logged call trace can be parsed plays an important role in solving the call trace collection problem. Note that, different instrumentations result in different grammar $G'$ which may be any class of context-free grammars, and not all the context-free grammars can be parsed deterministically. In other words, an instrumentation $\mathbb{I}$ affects how a logged call trace can be parsed.

There are two sufficient conditions for an instrumentation $\mathbb{I}$ to be a feasible solution with respect to a call trace model $G$. (1) Whenever there exist two production rules $N \rightarrow c_i|c_j$ in $G$, at least one of $c_i$ and $c_j$ must be in $\mathbb{I}$. (2) The derived grammar $G'$ based on $\mathbb{I}$ is a grammar that can be recognized by a predictive parser. Although there are many classes of predictive parsers, such as LL($k$) ($k = 1, 2, ...$), in this paper we restrict our study to use the LL(1) parser to parse logged call traces, since it is of practical interest and easy to construct. The methodology to study feasible solutions to the call trace collection problem using the LL(1) parser can be extended to other predictive parsers, such as LL($k$) ($k \geq 2$). Solving the call trace collection problem with minimal instrumentation is found to be non-trivial even under the restriction of using the LL(1) parser. It is an NP-hard problem as proved in Section 4. As any LL(1) grammar is also an LL(k) grammar ($k \geq 2$), solving call trace collection problem minimally using the LL(k) parsers ($k \geq 2$) subsumes the problem we studied and is at least an NP-hard problem.

Theorem 1 characterizes a class of feasible solutions to the call trace collection problem. Note that, if $G'$ is an LL(1) grammar, $G'$ must satisfy Definition 4, which covers the two sufficient conditions of a feasible instrumentation solution.

One important issue is the difficulty in verifying whether $G'$ satisfies the last constraint in Definition 4 when the program uses third-party libraries. A library function $f$ may implicitly invoke application functions (See Section 5.1). The library function is unavailable for static analysis. This may lead to incomplete result of finding *FOLLOW*. For example, a library API may invoke

two functions $f_1$ and $f_2$, respectively in some executions and an instrumented call site $c_i$ in $f_2$ may immediately follow the logged call trace $Func'_{f_1}$, which implies $c_i \in FOLLOW(Func'_{f_1})$. Since such information is difficult to obtain from static analysis, we would miss $c_i$ in $FOLLOW(Func'_{f_1})$. The incomplete result of *FOLLOW* function makes it difficult to verify the last constraint in Definition 4. To avoid that, we apply more restricted constraints as shown in Definition 6 to verify $G'$. Once the constraint (3) in Definition 6 is satisfied, the constraints (3) and (4) in Definition 4 are also satisfied. With the restricted constraints, we can directly leverage *FIRST* function to verify this constraint.

**Theorem 1.** An instrumentation $\mathbb{I}$ is a feasible solution to the call trace collection problem of a call trace model $G$, if the associated grammar $G'$ based on Definition 5 is an LL(1) grammar.

**Definition 6.** The grammar $G'$ based on Definition 5 is an LL(1) grammar if: (1) $G'$ does not contain left-recursion rules; and (2) for any two distinct production rules $N' \rightarrow \alpha_{c_i}|\alpha_{c_j}$ of $G'$, (a) $FIRST(\alpha_{c_i}) \cap FIRST(\alpha_{c_j}) = \emptyset$, and (b) none of $\alpha_{c_i}$ and $\alpha_{c_j}$ derive an empty string (i.e., $\epsilon \notin FIRST(\alpha_{c_i})$ and $\epsilon \notin FIRST(\alpha_{c_j})$).

With Theorem 1, let us examine the two types of call site inference relations identified in Section 3.1. First, if a call site $c_j$ is the only successor of a given call site $c_i$ (in either interprocedural or intraprocedural control flow), we can infer the execution of $c_j$ from $c_i$. Corresponding to $c_j$, there exists a production rule $R_k$: $N \rightarrow c_j\gamma_{c_j}$ in $G$. Here, $N$ is $Suc_{c_i}$ if $c_j$ is the successor call site of $c_i$; $N$ is $Fuc_{f_{c_i}}$ if $c_j$ is the successor of $c_i$ interprocedurally. Whenever we select $R'_k$ to parse a logged call trace, since $R'_k$ is only mapped to $R_k$, simultaneously we can use $R_k$ to derive the full call trace, which implies the execution of $c_j$. Second, if there are multiple successors of a given call site $c_i$ (in either interprocedural or intraprocedural control flow), it is feasible to determine which successor has been invoked by logged call sites. With respect to these $k$ successors, there exist $k$ ($k \geq 2$) production rules in $G$ as well as their $k$ derived production rules in $G'$. If $\mathbb{I}$ satisfies Theorem 1, when we parse a logged call trace and select a production rule $R'_k$ among the $k$ derived production rules, a logged call site is used to make the decision. Simultaneous with the decision of selecting $R'_k$, we use the original production rule of $R'_k$ to generate the full call trace, which implies how a logged call site can determine the successor by the logged call site. Therefore, the two types of call site inference are expressed by Theorem 1.

With Theorem 1 and Definition 6, we can verify whether the two instrumentations in Section 2 are feasible solutions to the call trace collection problem. By instrumenting all call sites and return sites, $G'$ is the same as $G$, which is an LL(1) grammar. By instrumenting the targets of the branch nodes in *CSCFG*, it guarantees that for any two production rules $N \rightarrow c_i\gamma_{c_i}|c_j\gamma_{c_j}$ in $G$, the derived rules are $N' \rightarrow c_i\gamma'_{c_i}|c_j\gamma'_{c_j}$. Both derived rules satisfy all constraints except the first one in Definition 6. The first constraint in Definition 6 can be violated in two situations: (1) there is only one production rule to derive $Func_f$, i.e., $Func_f \rightarrow c_i\ Func_f\ Suc_{c_i}$, where $f$ is invoked at $c_i$ and $c_i \notin \mathbb{I}$; and (2) there is only one production rule to expand $Suc_{c_i}$, i.e., $Suc_{c_i} \rightarrow c_i\ Func_{f_{c_i}}\ Suc_{c_i}$, where $Func_{f_{c_i}}$ can derive empty string and $c_i \notin \mathbb{I}$. These situations should never occur in a real program, since their occurrences imply a function $f$ recursively invokes itself at the entry call site without return or a call site resides within an infinite loop. Although these situations should never occur, we enhance the instrumentation in each situation by adding $c_i$ into $\mathbb{I}$ to break the left-recursions. As a result, the first constraint in Definition 6 is also satisfied. The enhanced instrumentations are feasible solutions to the call trace collection problem.

Once $\mathbb{I}$ is found to be a feasible solution based on Theorem 1 and Definition 6, we can generate the full call trace based on the logged call trace. Figure 5 (c) gives an example of how a full call trace is simultaneously generated with parsing a logged call trace. Same as the parsing of a full call trace, we adopt the same criteria of accepting a valid logged call trace. If a logged call trace is generated from a program execution with crash and it matches the terminals in the derivation of $Func'_{entry}$, we treat the logged call trace as valid. To determine when to stop parsing the logged call trace, we leverage the crash point information available from the crash stack. The parsing is stopped if there exists an interprocedural control flow path that traverses no call sites between the last call site in the derivation of $Func_{entry}$ and the crash point. Once we stop parsing the logged call trace, we also stop generating the full call trace. For example, in the last step of Figure 5 (c), from $c_5$ to the crash point, there is an interprocedural control flow path without going through any call site (or return site), so we stop parsing the logged call trace.

## 4. Minimizing Logged Call Trace

Theorem 1 tells us that a program can have multiple feasible instrumentation to collect call trace. In this section, we study how to find an optimal instrumentation that instruments minimal number of call sites, which leads to minimal logged call trace and incurs minimal runtime overhead. Though we prove that finding the optimal instrumentation is an NP-hard question, fortunately we can derive an $H_n$-approximate algorithm that works very well in practice.

### 4.1 Optimal Instrumentation

We define an optimal instrumentation for a program as an instrumentation with minimal number of instrumentation sites. Without knowing the execution frequency of every call site statically, this definition represents our best efforts to minimize the runtime overhead and logged trace size incurred by the instrumentation.

We prove that, even for a restricted subset of C-like programs, finding the optimal instrumentation is NP-hard. We call the subset programs $\mathcal{Y}$*-programs* and formally define it as follows:

**Definition 7.** A program is a $\mathcal{Y}$-program iff: (1) Its static call graph is acyclic, *i.e.* it does not have recursive calls or mutually recursive calls. (2) Every terminal function (i.e., the function that does not have callees) has more than one entry call sites.

The acyclic call graph guarantees that an optimal instrumentation for a $\mathcal{Y}$-program can be obtained by optimally instrumenting every function in the bottom-up manner on the call graph. We call this property *optimality condition*. The reason is the call trace produced by executing a function only involves the call sites of the transitively reachable callees (production rule form 1). Since a caller is unreachable from its callees on an acyclic call graph, the way to instrument a function does not depend on the way to instrument its callers. The optimality condition derives that the complexity of obtaining the optimal instrumentation is $O(nU)$, where $n$ is the number of functions in the program and $U$ is the complexity of optimally instrumenting a single function.

The rest of the proof requires a concept called *Call Site Conflict Graph* and it is defined as follows:

**Definition 8.** Given a call trace model $G$ and an instrumentation $\mathbb{I}$, a *Call Site Conflict Graph* (*CSCG*) of $f$ is an undirected graph $<V, E>$, where $V$ is a set of nodes representing call sites in $f$. There is an edge $e = <c_i, c_j> \in E$ iff: (1) $N \rightarrow c_i\gamma_{c_i}\ |c_j\gamma_{c_j}$ are two distinct production rules in $G$, with the derived production rules $N' \rightarrow \alpha_{c_i}\ |\alpha_{c_j}$ in $G'$; (2) $FIRST(\alpha_{c_i}) \cap FIRST(\alpha_{c_j}) \neq \emptyset$, or $\epsilon \in FIRST(\alpha_{c_i})$, or $\epsilon \in FIRST(\alpha_{c_j})$. Particularly, $CSCG^0_f$ is the *CSCG* for function $f$, where all transitively reachable callees of $f$ are instrumented except $f$.

From the definition of *CSCG*, we give a simpler way to decide a feasible solution for call trace collection in Theorem 2. The proof of

Theorem 2 is straightforward and hence we omit the proof to save space.

**Theorem 2.** Instrumentation $\mathbb{I}$ is a feasible solution defined in Theorem 1 iff the *CSCG* of every function is empty.

*CSCG* is a bridge for modeling our instrumentation problem as a graph problem and hence we derive our main results as follows.

**Theorem 3.** Optimally instrumenting a single function in a $\mathcal{Y}$-program is equivalent to vertex cover problem.

*Proof.* (*If part: Reduce vertex cover problem to the problem of finding an optimal instrumentation for a function*). Given a non-trivial undirected graph $G_{vc}$=($V$, $E$), where $|V| > 0$ and $|E| > 0$, we synthesize a $\mathcal{Y}$-program in three steps:

1. Construct a **main** function with $|V|$ entry call sites where the call site $c_i$ corresponds to the $i^{th}$ node in the graph $G_{vc}$. This can be easily constructed with a *if-else* statement that has $|V|$ branches. The $i^{th}$ branch contains call site $c_i$ that calls a distinct *broker function* $f_b^i$ (see step 3 for details).

2. Construct $|E|$ *terminal functions* where every terminal function $f_t^{ij}$ corresponds to an edge $<i,j>$ in graph $G_{vc}$ that connects two nodes $i$ and $j$. The terminal function contains a single *if-else* statement and two return sites under the *if* and *else* branches respectively.

3. The last step, we synthesize the broker functions. The functionality of broker function is dispatching the calls to terminal functions. For broker function $f_b^i$, we construct a *if-else* statement with $N_i$ branches, where $N_i$ is the number of edges incident to node $i$ in the graph $G_{vc}$. In the $j^{th}$ $(0 \leq j < N_i)$ branch, we insert a call site to call the terminal function $f_t^{ij}$.

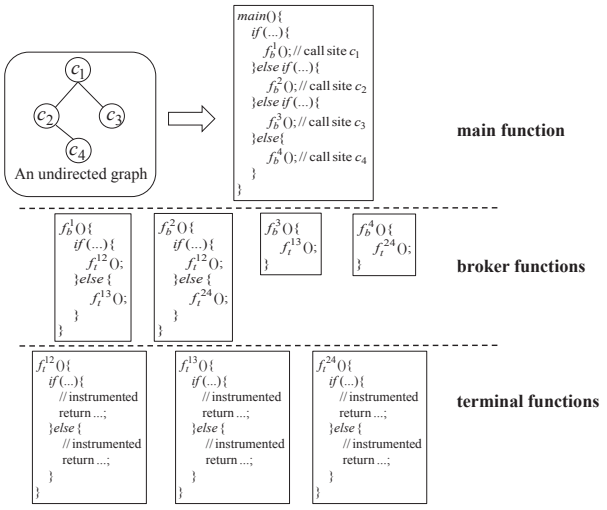



Figure 6: An example of reducing an undirected graph to a program.

We can easily verify the synthesized program is a $\mathcal{Y}$-program and the synthesis algorithm works in $O(|V| + |E|)$ time and space. An important characteristic of the synthesized program is that the $CSCG^0_{\textbf{main}}$ is exactly the graph $G_{vc}$. The reasons are:

1. Both the return sites in a terminal function $f_t^{ij}$ are instrumented. Otherwise we do not know how a terminal function exits.

2. None of the broker functions are instrumented. Suppose the derived production rule for every call site $c_i$ in the broker function is $\alpha_{c_i}$, we know that for any two call sites $c_i$ and $c_j$, $FIRST(\alpha_{c_i}) \cap FIRST(\alpha_{c_j}) = \emptyset$. This is because $c_i$ and $c_j$ call different terminal functions.

3. Two call sites $c_i$ and $c_j$ in the **main** function induce an edge in $CSCG^0_{\textbf{main}}$ iff there is an edge $<i,j>$ in $G_{vc}$. The key point is, from our synthesis algorithm step 3, the edge $<i,j>$ introduces a terminal function $f_t^{ij}$, which is called by both the broker functions $f_b^i$ and $f_b^j$. Since neither $f_b^i$ nor $f_b^j$ are instrumented, we conclude $I_{ij} \in FIRST(\alpha_{c_i})$ and $I_{ij} \in FIRST(\alpha_{c_j})$, where $\alpha_{c_i}$ and $\alpha_{c_j}$ are the derived production rules for call sites $c_i$ and $c_j$ in **main**. Therefore, $FIRST(\alpha_{c_i}) \cap FIRST(\alpha_{c_j}) \neq \emptyset$ and it introduces an $c_i \leftrightarrow c_j$ edge in $CSCG^0_{\textbf{main}}$. The proof for "non-$G_{vc}$ edge cannot induce an edge in $CSCG^0_{\textbf{main}}$" is similar.

Since $\epsilon \notin FIRST(\alpha_{c_i})$, $\forall c_i$ in **main** function, instrumenting $c_i$ eliminates all edges incident to $c_i$ in $CSCG^0_{\textbf{main}}$ without introducing new edges. This is the same to selecting node $i$ in $G_{vc}$ and covering the edges incident to $i$. Therefore, the optimal instrumentation for $CSCG^0_{\textbf{main}}$ is the answer to the vertex cover problem: node $i$ is selected iff the call site $c_i$ is instrumented.

(*Only if part: Reduce the problem of finding an optimal instrumentation to vertex-cover problem*). Given a function $f$, if $f$ is a terminal function, apparently we have to instrument all the return sites, otherwise we cannot know how the terminal function exits. Meanwhile, $CSCG^0_f$ is a complete graph, and the solution for vertex cover problem in this graph is selecting all vertices.

For a non-terminal function $f$, suppose all the callees of $f$ are optimally instrumented. We can draw two important conclusions. First, from the acyclic structure of $\mathcal{Y}$-program, any call site $c_i$ in $f$ can transitively call (at least) a terminal function. Since all terminal functions are instrumented, we know $\epsilon \notin FIRST(\alpha_{c_i})$. Second, $\forall c_i$, instrumenting $c_i$ only updates $FIRST(\alpha_{c_i}) = \{c_i\}$ and preserves $FIRST(\alpha_{c_j})$, $\forall c_j$. In other words, instrumenting $c_i$ eliminates all edges incident to $c_i$ without introducing new edges in the *CSCG*, which has the same effect of selecting a node and covering the incident edges in the vertex cover problem.

Therefore, adopting a vertex cover solution to instrument $f$ produces an empty *CSCG* of $f$, which is a feasible instrumentation according to Theorem 2. □

As a corollary, we conclude from Theorem 3 that the problem of finding an optimal instrumentation for a C-like program is NP-hard.

### 4.2 Suboptimal Instrumentation

Theorem 3 gives us opportunity to leverage the approximate algorithm for vertex cover problem to give a suboptimal instrumentation. But for an arbitrary program, we should transform it to a $\mathcal{Y}$-program to use the vertex cover results. The transformation can be conducted in two steps.

First, we eliminate the cycles in the call graph by instrumenting all the call sites that induce back edges identified by Tarjan's cycle finding algorithm [11]. After the cycle break, we can instrument the functions in a reverse topological order. For example, we can instrument our running example (Figure 2) in the order $C$, $D$, $A$, $H$, $B$, $E$, $main$.

Second, suppose we are instrumenting the function $f$ and all the callees of $f$ are instrumented. There are two cases that $f$ is not modeled as a $\mathcal{Y}$-program: (1) $f$ is a terminal function that has only one return site $c_i$; (2) $f$ is a non-terminal function that has a call site $c_i$ which invokes a terminal function as the case (1). In the case (1), according to Definition 8, $CSCG^0_f$ is empty, so no instrumentation in $f$ is required (See Theorem 2). In the case (2), any call site $c_i$ that makes $f$ not a $\mathcal{Y}$-program, has the property that $\epsilon \in FIRST(\alpha_{c_i})$. Thus, to transform $f$ into a $\mathcal{Y}$-program, we only need to guarantee that $\forall c_i$, $\epsilon \notin FIRST(\alpha_{c_i})$, where $\alpha_{c_i}$ is the derived production rule for the call site $c_i$. We call it *VC-transform* condition and specially instrument all the call sites $c_i$ where $\epsilon \in FIRST(\alpha_{c_i})$ to satisfy the condition. We record the specially instrumented sites in $\mathbb{I}_1$ for further optimization. Then, we apply the $H_n$-approximation algorithm (Ch.2 in [42]) to obtain the vertex cover for the *CSCG* of $f$ and use the selected call sites in vertex cover solution for

instrumentation. The approximation factor for our instrumentation problem is $H_d = \sum_{i=1}^{d} \frac{1}{i}$, where $d$ is the maximum degree of the vertices in *CSCG*. In practice, $H_d$ is usually very small.

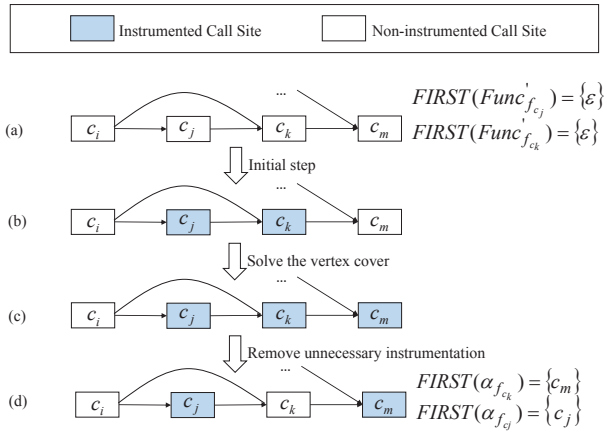



Figure 7: An example of instrumentation optimization. (a) A partial *CFG* before instrumentation; (b) Specially instrument the call sites to satisfy the VC-transform condition; (c) Add instrumentation from the vertex cover solution; (d) Remove some specially instrumented call sites and preserve the validity of remaining instrumentation.

We can further optimize $\mathbb{I}_1$ as some call sites in $\mathbb{I}_1$ can be removed without violating the constraints in Definition 6. Let us illustrate our idea by the example in Figure 7. Initially in Figure 7(a), none of the call sites are instrumented. At this step, $FIRST(Func'_{f_{c_j}}) = \{\epsilon\}$ and $FIRST(Func'_{f_{c_k}}) = \{\epsilon\}$, which violates the VC-transform condition. Therefore, we instrument $c_j$ and $c_k$ as described before (Figure 7 (b)) and run the approximate algorithm to instrument the rest of call sites. After this step, $c_m$ is instrumented (Figure 7 (c)). The last step, as shown in Figure 7 (d), we try to remove the instrumentation for $c_k$. According to Definition 6, we must guarantee, after removing the instrumentation for $c_k$, there is no violation of the constraints in Definition 6. For example, $FIRST(\alpha_{c_k}) = \{c_m\}$ and $FIRST(\alpha_{c_j}) = \{c_j\}$ after removing $c_k$ from the instrumentation. Therefore, $FIRST(\alpha_{c_k}) \cap FIRST(\alpha_{c_j}) = \emptyset$. Therefore, the instrumentation of $c_k$ can be removed. In general, we can check all the specially instrumented call sites one by one and removing all instrumentation sites that do not violate the constraints proposed in Definition 6.

The pseudo-code of our complete instrumentation algorithm is shown in Algorithm 1. The algorithm traverses and instruments the functions in a reverse topological order of call graph. It first specially instruments $\mathbb{I}_1$ the set of call sites that violate the VC-transform condition (Line 3 to 9 in Algorithm 1) and generate a $\mathcal{Y}$-program. Then, it builds the call site conflict graph of $f$ (i.e., $CSCG_f$) and applies the greedy algorithm to solve the vertex cover set of $CSCG_f$ (Line 9 to 12). Finally, the algorithm iteratively removes any call site $c_i$ in $\mathbb{I}_1$ such that removing $c_i$ will not violate the constraints in Definition 6 (Line 13 to 17). The final step is performed with a reverse topological order of *CSCFG* in $\mathbb{I}_1$, since the instrumentation of predecessor call sites relies on the instrumentation of successor call sites. Whether the instrumentation generated by Algorithm 1 is a feasible solution can be guaranteed by the Theorem 3 and the Theorem 1 with the constraints in Definition 6.

Section 2 discusses two related instrumentation approaches for call trace collection. We refer to the first approach [25] that instruments every call site baseline approach, and the second approach [24] that instruments the target call sites of branch nodes in the call site control flow graph branch-based approach. Apparently, the baseline approach is incapable to reduce the number of instrumented

**Algorithm 1:** Generate Instrumentation Solution

```
   Input: CallGraph: acyclic call graph (back edges have been handled)
   Output: 𝕀: the instrumentation solution
1  Function GenerateInstrumentationSol (CallGraph)
2      foreach function f in reverse topological order of CallGraph do
3          𝕀_1 ← ∅
4          foreach c_i in f do
5              if c_i violates the VC-transform condition then
6                  𝕀_1 = 𝕀_1 ∪ {c_i}
7              end
8          end
9          𝕀 = 𝕀 ∪ 𝕀_1
10         use 𝕀 to build the call site conflict graph of f as CSCG_f
11         𝕀_f ← Vertex-Cover Set of CSCG_f using Greedy Algorithm
12         𝕀 = 𝕀 ∪ 𝕀_f
13         foreach call site c_i ∈ 𝕀_1 in reverse topological order of CSCFG do
14             if IsRemovable(c_i) then
15                 𝕀 = 𝕀 - {c_i}
16             end
17         end
18     end
19 end
20 Function IsRemovable (c_i)
21     𝕀' = 𝕀 - {c_i}
22     if 𝕀' satisfies the constraints in Definition 6 then
23         return true
24     else
25         return false
26     end
27 end
```

call sites. The branch-based approach instruments both $c_i$ and $c_j$ whenever there is an edge $<c_i, c_j>$in the *CSCG*. In contrast, our approach considers the optimal solution on the vertex cover of *CSCG* and tries best efforts to optimize the specially instrumented sites. Therefore, we can guarantee the proposed suboptimal solution is better than the two reference approaches. For instance, by applying our approach to the example program in Figure 2, we only need to instrument $<c_4, c_5, c_6, c_7, c_8>$, which contains fewer call sites than the two reference approaches.

As discussed in Section 3.5, we enhance instrumentations to also guarantee the constraint left recursion free in Definition 6 under two situations: (1) a function $f$ recursively invokes itself at the entry call site without a return and (2) a call site is inside an infinite loop.

## 5. Implementation

We implemented our proposed approach in Casper (**Ca**ll trace**s** **per**egrination) framework for Java programs. Casper is implemented on top of Soot [23] and run static analysis such as control flow analysis and static call graph analysis on Soot Jimple IR. However, the instrumentation placement is performed on bytecode directly. At this moment, Casper is designed to collect call traces of a single thread and does not give promises to recover the thread interleaving. In below, we give the implementation details of Casper and show how Casper can work with real programs.

### 5.1 Handling Callback

Casper distinguishes application code (the code statically known to be reachable from entry functions) from library code (including the code in JDK, third-party libraries, non-Java code, and *etc.*). Casper only instruments the call sites in application code and gives up the instrumentation for library code. The reasons are twofold. First, due to the limitation of static analysis, certain code such as the dynamically loaded code or that invoked by reflection is unable to be processed in prior. Second, instrumenting the call sites in library classes is less cost-effective: they incur high runtime overhead but are of less interest to subsequent clients [7, 46].

However, in the presence of callback mechanism that allows library code to call application code, simply ignoring the call sites in library code poses the challenges to correctly reproduce the full call trace. Let's see an example in Figure 8. The internal control flow of both `main` and `AppMethod` are a straight line, none of the call sites are instrumented by Algorithm 1 and the instrumentation gives us an empty trace log for any execution. However, when we try to recover the full call trace, we have to answer two questions. The first question is whether `AppMethod` was executed. If the answer is positive, the second question is which call site ($c_1$, $c_2$, $c_3$, or $c_4$) transitively called `AppMethod`. Without the exact answers to the two questions, we are unable to recover the genuine call trace.

```
1  public class AppClass{
2    public void AppMethod(){
3      .......  // do something (no call sites)
4      return;  //c5
5    }
6
7    public static void main(String[] args){
8      Class c = Class.forName(args[1]);  //c1
9      Object object = c.newInstance();     //c2
10     Method m = c.getDeclaredMethod(args[2]); //c3
11     m.invoke(object);  //c4, call AppMethod
12   }
13 }
```

Figure 8: Challenges in trace recovery caused by function callback.

Our overall idea to handle callback is dynamically recording the call sites that can transitively call back to application code and the unique ID of the callback functions. For any call site $c_i$ in application code that may call a library function, we place a special instrumentation that manipulates a callback handling stack *CHS* for application code to witness the occurrence of function callback. The callback handling stack *CHS* is exemplified in Figure 9 (b), where each frame maintains a unique call site ID $c_i$ (or $\emptyset$) and the corresponding *ExpAppDepth* for $c_i$ (will explain later).

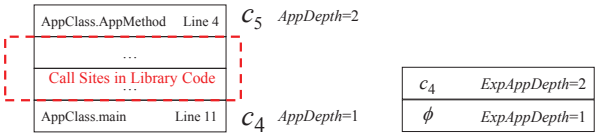



Figure 9: The snapshot of (a) the JVM system call stack and (b) the corresponding callback handling stack when `AppMethod` is on execution.

We define *AppDepth* for every application function on the JVM call stack as the number of other application functions that appear before it. Such as in Figure 9(a), the *AppDepth* for $c_4$ is 2, because we only count the `main` function and all other library functions are omitted. *AppDepth* can be easily tracked by using a thread local variable: whenever entering or leaving an application function, we increase or decrease *AppDepth* by 1 respectively. The functionality of *AppDepth* is to compute *ExpAppDepth* for every call site $c_i$ that can call a library function, where *ExpAppDepth* is the expected *AppDepth* for the first occurrence callback function transitively called by $c_i$. When $c_i$ is pushed to *CHS*, we set its *ExpAppDepth* $=$ *AppDepth* $+ 1$. For any other call sites $c_k$ that do not call a library function, we do nothing change to *CHS*. Therefore, the stack *CHS* is updated infrequently since only a small fraction of call sites can modify it. Subsequently, at the entry of each application function $f_{app}$, we check whether the current *AppDepth* equals to the *ExpAppDepth* in the top frame of *CHS*. If the answer is yes, we know $f_{app}$ is invoked through callback mechanism and then Casper logs $c_i$ and the unique ID of $f_{app}$.

We show how this dynamic mechanism works for the example in Figure 9 (b). First, the *AppDepth* for call site $c_4$ at Line 11 is 1. When `AppMethod` is executed through $c_4$, *AppDepth* is increased to 2, which is equal to the *ExpAppDepth* of $c_4$. By this means, we know `AppMethod` is a callback invocation and it is invoked by $c_4$ but not other call sites such as $c_1$, which answers the two questions asked previously.

Besides, this dynamic mechanism can also be used to automatically discover the entry function for each thread, which serves as the start point in our models $G$ and $G'$ to recover the full call trace. Whenever a thread is started, Casper creates a callback handling stack *CHS* for this thread and pushes an initial frame with *ExpAppDepth* $= 1$ to *CHS* (e.g., the bottom frame in Figure 9 (b)). Once a function is executed and it discovers the last *ExpAppDepth* is 1, this function is marked thread entry function and its unique ID is logged.

### 5.2 Handling Virtual Call

Virtual call sites occupy a very large portion in Java programs and hence, the performance gained by eliminating unnecessary virtual call sites instrumentation is dramatic. However, due to the dynamic nature of Java, it is unable to rely on a complete static call graph to decide which virtual call sites are safe to be removed from instrumentation. Our key insight is combining points-to analysis with the mechanism proposed to handle callback functions in Section 5.1 to safely handle all virtual call sites. We first use points-to analysis to find the virtual call sites that are possibly called on an object generated by newInstance function. This is achievable with a sound whole program analysis such as Soot SPARK. These call sites are instrumented and handled by the same mechanism described in Section 5.1. For the rest of the call sites, we first decompose the polymorphic call sites into a set of individual monomorphic call sites. For example, for a call site $c_v$ that has multiple call targets such as $f_1$, $f_2$, ... $f_k$, we generate $k$ artificial call sites $c_1$, $c_2$, ... $c_k$, where $c_i$ calls $f_i$. Then, we apply Algorithm 1 to make the instrumentation decision and instrument $c_v$ if any of its artificial call sites $c_i$ is decided to be instrumented. This approach handles polymorphic and monomorphic call sites uniformly and leverages Algorithm 1 to instrument minimal set of virtual call sites.

### 5.3 Handling Implicit Function Call by JVM

The method *finalize* and the static initializer <*clinit*> are implicitly invoked by JVM. The invocation of these methods is not modeled by the call trace model. To handle these methods, we instrument at their entry so that their executions are logged. In the recovering step, the call traces of these methods can also be recovered, but they are not used for the inference of the whole call trace.

### 5.4 Exception Handling

Both the call trace model and logged call trace model of a program rely on its call site control flow graph. In Casper's implementation, we utilize Soot to perform the control flow analysis by considering the exception handling paths, so the caught exceptions are modeled in $G$ and $G'$. For uncaught exceptions, we implement the *UncaughtExceptionHandler* and register it in JVM. When a thread crashes, our handler logs its crash stack and call trace. Therefore, we can model the stack unwinding via exception escape as a return site.

### 5.5 Call Site Encoding

To distinguish different call sites, we encode each call site using a unique integer identifier. It is sufficient to encode all call sites using an 32-bit integer in practice. For example, the number of call sites in our largest subject Eclipse is 143,114, which is far from hitting the upperbound of $(2^{32} - 1)$.

### 5.6 Asynchronous and Compressed Logging

Time overhead is a practical issue. The data of logged traces are saved in files so that full call traces can be regenerated later. Therefore, I/O operations are a main threat to time overhead. To tackle this, we shift heavy I/O operations from application threads to a separate I/O thread. The trace data logged by an application thread are cached locally in a memory buffer. When the buffer is full, the cached data are queued and saved to files by the separate I/O thread.

Space overhead is another practical issue. Although the compact call traces to be logged are much shorter than full call traces, the logged data could still be large. To reduce the space overhead, we applied the Java built-in library that implements the data compression algorithm "Deflate" [12] to compress the logged data.

## 6. Experiments

We have implemented three different instrumentation approaches to collect call traces: (1) instrumenting every call site, denoted as *FI* for short, (2) instrumenting only the target call sites of branch nodes in CSCFG, denoted as *BRI* for short, and (3) instrumenting call sites using *Casper*. The experiments were performed on a computer with an 3.30GHz Intel i3 CPU and 12.0GB of RAM, under Windows 7 Enterprise. The JVM used is Oracle's Java HotSpot(TM) 64-Bit Server VM with version 1.7.

We selected Dacapo Benchmark 2006 [6] as the evaluation subjects (Table 1). Table 1 shows the static information of each subject, including the number of call sites and returns, the number of library calls, and the number of virtual calls. We omitted the subject *bloat* in the benchmark due to an outstanding implementation problem in logging. The huge volume of generated log data by *bloat* causes an *OutOfMemoryError* exception. We compared the instrumentation overhead, time overhead and space overhead. To compare the time and space overhead, we ran each subject separately instrumented using *FI*, *BRI* and *Casper* for 100 times under the same default settings, and took the average as the final results. We conducted Mann-Whitney U-test [29] on each subject to test whether the performances of *FI*, *BRI* and *Casper* significantly differ.

| subject | #call sites & returns | #library calls | #virtual calls |
|---|---|---|---|
| antlr | 23,544 | 8,572 | 2,254 |
| chart | 26,003 | 9,275 | 2,483 |
| eclipse | 143,114 | 39,516 | 17,202 |
| fop | 26,944 | 8,316 | 3,227 |
| hsqldb | 17,870 | 4,549 | 1,028 |
| jython | 36,229 | 8,468 | 3,188 |
| luindex | 5,754 | 2,348 | 430 |
| lusearch | 6,892 | 2,743 | 434 |
| pmd | 27,737 | 6,723 | 2,684 |
| xalan | 28,629 | 8,093 | 2,941 |

Table 1: The experimental subjects.

### 6.1 Instrumentation Overhead Comparison

Table 2 reports two kinds of instrumented sites for both *BRI* and *Casper*. To guarantee that call traces will not miss any call site in the application functions, we propose a mechanism to handle virtual calls and callbacks. Since the instrumentation for checking virtual calls and callbacks does not directly log call sites, it is more lightweight than the instrumentation for logging. Table 2 reports the reduction of *BRI* and *Casper* with respect to *FI*, which instruments all call sites and returns. As the number of instrumented call sites correlates with the time and space overhead, we expect that *Casper* incurs less overhead.

### 6.2 Time Overhead Comparison

Figure 10 shows the comparison of time overhead among the three instrumentation approaches. *FI* resulted in the highest overhead, with an average of 213.9% (7.1% to 1617.5%). *BRI* reduced the overhead of *FI*, and only resulted in 128.4 % overhead on average, ranging from 6.7% to 808.3%. *Casper* outperforms the other two approaches. The overhead of *Casper* is 6.3% in the best case and 278.1% in the worst case, with an average of 68.0%. The time overhead of *Casper* is only 31.8 % of *FI*. The reduction of *BRI* over *FI* is ranging from 5.9% to 57.7%, and on average 29.7%. The reduction of *Casper* is more significant than *BRI*. The reduction of *Casper* over *FI* is 55.0% on average. Besides, the reduction of *Casper* over *BRI* is also significant, on average 35.1%. We conducted the Whitney U-test with a null hypothesis that *Casper* does not outperform the other approaches significantly. The results rejected the null hypothesis with a confidence level over 0.95 (i.e. $p$-values are less than 0.05), in all the subjects except *Eclipse*. In *Eclipse*, *Casper* is not significant better than *BRI* ($p$-value is 0.11). The main reason is that, the number of call sites that are logged in the execution of *Eclipse* is small and it does not impose much overhead. Table 3 also showed that the space size of logged data in *Eclipse* is smallest, compared with other subjects. The evaluation results showed that, the *Casper* can effectively reduce the time overhead.

| Subject | *BRI* | | | *Casper* | | |
|---|---|---|---|---|---|---|
| | #log | #check | %reduction | #log | #check | %reduction |
| antlr | 15,741 | 10,524 | 33.1% | 5,403 | 10,423 | 77.1% |
| chart | 15,484 | 9,655 | 40.5% | 8,451 | 9,609 | 67.5% |
| eclipse | 96,108 | 50,983 | 32.8% | 53,332 | 49,831 | 62.7% |
| fop | 16,444 | 10,135 | 39.0% | 8,085 | 9,805 | 70.0% |
| hsqldb | 10,420 | 5,146 | 41.7% | 6,050 | 5,059 | 66.1% |
| jython | 20,618 | 11,262 | 43.1% | 10,714 | 11,206 | 70.4% |
| luindex | 3,947 | 2,542 | 31.4% | 2,135 | 2,535 | 62.9% |
| lusearch | 4,750 | 2,985 | 31.1% | 2,560 | 2,966 | 62.9% |
| pmd | 18,147 | 8,365 | 34.6% | 10,489 | 8,023 | 62.2% |
| xalan | 18,400 | 9,981 | 35.7% | 10,843 | 9,538 | 62.1% |
| Average | 22,006 | 12,158 | 36.3 % | 11,806 | 11,900 | 66.4 % |

#log: the number of instrumented sites for logging.
#check: the number of instrumented sites for checking virtual calls and callbacks.
%reduction: the reduction over FI considering #log

Table 2: Instrumentation reduction.

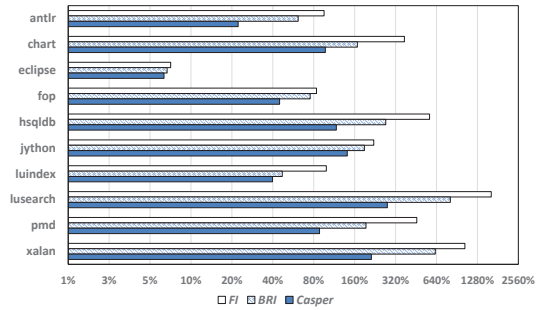



Figure 10: Time overhead comparison.

### 6.3 Space Overhead Comparison

Table 3 shows the size of traces logged by *FI*, *BRI* and *Casper*, as well as the reduction of *BRI* and *Casper* over *FI*. The space reported in the table is the size of the associated compressed log files. Like the results for time overhead, *FI* requires the largest space size. *Casper* achieved the highest reduction of the space overhead. The reduction ratio of *Casper* over *FI* is 63.9% on average, which is consistent with the instrumentation reduction ratio (66.4%). We also observed similar space reduction ratios of *Casper* over *FI* and *BRI*, in terms of the size of log files after decompression (See Table 4). Besides, we conducted the Whitney U-test with the null hypothesis that *Casper* does not achieve significantly less space overhead than the other two approaches. Our results rejected this hypothesis with a confidence level over 0.95 (i.e., $p$-values are less than 0.05), in all subjects. Overall, our experiments show that *Casper* can significantly reduce the space overhead.

| Subject | *FI* | *BRI* | | *Casper* | |
|---|---|---|---|---|---|
| | space size (KB) | space size (KB) | reduction | space size (KB) | reduction |
| antlr | 51,126 | 24,508 | 52.1% | 13,600 | 73.4% |
| chart | 65,782 | 30,154 | 54.2% | 17,809 | 72.9% |
| eclipse | 2,495 | 1,634 | 34.5% | 1,021 | 59.1% |
| fop | 22,488 | 13,581 | 39.6% | 8,790 | 60.9% |
| hsqldb | 78,973 | 35,999 | 54.4% | 24,989 | 68.4% |
| jython | 110,344 | 52,990 | 52.0% | 44,542 | 59.6% |
| luindex | 210,945 | 106,075 | 49.7% | 98,342 | 53.4% |
| lusearch | 298,387 | 119,853 | 59.8% | 81,171 | 72.8% |
| pmd | 136,002 | 62,057 | 54.4% | 43,621 | 67.9% |
| xalan | 339,505 | 198,421 | 41.6% | 140,702 | 58.6% |
| Average | 131,604 | 64,527 | 49.2% | 47,458 | 63.9% |

Table 3: Space overhead comparison (Compression).

| Subject | *FI* | *BRI* | | *Casper* | |
|---|---|---|---|---|---|
| | space size (KB) | space size (KB) | reduction | space size (KB) | reduction |
| antlr | 1,442,312 | 702,452 | 51.3% | 415,707 | 71.2% |
| chart | 3,384,046 | 1,703,534 | 49.7% | 1,080,787 | 68.1% |
| eclipse | 36,061 | 24,034 | 33.4% | 15,547 | 56.9% |
| fop | 310,398 | 189,445 | 39.0% | 125,188 | 59.7% |
| hsqldb | 3,231,980 | 1,630,190 | 49.6% | 1,143,102 | 64.6% |
| jython | 5,405,229 | 3,023,327 | 44.1% | 2,554,416 | 52.7% |
| luindex | 5,930,005 | 2,664,094 | 55.1% | 2,272,101 | 61.7% |
| lusearch | 5,741,900 | 3,158,803 | 45.0% | 2,628,250 | 54.2% |
| pmd | 6,158,416 | 3,127,378 | 49.2% | 2,353,579 | 61.8% |
| xalan | 4,771,881 | 2,928,493 | 38.6% | 2,094,564 | 56.1% |
| Average | 3,641,223 | 1,915,175 | 47.4% | 1,468,324 | 59.7% |

Table 4: Space overhead comparison (Decompression).

## 7. Related Work

**Program tracing and profiling**. Tracing and profiling program executions have wide applications, such as to identify performance bottlenecks [3, 10, 17, 31], to optimize code [13], to debug the program [21, 32, 45] and so on. Due to the importance of these applications, Ball and Larus [4] proposed instrumenting each call site with a "blocking" placement in order to trace and profile program executions. Later, they [5] proposed a path profiling algorithm, which encodes acyclic control flow paths of each function. Both works by Ball and Larus [4, 5] focused on capturing program behaviour within a function. In constrast, we address the call trace collection problem, which governs program behaviour across multiple functions. Larus later proposed *WPP* (whole program paths) technique [25] to capture and represent interprocedural control flow paths. Like Ball and Larus's approach [4], *WPP* instruments the entry and exit of each call site. As a result, the instrumented code suffers from large overhead. Our work improves *WPP* by adopting a more efficient way to instrument call sites. Other techniques [16, 47] have been proposed to utilize the power of multi-core systems to speed up the profiling and analysis process. These techniques shift heavy analysis tasks from application threads to non-application threads [16, 47] and parallelize these tasks [47]. This leads to faster analysis. Our work is orthogonal to them. It reduces both the runtime and space overheads. In future, we will consider integrating these techniques with ours to further improve the performance.

**Anomalous Program Behaviours Detection**. Call traces is important in detecting anomalous program behaviours. Some anomalous techniques [18, 39] characterized normal program behaviours from system call traces using different models, such as N-gram and automaton-based model. M. Christodorescu et al. [9] proposed to derive the specifications of malicious behaviours by comparing the system call traces of known malware with benign programs.

**Debugging**. Debugging is difficult and time-consuming. To assist developers, many debugging techniques were proposed. *BugRedux* [21] was proposed to reproduce the field failures based on different kinds of execution data, among which the call sequence is the most efficient one. *CrashLocator* [43] uses crash stack to approximate faulty crash traces and locates the crashing faults. Jiang and Su [20] proposed the context-aware statistical debugging and showed that relevant control flow paths that may contain bug locations are more informative than stand-alone bug locations. Besides statistical debugging, Ohmann and Liblit [35] proposed to leverage intraprocedural control flow paths to assist post-mortem analysis. Our work facilitates both bug reproduction and fault localization by providing an efficient approach to collecting call traces.

**Performance Diagnosis**. Finding performance problem is challenging. Many techniques were proposed to assist the diagnosis of performance problems. *StackMine* [17] mined patterns from a sequence of call stacks in large. The adoption of *StackMine* helped developers discover highly impactful performance bugs. Yu et al. [44] proposed a new approach to analyze the execution traces combining impact analysis an causality analysis. Mi [31] proposed *CloudDiag* to pinpoint the causes of the performance bugs by using a statistical technique and a fast matrix recovery algorithm. *CloudDiag* collected the sequence of method invocation and performed analysis on it. Since the trace data used in the above work is variant of call traces, our work can facilitate these techniques.

**Program Comprehension**. Program specifications are important for the understanding of the program. Gabel and Su [15] proposed to mine the temporal properties from call traces, which can be used for software development, bug detection and software maintenance. Pradel and Gross [37] proposed to learn the object usage specifications from a large volume of call traces. In another study, Pradel and Gross [38] mined the specification using call traces, integrated the specifications with test generation, and detected bugs at runtime. All the above works use call traces. As such, they can benefit from our efficient approach to collecting call traces.

## 8. Conclusion and Future Work

In this paper, we propose a novel LL(1) grammar based model to represent every possible call trace, as well as a grammar model to represent every possible logged call trace. Based on the models, we define the criteria of a feasible solution to the call trace collection problem, and theoretically prove that optimally solving this problem is NP-hard. Then, we design an efficient approach to obtaining a suboptimal solution, and implement the proposed approach as the tool Casper. Our experimental results show that Casper significantly outperforms existing approaches. It imposes on average 68.0% runtime overhead on the studied Java programs, which is only 31.8% of the overhead for collecting full call traces. Also, the size of the log data generated by Casper is only 36.1% of that generated by instrumenting all call sites.

This work presents the first theoretical results on the optimal instrumentation for call trace collection. Our approach makes an important step towards reducing the cost of collecting call traces. In the future, we will apply our approach to more projects and evaluate its effectiveness and possible applications in practice. We will also explore the techniques that can further reduce the overhead, such as the synergy between our approach and a sampling-based approach [27, 28]. Moreover, the methodology to study feasible solutions to the call trace collection problem using the LL(1) parser can be extended to other predictive parsers (e.g., LL(k) ($k \geq 2$)), which may further reduce the cost and deserves to be further explored.

## Acknowledge

This research is supported by the Hong Kong SAR RGC/GRF grants 611813 and 16212314, as well as NSFC grant 61272089. We thank the anonymous reviewers for their insightful comments on an initial version of this paper.

## References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley

Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *PLDI*, 1997.

[3] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP*, 2004.

[4] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. *TOPLAS*, 1994.

[5] T. Ball and J. R. Larus. Efficient Path Profiling. In *MICRO*, 1996.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[7] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *ACM SIGPLAN Notices*, 2007.

[8] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses. In *PLDI*, 2010.

[9] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC/FSE*, 2007.

[10] R. F. Cmelik, S. I. Kong, D. R. Ditzel, and E. J. Kelly. *An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks*. Springer, 1991.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[12] P. Deutsch. Rfc1951, deflate compressed data format specification, 1996.

[13] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *ACM SIGPLAN Notices*, 1984.

[14] M. Fowler. Refactoring: Improving the design of existing code, 1997.

[15] M. Gabel and Z. Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *FSE*, 2008.

[16] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *ACM SIGPLAN Notices*, 2009.

[17] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, 2012.

[18] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 1998.

[19] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *ICSE*, 1997.

[20] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *ASE*, 2007.

[21] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *ICSE*, 2012.

[22] V. Krunic, E. Trumpler, and R. Han. Nodemd: Diagnosing node-level faults in remote wireless sensor systems. In *MobiSys*, 2007.

[23] P. Lam, F. Qian, O. Lhotak, and E. Bodden. Soot: a java optimization framework. *See http://www. sable. mcgill. ca/soot*, 2002.

[24] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software: Practice and Experience*, 1990.

[25] J. R. Larus. Whole program paths. In *PLDI*, 1999.

[26] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *Usenix Security*, 1998.

[27] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *ACM SIGPLAN Notices*, 2003.

[28] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, 2005.

[29] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 1947.

[30] D. Melski and T. Reps. Interprocedural path profiling. In *CC*, 1999.

[31] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel & Distributed Systems*, 2013.

[32] B. P. Miller and J.-D. Choi. *A mechanism for efficient debugging of parallel programs*. ACM, 1988.

[33] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred call path profiling. In *OOPSLA*, 2009.

[34] A. Nistor and L. Ravindranath. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *ISSTA*, 2014.

[35] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *ASE*, 2013.

[36] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *EuroSys*, 2012.

[37] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE '09*, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[38] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE '12*, pages 288–298, Piscataway, NJ, USA, 2012. IEEE Press.

[39] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *S&P*, 2001.

[40] M. Serrano and X. Zhuang. Building approximate calling context from partial call traces. In *CGO*, 2009.

[41] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. *TSE*, 2012.

[42] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001. ISBN 3-540-65367-8.

[43] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim. Crashlocator: locating crashing faults based on crash stacks. In *ISSTA*, 2014.

[44] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS '14*, 2014.

[45] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *ACM SIGOPS Operating Systems Review*, 2006.

[46] Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu. Deltapath: Precise and scalable calling context encoding. In *CGO*, 2014.

[47] Q. Zhao, I. Cutcutache, and W.-F. Wong. Pipa: Pipelined profiling and analysis on multicore systems. *TACO*, 2010.

[48] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, 2006.