DeltaPath: Precise and Scalable Calling Context Encoding

Qiang Zeng†, Junghwan Rhee‡, Hui Zhang‡, Nipun Arora‡, Guofei Jiang‡, Peng Liu†
†Penn State University, ‡NEC Laboratories America

ABSTRACT

Calling context provides important information for a large range of applications, such as event logging, profiling, debugging, anomaly detection, and performance optimization. While some techniques have been proposed to track calling context efficiently, they lack a reliable and precise decoding capability; or they work only under restricted conditions, that is, small programs without object-oriented programming or dynamic component loading. These shortcomings have limited the application of calling context tracking in practice. We propose an encoding technique without those limitations: it provides precise and reliable decoding, supports large-sized programs, both procedural and objected-oriented ones, and can cope with dynamic class/library loading. The technique thus enables calling context tracking in a wide variety of scenarios. The evaluation on SPECjvm shows that its efficiency is comparable with that of the state-of-the-art approach while our technique provides precise decoding and demonstrates scalability and flexibility.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging-Monitors, Testing tools

General Terms

Performance, Measurement, Reliability

Keywords

Calling context encoding, object-oriented programming

1. INTRODUCTION

A *calling context* is the sequence of active function/method invocations that lead to a program location. It provides critical information about dynamic program behavior. The usage has been demonstrated in a wide range of applications, such as debugging [9, 11, 15, 18, 19, 20, 22, 23, 27, 28, 31, 32, 39, 37], event logging and error reporting [45, 25, 35], testing [12, 16, 24, 38], anomaly detection [21, 26, 44], performance optimization [29], and profiling [10, 33, 36, 34, 30, 43]. For example, system call event logging is critical for the analysis and diagnosis of the program execution in many production systems. Simply logging the system call events fails to record how program components interact when a system call is issued, while recording calling contexts would be very informative. Take profiling as another example; context sensitive profiling is powerful as it associates data such as execution frequencies, overhead and object life time with calling contexts, and thus provides precise information for program understanding and optimization [8].

It is straightforward to obtain calling contexts through stack walking, which, however, is expensive [6, 3]. A few encoding techniques, which represent a calling context using one or more integers, have been proposed to track calling contexts continuously with low overhead. Bond and McKinley [14] proposed a technique, *probabilistic calling context* (PCC), that computes a probabilistically unique integer ID, essentially a hash value, for each calling context. Although PCC encoding is efficient and compact, it does not provide decoding, which is essential to applications that require inspecting and understanding contexts, such as debugging, error reporting and event logging [41].

In order to enable the decoding capability, *Breadcrumbs* was built on PCC [13]. It collects additional dynamic information to assist decoding. Specifically, it records encoding values at relatively *cold* call sites. Depending on the threshold defining the hot code, the technique either incurs large overhead or sacrifices decoding accuracy and reliability. Besides, the decoding has to be offline because it involves expensive computation (their evaluation used the limit of 5 seconds) for recovering one context.

Computing calling contexts with low overhead and precise and reliable decoding capability is challenging. Recent progress was made by Sumner et al. [41], who proposed *precise calling context encoding* (PCCE) evolved from path profiling [10]. This technique iterates every possible calling context through static analysis and represents each using a unique encoding during runtime, so the context can be recovered precisely from the encoding.

However, as pointed out in previous research [13], PCCE would not work in the presence of virtual methods and dynamic class loading. Besides, handling large-scale software is a challenge to this technique, as it lacks a scalable solution to the encoding for a large number of calling contexts. More discussion about PCCE is in Section 2. The challenges limit the application of the encoding technique in a large range of scenarios, as a lot of software nowadays is written in object-oriented languages with dynamically loaded components and a large number of calling contexts .

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CGO '14, February 15 – 19 2014, Orlando, FL, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2670-4/14/02 ...$15.00.

This paper presents *DeltaPath*, an efficient calling context encoding technique with a precise decoding capability and the support for both procedural and object-oriented programs. Similar to PCCE, the technique leverages the Ball-Larus path profiling algorithm [10]. It obtains a unique encoding for each context at runtime by summing up the *addition values* of the call sites forming the context. Addition values are computed using our encoding algorithm based on static analysis of the target program, then an addition value and the addition operation are assigned to each call site through instrumentation. Unlike PCCE, which assumes small or medium-scale software without object-oriented programming or dynamically loaded components, DeltaPath does not have those limitations. It supports both procedural and object-oriented programming, allows dynamic class loading, and works with large-scale software.

DeltaPath resolves the limitations based on the following insights and ideas. The addition value of a call site is related to the number of calling contexts ending at it, while a *big* program usually contains a very large number of calling contexts, such that addition values may go beyond, i.e., overflow, the encoding integer. To avoid integer overflows, it is very inefficient to represent and operate on addition values using some class (e.g., `BigInteger` in Java). The insight is that a long calling context can be divided into several shorter pieces, each can be encoded using an integer. The DeltaPath encoding algorithm automatically finds a small number of functions acting as such *dividers* for the whole program, avoiding integer overflows systematically with low overhead. In addition, a virtual function call can be dispatched to many possible functions, while PCCE computes an addition value for each target. It is infeasible to insert a bulky `switch` statement at each virtual function call site and execute it to choose the addition value, for virtual function call sites are ubiquitous and frequently invoked in object-oriented (OO) programs. Our encoding algorithm computes a single addition value for one call site to minimize the code cache pressure and execution slowdown. Moreover, dynamically loaded classes introduce calling contexts that are not considered during static analysis. A call path tracking technique is designed to detect unexpected calling contexts and keep the encoding correct.

We implemented DeltaPath and performed experiments with a variety of Java programs. Our evaluation results show that its performance is comparable with that of PCC [14], which is a highly efficient and the state of the art calling context encoding technique capable of working in the presence of object-oriented programming and dynamic class loading. Compared to PCC, DeltaPath introduces precise and reliable decoding.

We made the following contributions.

- We propose a new precise calling context encoding technique that supports both procedural and OO programs.
- Encoding space pressure is addressed systematically. It thus allows the encoding technique to be applied to large-scale software.
- Dynamic class loading is handled properly using the call path tracking technique.
- We implemented DeltaPath and evaluated it on a variety of Java programs. The efficiency of DeltaPath is comparable with that of PCC while it provides precise and reliable decoding.

The rest of this paper is organized as follows. Section 2 presents the encoding background. Section 3 presents DeltaPath. Section 4 discusses several practical issues. The implementation details and evaluation results are presented in Section 5 and Section 6, respectively. The related work is discussed in Section 7, and the future work in Section 8. The paper is concluded in Section 9.

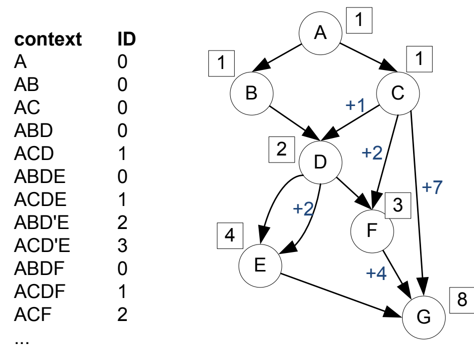

Figure 1: Example for PCCE encoding. Edge annotations are addition values; an addition value "*+c*" means "*ID+=c*" is executed before the invocation and "*ID-=c*" is executed after; the superscript on `D` (`D'`) disambiguates two call sites in `D` both invoking `E`.

2. BACKGROUND

Ball and Larus proposed an efficient algorithm (referred to as the BL algorithm) to encode intra-procedural control flow paths [10]. Each of the paths leading from the entry of a function to the end of it obtains a unique encoding. The algorithm has become canonical in control flow encoding and path profiling.

PCCE [41, 42] leverages the BL algorithm and adapts it to encoding calling contexts, which are essentially inter-procedural paths in a call graph. PCCE encodes the calling context using a small number of integer identifiers (IDs), ideally one. It instruments function calls with additions to an integer ID such that *the calling context ending at any program point can be uniquely represented by the ID along with the program counter of the point*.

The algorithm calculating addition values consists of two steps of analysis of the target program's call graph. First, it computes the *number of calling-contexts* (NC) of each node in the call graph by summing the NC of each of the nodes' predecessors (the NC of `main` is 1). Second, with respect to each node, the addition value for the first edge is 0, and for each of the rest edges the addition value is the sum of the NCs of the predecessors appeared in the previously processed edges.

Consider the call graph in Figure 1. The annotation of each node indicates its NC. `D`'s NC= 2, for example, is the sum of the NCs of `B` and `C`, denoting there are two possible contexts when `D` is invoked. The number along each edge is the addition value. Some edges do not have such numbers, meaning the addition values are 0. `CG`'s addition value is calculated after `EG` and `FG`, thus it is the sum (7) of the NC of `E` (4) and that of `F` (3).

Take the context `ACFG` as an example for encoding, the ID increases along `CF` and `FG`, respectively, so the result is 6. The table in Figure 1 shows the encodings of other calling contexts. Some contexts have the same ID values, for example, `AB` and `AC`. It is fine because an encoding is represented by both the ID and the ending node.

Given an encoding, we can precisely recover the calling context from bottom to top. Consider the ID 6 obtained at node `G`, for example, the edge whose addition value is the greatest but not greater than the ID value is taken, that is, `FG`. We then jump to node `F` meanwhile decrease the ID by the addition value, 4, for `FG`. The

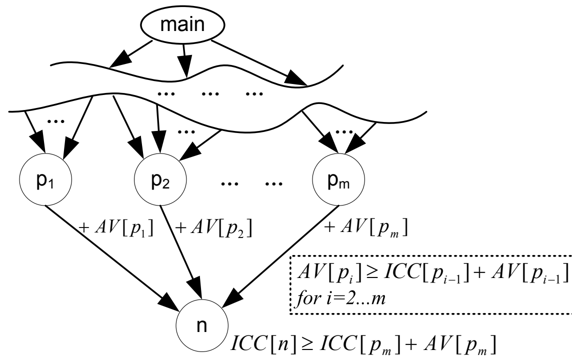

Figure 2: Intuition of our algorithm.

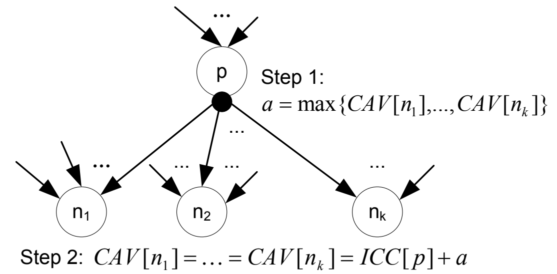

Figure 3: Intuition of encoding with dynamic dispatch. A virtual function call in p can be dispatched to multiple methods n_1, n_2, ... , n_k. The candidate addition value of a node n_i, denoted by $CAV[n_i]$, is initially 0, and it is updated each time the addition value of an edge leading to n_i is calculated. We process the call in two steps. First, the addition value $a = max\{CAV[n_1], \ldots, CAV[n_k]\}$ is obtained; second, $CAV[n_i]$ is updated as $ICC[p] + a$, for $i = 1, \ldots, k$. The updated $CAV[n_i]$ is then used to calculate the addition value for the next incoming edge of n_i.

decoding continues with `F` and ID value 2, from which we can recover edges `CF` and `AC` the same way.

If cycles exist in the call graph, which implies there are recursions in the program, a recursive call path is divided into acyclic sub-paths, each of which is encoded separately, such that a stack of IDs is used to represent a call path of recursions. We refer the readers to [42] for more details. Since recursions are handled in the same way by our technique, we omit its discussion and assume acyclic call graphs in the rest of the paper.

As a first step towards calling context encoding that allows precise decoding, PCCE works under restrictive conditions: programs in procedural languages without dynamically loaded components or high encoding space pressure. This has limited the application of PCCE in a variety of scenarios. Our goal is to deliver a calling context encoding technique without those limitations.

3. DELTAPATH ENCODING

This section presents DeltaPath. Section 3.1 covers encoding in the presence of virtual function calls, while Section 3.2 considers encoding space pressure along with object-oriented programming and describes a systematic solution.

3.1 Encoding for Object-oriented Programs

With object-oriented programming a function call can be dispatched to multiple targets. Such polymorphism complicates encoding, as a call site may have conflicted addition values due to the multiple dispatch targets. For example, assume edges `D'E` and `DF` in Figure 1 are due to the same virtual function call site. According to the PCCE algorithm, the addition values for edge `D'E` and `DF` are 2 and 0, respectively.

It is straightforward and tempting to choose the addition value based on the dynamic dispatch result using a `switch` statement, which, however, will significantly increase the code to be inserted and slow down the program execution, as virtual function call sites are massive and frequently invoked in OO programs.

In order to minimize the encoding overhead, our goal is to calculate a single addition value for each call site. However, addition values generated by the PCCE algorithm do not work for the purpose. For example, if 2 is used as the single addition value in the case above, the calling contexts `ABDF` and `ACF` will both be encoded to 2, which violates the principle of a unique encoding for each different context. The computation of the addition values thus requires a new encoding algorithm.

The upper bound of the encoding space representing the calling contexts ending at a node is called its *inflated calling-context count* (ICC). That is, the calling contexts of a node n are encoded using the integers in $[0, ICC[n])$. The basic idea of our algorithm is to ensure the invariant that *for any given node, its encoding space is divided into disjoint sub-ranges, with each sub-range encoding calling contexts along one incoming edge of the node.* Figure 2 illustrates the intuition behind the idea. Assume for a given node n, it has totally m incoming edges. The addition value along $p_i n$ is denoted by $AV[p_i]$.[1] The invariant is kept if the following conditions are satisfied: (1) $AV[p_1] \geq 0$ and $AV[p_i] \geq ICC[p_{i-1}] + AV[p_{i-1}]$ for $i = 2, \ldots, m$, and (2) $ICC[\texttt{main}] = 1$ and $ICC[n] \geq ICC[p_m] + AV[p_m]$ if $n \neq \texttt{main}$.

It can be proved using induction. The encoding ID is initialized as 0 at the `main` function, and $ICC[\texttt{main}] = 1$; the encoding ID is in $[0, ICC[\texttt{main}])$, thus the invariant is satisfied at `main`. Assume all the predecessors of n satisfy the invariant. The calling contexts ending at n are encoded into m disjoint sub-ranges: $[AV[p_1], ICC[p_1] + AV[p_1])$ encodes the contexts along edge $p_1 n$, and generally, $[AV[p_i], ICC[p_i] + AV[p_i])$, for $i = 2, \ldots, m$, encodes the contexts along $p_i n$, where $AV[p_i] \geq ICC[p_{i-1}] + AV[p_{i-1}]$, that is, $AV[p_i] \geq$ the upper bound of the sub-range encoding the contexts along $p_{i-1} n$. Plus, according to condition (2), all the sub-ranges fall in the range $[0, ICC[n])$.

Figure 3 illustrates how ICCs and addition values are computed satisfying the conditions above. The virtual function call inside p can be dispatched to $n_1, n_2, \ldots, n_k$. Each node n_i is associated with a variable $CAV[n_i]$, denoting the *candidate addition value* that should be considered when computing the addition value for the next incoming edge of n_i. $CAV[n_i]$ is initially 0 and keeps updating along with the calculation of the addition value. In this case, after the addition value for the call site is assigned as $a = max\{CAV[n_1], \ldots, CAV[n_k]\}$, the candidate addition values are updated as $CAV[n_i] = ICC[p] + a$. Later the updated $CAV[n_i]$ is used to calculate the addition value for the next incoming edge of n_i. By computing the addition value of a call site as the maximum value among the candidate addition values of the nodes that the call site can be dispatched to and updating the CAVs using the chosen addition value, condition (1) described above is satisfied. After the addition value for the last incoming edge of a node n is calculated and

[1] An addition value is associated with a call site, so more precisely it should be denoted as $AV[p_i n]$; we use $AV[p_i]$ instead for the sake of conciseness.

Algorithm 1 Encoding with dynamic dispatch

1: **function** ENCODING(N, E)
2: $ICC[\texttt{main}] \leftarrow 1$
3: **for** $n \in N$ **do**
4: $CAV[n] \leftarrow 0$
5: **for** $n \in N$ in topological order **do**
6: **for** $e = \langle p, n, l \rangle$ of the incoming edges of n **do**
7: $cs = \langle p, l \rangle$
8: **if** $cs \in processedSites$ **then**
9: continue
10: $processedSites \leftarrow processedSites \cup \{cs\}$
11: $AV[cs] \leftarrow CalculateIncrement(cs)$
12: **if** $n \neq \texttt{main}$ **then**
13: $ICC[n] \leftarrow CAV[n]$
14: **function** CALCULATEINCREMENT($cs = \langle p, l \rangle$)
15: $a \leftarrow 0$
16: **for each** $e = \langle p, n, l \rangle$ dispatched from cs **do**
17: **if** $CAV[n] > a$ **then**
18: $a \leftarrow CAV[n]$
19: **for each** $e = \langle p, n, l \rangle$ dispatched from cs **do**
20: $CAV[n] \leftarrow ICC[p] + a$
21: **return** a

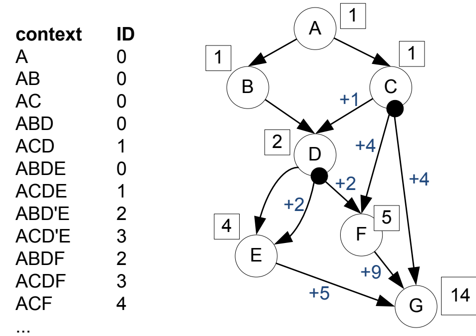

Figure 4: Example for encoding with dynamic dispatch. The superscript on D (D′) disambiguates two call sites in D both invoking E. D′E and DF are due to a virtual function call in D, and CF and CG are due to a virtual function call in C. Node annotations are ICC values.

$CAV[n]$ is updated for the last time, $ICC[n]$ is assigned as $CAV[n]$, satisfying condition (2).

Algorithm 1 shows the encoding algorithm. The input of the algorithm is the call graph of the target program, $CG = \langle N, E \rangle$, where N is a set of nodes with each representing a function and E a set of directed edges. Each call edge $e \in E$ is a triple $\langle n, m, l \rangle$ where $n, m \in N$, are the caller and callee, respectively, and $\langle n, l \rangle$ is a call site potentially invoking m. In Java, for example, l is the byte code index of the call site in n. A call edge in our algorithm is modeled as a triple instead of a caller and callee pair in order to distinguish multiple call sites in the caller that may invoke the same callee. In the encoding examples below we omit l and simply use nm to denote an edge for the sake of simplicity, though.

In the beginning, $ICC[\texttt{main}]$ is set to 1 (Line 2), and the CAV of each node is initialized to be 0 (Line 3–4). Then the nodes are visited in a topological order; a node is visited after all its predecessors have been visited. For each non-main node n, $ICC[n]$ is assigned (Line 13) after all its incoming edges are processed (Line 6–11). For each incoming edge of the node being visited, the algorithm identifies its call site (Line 7), and determines whether it has already been processed by searching the call site in *processedSites*, which stores all the call sites that have been processed; as multiple edges can be due to one call site, this ensures that the algorithm processes each call site only once. Function CALCULATEINCREMENT calculates the addition value for the call site and then updates the CAV of each of the nodes that the call site can be dispatched to. Due to the topological traversal order, when node n is processed, the *ICC* values of its predecessor nodes are already assigned, so the update of $CAV[n]$ can refer to $ICC[p]$ without problems (Line 20).

Consider the example in Figure 4. $ICC[\texttt{A}]$ is set to be 1 (Line 2), and all the CAVs are initialized as 0 (Line 3–4). Upon the visit of B, the addition value for AB is $CAV[\texttt{B}] = 0$ (Line 18), then $CAV[\texttt{B}] = ICC[\texttt{A}] + 0 = 1$ (Line 20) and $ICC[\texttt{B}] = CAV[\texttt{B}] = 1$ (Line 13). Node C is processed similarly. Then the topological traversal reaches D. BD is first processed, so $CAV[\texttt{D}] = 0$ is used as the addition value. Next, $CAV[\texttt{D}]$ is updated as $1 (= ICC[\texttt{B}] + 0)$, which is then used as the addition value for the next incoming edge CD, and $CAV[\texttt{D}]$ is updated as $2 (= ICC[\texttt{C}] + 1)$ (Line 20) . As CD is the last incoming edge of D, $ICC[\texttt{D}]$ is assigned as $2 (= CAV[\texttt{D}])$ (Line 13).

The traversal then visits node E. After DE is processed, $CAV[\texttt{E}] = ICC[\texttt{D}] + 0 = 2$ (Line 20) . Next, the last incoming edge of E, D′E, is processed. Note that the two edges D′E and DF are due to the same call site, and $CAV[\texttt{F}] = 0$. The addition value for this call site is hence calculated as $max\{CAV[\texttt{E}], CAV[\texttt{F}]\} = 2$ (Lines 16–18). Then $CAV[\texttt{E}]$ and $CAV[\texttt{F}]$ are updated as $4 (= ICC[\texttt{D}] + 2)$ (Lines 19–20). After that, $ICC[\texttt{E}]$ is updated as $CAV[\texttt{E}] = 4$ (Line 13). Other nodes are similarly processed following the algorithm. In the end, each call site in Figure 4 obtains a single addition value, while each calling context can be encoded uniquely.

In order to accommodate virtual function calls, the algorithm inflates the number of calling contexts (NC) of a node and uses it as the upper bound of its encoding space, which is thus termed as the inflated calling-context count (ICC). For example, $NC[\texttt{F}] = 3$, while $ICC[\texttt{F}] = 5$; the gap between the two enables a uniform addition value 2 for the virtual call site leading to edges D′E and DF. It is interesting to note that when there is no virtual function in a program, $ICC[n] = NC[n]$ for any given node n, which is the case in the PCCE encoding for procedural programs.

The decoding remains the same (Section 2), so it is omitted.

3.2 Encoding for Large-scale Object-oriented Programs

In both PCCE and Algorithm 1, the addition value at a call site reflects the number of calling contexts ending at the call site, while the number of calling contexts grows exponentially with the size of a call graph. Thus the computation of addition values during *static analysis* can incur integer overflows. Moreover, *runtime* integer overflows may also occur when computing the encoding ID by summing up addition values.

The runtime integer overflows can be easily resolved despite some cost: before each addition operation, the encoding judges whether an integer overflow can occur; if so, the current ID value is pushed onto a stack, and the ID is reset to 0 before the encoding continues. The encoding result is then represented as a stack of IDs along with the current ID value. The integer overflow problem during static analysis is more challenging. Implementation using some *big integer* classes, for example, Java library provides the `BigInteger` class supporting representation of huge integers, is straightforward but would be very inefficient, as addition values are represented as objects and during runtime each addition operation becomes a function call. Our goal is to completely avoid the

runtime integer overflow and resolve the integer overflow during static analysis without using big integer classes.

While PCCE achieves this goal by pruning edges during static analysis to ensure that the resultant call graph can be encoded by a single integer, the technique is not scalable for encoding large-sized programs, as massive edges at the deep portion of the call graph would be pruned and the pruned edges are handled at a relatively high runtime cost the same way a runtime integer overflow is processed as aforementioned. We aim at a scalable and efficient solution to the integer overflow problem.

The observation is that so far all calling contexts begin at the `main` function, such that the encoding space keeps growing when encoding those calling contexts ending at the deep portion of a call graph. So instead of encoding a calling context as a whole, we *divide it into pieces, each of which can be encoded using an integer without overflows*; it implies that each addition value can be represented by an integer, since the encoding value of each piece is the sum of the addition values within the piece.

We thus design an advanced version of the encoding algorithm, which chooses a set of *anchor* nodes dividing all long calling contexts in a program into shorter pieces. Each piece begins at an anchor node and is encoded relative to it. During runtime the encoding maintains a stack. When the program invokes an anchor node *p*, the current encoding ID value along with the anchor node's identifier is pushed onto the stack, then the ID is reset to 0 and the encoding continues. When the invocation of *p* returns, the ID is recovered with the popped ID value. In this way the previously global encoding space pressure is distributed along anchor nodes, and each calling context piece can be encoded separately and locally. The anchor nodes virtually act as *barriers* that keep the encoding space pressure from flowing *downstream* along the call graph.

There are two challenges to be resolved when designing the algorithm. First, it needs to find the anchor nodes. Second, given a call site, there exist multiple calling context pieces all reaching the call site while starting from different anchor nodes; hence, multiple addition values for the call site may be obtained due to the encoding relative to different anchor nodes.

We revise the static analysis in Algorithm 1 to resolve the two challenges, as shown in Algorithm 2. It automatically picks anchor nodes. Initially only the `main` node is in the anchor node set *An* (Line 2). Whenever an integer overflow occurs while processing an edge $\langle p,n,l\rangle$, *p* is added into the set of anchor nodes *An* (Line 15) and the static analysis is rerun (Line 16).

In order to cutting calling contexts correctly, function IDENTIFYTERRITORIES first walks the "territory" of each anchor node. Specifically for each anchor it finds out the nodes and edges that can be reached through a bounded depth-first search, which starts the traversal from the anchor node and retreats at other anchor nodes; those anchor nodes form the boundary of the territory. From the nodes and edges in each territory we derive $nanchors[n]$ and $eanchors[e]$, which represent the anchor nodes that can reach node *n* and edge *e*, respectively.

The territories of anchor nodes overlap, which explains the reason behind the second challenge. In order to resolve it, the candidate addition values (*CAV*) and the inflated calling-context counts (*ICC*) of a node are extended to two-dimensional arrays with the first index still representing the node and the second an anchor node, taking into account of multiple anchor nodes that can reach the node. For example, $CAV[n][r]$ denotes the candidate addition value that should be considered when processing the next incoming edge *e* of *n* if and and only if $r \in eanchors[e]$.

The addition value for a given call site is determined by the max-

Algorithm 2 Encoding resolving encoding space explosion.

```
 1: function ENCODING(N, E)
 2:     An ← {main}
 3:     again: IdentifyTerritories(N, E, An)
 4:     for n ∈ N do
 5:         for r ∈ nanchors[n] do
 6:             CAV[n][r] ← 0
 7:     for n ∈ N in topological order do
 8:         for e = ⟨p, n, l⟩ of the incoming edges of n do
 9:             cs = ⟨p, l⟩
10:             if cs ∈ processedSites then
11:                 continue
12:             processedSites ← processedSites ∪ {cs}
13:             AV[cs] ← CalculateIncrement(cs)
14:             if AV[cs] =-1 then // Overflow detected.
15:                 An ← An ∪ {p}
16:                 goto again // Restart the encoding.
17:         if n ∉ An then
18:             for r ∈ nanchors[n] do
19:                 ICC[n][r] ← CAV[n][r]
20:         else
21:             ICC[n][n] ← 1
22: function IDENTIFYTERRITORIES(N, E, An)
23:     for r ∈ An do
24:         ⟨visitedN, visitedE⟩ ← BoundedDFS(r)
25:         for n ∈ visitedN do
26:             nanchors[n] ← nanchors[n] ∪ {r}
27:         for e ∈ visitedE do
28:             eanchors[e] ← eanchors[e] ∪ {r}
29: function CALCULATEINCREMENT(cs = ⟨p, l⟩)
30:     a ← 0
31:     for each e = ⟨p, n, l⟩ dispatched from cs do
32:         Assume eanchors[e] = {r_1, ..., r_k}
33:         a′ ← max{CAV[n][r_1], ..., CAV[n][r_k]}
34:         if a′ > a then
35:             a ← a′
36:     for each e = ⟨p, n, l⟩ dispatched from cs do
37:         for r ∈ eanchors[e] do
38:             CAV[n][r] ← ICC[p][r] + a
39:             if CAV[n][r] incurs an integer overflow then
40:                 return -1
41:     return a
```

imum of the candidate addition values of all its possible dispatch target nodes to address the dynamic dispatch (Line 31–35). All the anchor nodes that can reach each of the possible dispatch edges are taken into account, so that the territory conflict is resolved.

After a node's incoming edges are processed (Line 8–16), the ICC values are updated. For a non-anchor node *n*, the ICC relative to each anchor in $nanchors[n]$ is updated (Lines 18–19), while the ICC of each anchor node is set to 1 (Line 21), which is equivalent with $ICC[\texttt{main}] \leftarrow 1$ in Algorithm 1. The encoding algorithm is correct because the invariant described in Section 3.1 is satisfied. Specifically, the encoded ID obtained at *n* is in the range of $[0, ICC[n][top])$, where *top* is the top anchor node saved on the stack. It is notable that the runtime overflow checks are not needed, as the algorithm ensures that there is no integer overflow for ICC values (Line 19), which are the upper bounds of the encoding spaces.

Figure 5 shows an example of encoding involving two anchor nodes `C` and `D`. Consider the encoding of the calling context `CFG`. Edge `CF` is first processed. Edges `CF` and `CG` are due to the same

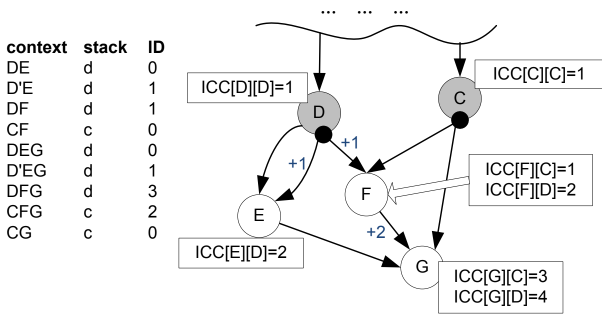

Figure 5: Example for encoding in large programs. C and D are anchor nodes. The superscript on D (D′) disambiguates two call sites in D both invoking E. D′E and DF are due to a virtual function call in D, and CF and CG are due to a virtual function call in C. The annotation of a node denotes the ICC values of the node relative to anchor nodes; for example, $ICC[E][D] = 2$ means the *ICC* of E relative to anchor D is 2. A stack is used to store the anchor node identifiers and the ID values when invoking them. The encoding of a call path CFG, for example, is represented by *c* on the stack, where *c* contains anchor C's identifier and the encoding ID value when C is invoked, along with the encoding ID value 2, which is the sum of addition values of CF and FG.

virtual call site. As $CAV[F][C]$ and $CAV[G][C]$ are initially both 0 (Line 4–6), the addition value for the call site is thus calculated as $max\{CAV[F][C], CAV[G][C]\} = 0$ (Line 33). Then $CAV[F][C]$ and $CAV[G][C]$ are updated as $ICC[C][C] + a = 1$ (Line 38). The addition value for the virtual function call in D are calculated similarly. After DE, D′E and DF are processed, now focus on node G. CG is already processed and the related call site is included in *processedSites* (Line 10–11) due to the dynamic dispatch at C. EG is processed next with addition value 0. $CAV[G][D] = ICC[E][D] + 0 = 2$ (Line 38), and $CAV[G][C]$ is still 1, so $max\{CAV[G][D], CAV[G][C]\} = 2$ is used as the addition value for FG (Line 33).

For decoding, we first recover the deepest piece of the calling context according to the current ID and the anchor node on the stack top. Then pop the anchor node and the ID to continue decoding the next piece of the calling context. The process repeats until the stack is empty. Given the ID 2 obtained at G and the anchor node C on stack stop, we recover a call path CFG. Then anchor node C and the saved ID are popped from the stack to continue decoding the next piece of the calling context.

4. PRACTICAL ISSUES

4.1 Dynamic Class Loading

So far we assume a complete call graph for static analysis. However, in reality the dynamic characteristics of a program can render this assumption invalid. Dynamic class loading, which is common in Java for example, makes the generation of a complete call graph ahead of runtime unresolvable.

As a result, relative to a call graph generated for static analysis, dynamically loaded classes introduce *unexpected call paths* (UCPs). Figure 6 shows such an example, where node X and its incoming and outgoing edges are missing during static analysis stage due to dynamic class loading. The context ABXE contains a UCP B $\rightarrow$ X $\rightarrow$ E; its encoding ID value is 0. However, if we decode it, an incorrect calling context ACE is obtained. This kind of UCPs is *hazardous* as they lead to incorrect encoding and decoding. Consider another context ABXD, which contains a UCP B $\rightarrow$ X $\rightarrow$ D, and BD

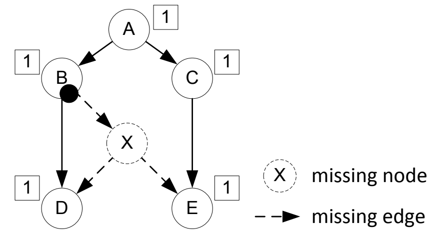

Figure 6: Incomplete call graph. BD and BX are due to the same virtual function call, where X is from a dynamically loaded class unexpected by static analysis.

and BX are due to the same virtual call site. Its encoding ID 0 can be decoded to ABD. Although the decoded result does not contain the dynamically loaded node X, it includes all the other nodes in the right order. We thus consider the encoding correct and this kind of UCPs *benign*.

In order to keep the encoding correct in the presence of dynamic class loading, we need to detect hazardous UCPs and respond accordingly. Inspired by the control flow integrity (CFI) technique [7], we propose a call path tracking technique that checks call transfers, and apply it to detect hazardous UCPs.

The technique consists of static analysis and runtime enforcement. In the beginning of the static analysis each node in the call graph is in a separate set. The analysis then traverses the call graph. For each call site, it finds out the dispatch target nodes, and merge the sets that contain those nodes. In the end each of the sets left is assigned with a unique set identifier (SID), and nodes in the same set share the SID. In runtime before a call is issued, the *expected* callee node's SID along with the call site and the current encoding ID value is saved. At the entry point of each statically loaded function, the expected SID is compared against the function's SID. If they are not equal, a hazardous UCP is detected. Consider the UCP B $\rightarrow$ X $\rightarrow$ E for example. E is able to detect it as hazardous since the expected SID set by B is not equal to E's SID.

Once a hazardous UCP is detected, the encoding responds as follows. First, the expected SID, the call site, the encoding ID, and the current *function's identifier* (E, in this case) are pushed onto stack. The encoding ID is reset to zero, and the encoding continues. At the exit point of E, the pushed information is popped to balance the stack and recover the encoding variables. It is easy to see that the technique allows benign UCPs; for example, in the case of B $\rightarrow$ X $\rightarrow$ D, the expected SID set by B is equal to D's SID.

During decoding, whenever the ID is 0 and the current function's identifier is equal to the one on the stack top, it pops the information from the stack to continue the decoding. Recall that the information pushed on the stack can be due to invocation to an anchor node, a hazardous UCP or a recursion. An extra integer can be used to indicate the information type in each stack element.[2]

Alternative solutions exist. For example, we can maintain a variable representing the depth of invocations of dynamically loaded functions by incrementing and decrementing the variable at each dynamically loaded function's entry and exit points, respectively. Such that each statically loaded function detects a UCP if the depth is not zero. The reset and recover of the depth variable using a stack are required when statically and dynamically loaded function calls interleave. The main engineering advantage of the call path track-

[2]Our implementation borrows two bits from the method identifier integer to identify the type of a push, so it does not need an extra integer.

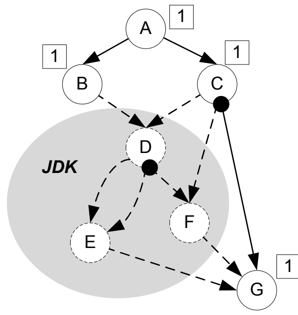

Figure 7: JDK library classes are excluded from encoding.

ing solution is that instrumentation of dynamically loaded classes is completely avoided, while in practice it is sometimes difficult or infeasible to instrument them. For example, if they are loaded by custom class loaders, the instrumentation component usually needs to modify the custom loaders; and the security policy of some third-party components does not allow instrumentation.

4.2 Flexible Encoding

It is desirable that we can perform a selective encoding for the components of interest in order to reduce encoding overhead. For example, the JVM and library methods are usually of less interest compared to application ones, as JVM implementation and Java libraries are often considered "black boxes" [13]. When recovering calling contexts, users may want to obtain all application methods, while JVM and library methods can be ignored. PCC, for example, redefines that a calling context consists of application functions only and hence encodes application functions solely [14, 13].

Leveraging the call path tracking technique we can skip encoding components of no interest the same way we handle dynamically loaded classes to achieve a more efficient and flexible encoding. As illustrated in Figure 7, JDK methods and the associated edges (denoted by dashed circles and lines) are of no interest and excluded from the call graph when running Algorithm 2, and the encoding is performed only on application methods. During runtime we rely on the call path tracking to detect unexpected UCPs and encode correctly.

Consider the calling context `ABDFG` for example. only edge `AB` is encoded, while edges `BD`, `DF` and `FG` are skipped and hence no overhead is incurred. `G` detects the hazardous UCP at its entry point, so it responds as discussed in Section 4.1 and continues the encoding. Finally, `ABG`, which consists of application methods only, can be recovered from the encoding result.

It illustrates another advantage of the call path tracking solution over the depth tracking one: no encoding or UCP detection code is executed inside the excluded components, such that the more components are excluded from encoding, the less overhead is incurred by encoding.

5. IMPLEMENTATION

The implementation of DeltaPath consists of static analysis and a runtime component. The input of the static analysis is the bytecode of the target program. Besides, the user can optionally specify classes of interest for selective encoding. We use WALA [5] to generate the call graph based on a context insensitive control flow analysis, 0-CFA [40]. Algorithm 2 is then run to compute addition values.

The runtime component is implemented as a Java agent [2]. During runtime it hooks the loading of each class and instruments the call sites based on the static analysis results using Javassist [17].

Our implementation does not require the source code. It is not dependent on a specific Java Virtual Machine (JVM); it can work with any JVM compatible with JDK 5.0.

6. EVALUATION

In this section we present the effectiveness and efficiency our encoding technique. We use the SPECjvm2008 benchmark suite [4], which contains a variety of Java programs including compilers (`compiler.*`), cryptography applications (`crypto.*`), scientific computation (`scimark.*`), and xml applications (`xml.*`). All experiments were performed on a machine with an Intel Core i7 CPU and 8GB RAM. We used Ubuntu 10.04 and Sun JDK 1.6.0.24 on this machine.

6.1 Static Program Characteristics

In SPECjvm a common dispatcher is used in all benchmarks to configure the workload and collect results. It is excluded from the encoding so that the statistics properly represent the characteristics of individual benchmark programs.

The static analysis is performed in two encoding settings. One is to encode functions of both JDK and application classes (*encoding-all*) and the other encodes applications functions only (*encoding-application*). The encoding-application setting corresponds to the scenario that the user is only interested in application functions in a calling context. Based on the call path tracking technique, DeltaPath provides the flexibility of encoding only the components of interest.

Table 1 presents the static characteristics of the benchmark programs with the two settings: encoding-all and encoding-application. For each program, the table details the program size in bytes (*size*), and for each encoding setting, the number of nodes (*nodes*) and edges (*edges*) in the call graph, the number of call sites to be instrumented (*CS*), the number of virtual function call sites (*VCS*), and the static maximum encoding ID value (*max. ID*) which represents the encoding space needed.

With the encoding-all setting, the majority of the benchmarks (13 out of 15) need an encoding space larger than a million. Two benchmarks (`sunflow` and `xml.validation`) in particular require huge encoding spaces with numbers shown in bold, such that even a 64-bit integer is insufficient for encoding. Algorithm 2 resolves the integer overflow problem automatically by adding 6 and 7 anchor nodes for `sunflow` and `xml.validation`, respectively.

With the encoding-application setting, all programs need much smaller encoding spaces. `xml.transform` benchmark needs a 64-bit integer for encoding, while the encoding space of each other benchmarks can fit into a 32-bit integer. In addition we observe that with the encoding-application setting the call graph and the number of instrumented call sites of each program are much smaller than with the encoding-all setting, which implies efficiency benefit due to the encoding flexibility.

6.2 Performance Comparison

We compare our work with the state of the art encoding technique, Probabilistic Calling Context (PCC) encoding [14], which is a purely runtime mechanism representing each calling context as an integer hash value. Similar to DeltaPath, PCC works with object-oriented programs. The original PCC was implemented as a module inside Jikes RVM, which is a research JVM allowing internal optimization options. For example, Jikes RVM' inlining information enables PCC to optimize the instrumentation by combining multiple encoding computations within inlined functions into a single one. Its performance overhead is very low on Jikes RVM, 3%

program	size (bytes)	encoding-all					encoding application				
		nodes	edges	CS	VCS	max. ID	nodes	edges	CS	VCS	max. ID
compiler.compiler	114K	2308	7329	7003	2839	7.8e7	112	77	93	31	12
compiler.sunflow	85K	1846	4185	5511	2490	9.6e7	117	83	104	43	12
compress	59K	1298	2675	3391	1394	4e5	98	65	93	57	32
crypto.aes	133K	2656	8201	8369	3487	2.5e9	99	69	91	40	25
crypto.rsa	133K	2656	8204	8386	3500	3.6e8	99	76	96	41	16
crypto.signverify	135K	2694	8290	8548	3576	2.5e9	96	68	108	47	37
mpegaudio	261K	3132	9734	9579	4116	3.3e14	252	284	497	317	130
scimark.fft.large	57K	1279	2636	3321	1347	4e5	78	37	41	19	5
scimark.lu.large	57K	1273	2616	3304	1331	2.2e6	76	34	40	10	4
scimark.monte_carlo	56K	1260	2590	3262	1311	1.4e6	62	22	24	10	4
scimark.sor.large	57K	1269	2614	3303	1339	1.4e6	73	28	32	10	4
scimark.sparse.large	57K	1265	2605	3291	1330	2.2e6	69	26	31	9	4
sunflow	458K	7727	25485	27135	13348	**4.4e21**	1069	2093	2995	1485	1.2e6
xml.transform	752K	9766	38010	44266	24969	1.2e17	1908	4389	6035	2162	1.2e10
xml.validation	478K	6703	23092	28333	15493	**4.6e19**	102	75	97	38	17

Table 1: Static program characteristics (SPECjvm2008). The maximum encoding ID values of `sunflow` and `xml.validation`, shown in bold, are larger than the maximum value of a 64-bit integer (around 1.8e19), so the two benchmarks need anchor nodes.

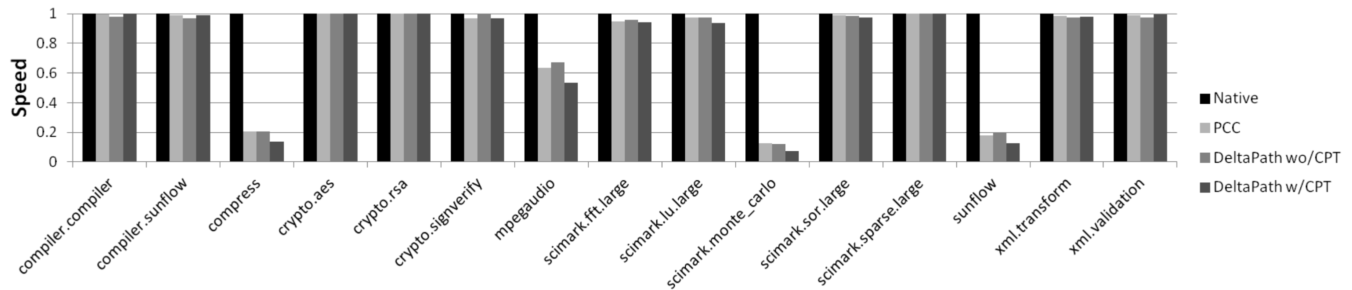

Figure 8: Execution speeds applying PCC, DeltaPath without call path tracking (wo/CPT), and DeltaPath with call path tracking (w/CPT), respectively. The speed means the throughput (operations per minute) that reflects the rate at which the system was able to complete invocations of the workload of that benchmark, and it is normalized against the native runtime.

on average and 9% at most.

In order to have a head-to-head comparison on our platform, we implemented PCC as a Java agent as well. Therefore, such low-level optimizations are not performed. Note that this is not a limitation of our algorithm but rather due to the implementation choice. If our scheme is implemented in Jikes RVM as the original PCC, both techniques can equally benefit from such low-level optimizations. In fact the composition of our encoding computations (simply adding up multiple additions into one) is as easy as in PCC, and the call path tracking for the inlined functions can also be done at compile time. The main focus of our experiments is to find out whether our technique is comparable with PCC in terms of efficiency and effectiveness using the same implementation technology.

As the original PCC work encodes application functions only, we adopt the encoding-application setting for DeltaPath to instrument the same set of functions. Figure 8 illustrates the normalized execution speed when our encoding technique and PCC are applied, respectively. We use the geometric mean to calculate the average slowdown. Overall DeltaPath (without call path tracking) incurs 32.51% slowdown on average with call path tracking introducing extra 6.79% slowdown, while PCC incurs 0.5% higher overhead than DeltaPath (without call path tracking). For the majority of the benchmarks (11 out of 15 benchmarks), DeltaPath with call path tracking incurs less than 6.5% overhead. In addition, both incur high overhead in a few benchmarks (`compress`, `mpegaudio`, `scimark.monte_carlo` and `sunflow`), as they contain a few small *hot* functions; the overhead can be largely reduced if the optimization of combining instrumentations is performed for inlined functions. The results show that DeltaPath is comparable with PCC in all benchmarks, which reflects the similarity of their runtime implementation: both instrument the same set of call sites with simple arithmetic operations.

At a low cost, *DeltaPath provides a reliable and precise decoding capability, which is the most critical difference between DeltaPath and PCC*. In contrast, Breadcrumbs [13], which is built on PCC with the purpose of providing decoding capability, either incurs almost 100% overhead for a "very accurate" decoding, or sacrifices reliability and accuracy significantly even at a moderate cost (extra 20% overhead over PCC).

6.3 Dynamic Program Characteristics

In order to evaluate the effectiveness of DeltaPath and PCC, we collect the encoded calling contexts at the entry of the instrumented application functions. Table 2 presents the statistics. It details the total number of calling contexts collected (*total contexts*), the maximum/average number of functions in a calling context (*max. depth* and *avg. depth*), the number of unique calling context encodings (*unique contexts*) collected by both techniques. For DeltaPath we present additional information about the encoding stack. The maximum/average depth of the the stack (*max. depth* and *avg. depth*), the maximum/average number of hazardous UCPs

program	collected calling contexts			PCC	DeltaPath					
	total contexts	max. depth	avg. depth	unique contexts	unique contexts	max. depth	avg. depth	max. UCP	avg. UCP	max. ID
compiler.compiler	92634	15	5.1	141	165	11	2.3	3	1.8	4
compiler.sunflow	63705	12	5.4	156	185	8	2.3	2	1.6	4
compress	3243640985	12	10.0	113	114	2	1.0	2	0.0	31
crypto.aes	14431	9	5.6	194	217	2	1.6	2	1.0	15
crypto.rsa	538625	9	6.0	156	179	2	2.0	2	1.0	9
crypto.signverify	541682	9	6.0	228	242	2	2.0	2	1.0	23
mpegaudio	2489700943	17	13.4	389	427	3	1.0	2	0.0	66
scimark.fft.large	566237360	12	10.0	65	101	3	1.0	2	0.0	4
scimark.lu.large	188838329	10	10.0	53	54	2	1.0	2	0.0	2
scimark.monte_carlo	5033167760	11	10.0	34	35	2	1.0	2	0.0	1
scimark.sor.large	293603875	10	10.0	48	67	3	1.0	2	0.0	2
scimark.sparse.large	252002429	11	10.0	46	47	2	1.0	2	0.0	2
sunflow	2840077292	39	21.8	196612	200452	26	4.4	3	1.0	842711
xml.transform	92333406	55	15.5	24422	24556	25	3.1	3	0.1	66412
xml.validation	12900727	11	9.0	127	141	2	2.0	2	1.0	5

Table 2: Dynamic program characteristics (SPECjvm2008).

detected per calling context (*max. UCP* and *avg. UCP*), and the maximum dynamic encoding ID (*max. ID*) are presented.

DeltaPath represents each calling context using a stack. Ideally, the stack only contains one element recording the entry node; when the calling context contains recursions, anchor nodes or hazardous UCPs, the depth increases. The maximum/average number of hazardous UCPs is not high, showing that hazardous UCPs are detected in all benchmarks although they are infrequent. The average stack depth is 1∼4.4 varying among the 15 benchmarks, compared to the original calling context depth (5.1∼21.8). PCC uses only one integer to represent a calling context, which is more concise than DeltaPath. However, PCC collects fewer unique calling context encodings due to hash collisions, implying that some calling contexts obtain identical encoding results. Therefore, for applications that do not need decoding and allow occasional encoding collisions, such as test coverage and profiling, PCC is a better choice; however, for applications that require precise encoding or decoding, DeltaPath is generally superior to PCC and Breadcrumbs.

7. RELATED WORK

Stack Walking. Stack walking is commonly used to obtain calling contexts in debugging [1] and error reporting [6, 3]. However, for applications that need continuously capture calling contexts, it usually incurs excessive overhead.

Dynamic Calling Context Tree. A dynamic calling context tree (CCT) [8, 46] is a summary of calling contexts represented as a tree data structure. Maintaining a complete CCT incurs large space and time overhead, while a CCT obtained using sampling may miss contexts of interest. In contrast, calling context encoding approaches [41, 42, 14, 13] including our work provide concise representation of all calling contexts.

Path Profiling. Ball and Larus developed an algorithm to encode intraprocedural control flow paths [10]. Melski and Reps extended the work by capturing both inter- and intraprocedural control flow [36]. It encodes the whole control flow transfer history leading to a program point, but their approach does not scale, because there exist too many possible paths for nontrivial programs, and the inserted code is complex. Calling context encoding targets a succinct representation of active methods on the call stack with high efficiency.

Precise Calling Context Encoding. Sumner et al. proposed a precise calling context encoding technique that allows decoding [41, 42]. However, it does not work well with object-oriented programming, large-scale software, and dynamic class loading. The challenges are resolved in our work.

Probabilistic Calling Context. Probabilistic calling context [14] by Bond et al. provides an encoding technique that works with OO programs. The encoding result is essentially a hash value of the identifiers of the functions in the calling context. Its advantages over DeltaPath are that the encoding result is consistently only one integer and it does not need static analysis. However, PCC does not provide decoding. Their later work addresses this shortcoming by combining call graph analysis and recording of encoding results [13]. Due to the hash based encoding nature, it remains as a probabilistic approach in essence thus leaving the decoding stage inaccurate, unreliable and/or expensive. In addition, its decoding has to be offline. In production systems where precise and timely response is needed, deterministic and instant decoding, which is supported by DeltaPath, is a highly desired property.

8. FUTURE WORK

Pruned and Relative Encoding. In applications such as event logging and profiling where the functions of interest are known and the user only needs to capture the calling contexts of those target functions, we can perform a pruned encoding by skipping encoding functions that do not invoke target functions directly or indirectly. For example, in Figure 4, assume the user's interest is the calling contexts of functions `D` and `F`; with a simple static analysis we can find `E` and `G` do not lead to the target functions, thus we can skip the encoding operations in `E` and `G`. This should boost the encoding efficiency for those applications. Moreover, we can exploit the relative positions of the target functions for encoding. For example, after the encoding result of `ABD` is stored, to encode `ABDF`, we simply represent the result as a reference to the previous encoding result and an encoding relative to `D`, that is, `DF`. It may reduce the storage space of encoding results.

Hybrid Encoding. PCC has a compact representation of the encoding result with a lack of precise decoding capability, while DeltaPath complements PCC. So a hybrid encoding approach combining PCC and DeltaPath can be interesting and useful. For example, we can perform profiling to establish the mapping between a set of calling contexts that are most frequently generated in a pro-

gram and their PCC encoding values, so that we can decode such a PCC value based on the mapping. The functions in those calling contexts form the *trunk* in the program's call graph, where we run PCC encoding. In the remaining part of the program we perform DeltaPath encoding with the functions in the trunk working as anchor nodes. Alternatively, we can substitute a calling context tree construction for the PCC encoding. Either hybrid encoding has the potential to shorten the encoding result and reduce the encoding space pressure without harming the decoding capability of DeltaPath.

Optimizations. In addition to the composition of instrumentation operations in inlined functions, there are other optimization opportunities. Our implementation accesses thread-local variables, which are used to store the current encoding result for each thread, via Java library APIs. If we build DeltaPath into a custom JVM, we can reserve a register to point to thread-local storage to speed up, which is adopted in the implementation of PCC in Jikes RVM. PCCE profiles the program and then picks *hot* edges as encoding free ones, that is, those with the addition value as zero. DeltaPath can also benefit from this strategy. Moreover, since the invocation target of a call to a `private`, `static` or `final` function is fixed, it is impossible that such a call invokes a method in a dynamically loaded class, so those calls do not need to be tracked to detect dynamically loaded classes; if the functions in statically loaded classes that interact with (i.e., invoke or are invoked by) functions in dynamically loaded classes are pre-known, we only need to enforce call path tracking in those functions.

9. CONCLUSION

We present DeltaPath, a precise calling context encoding technique that works with both procedural and object-oriented programs. It encodes virtual function calls correctly, and resolves the encoding space explosion problem in large-scale programs. Call path tracking is proposed to deal with dynamic class loading and is applied to achieve the flexibility of encoding only the program components of interest. Compared to probabilistic calling context encoding, DeltaPath has similarly high efficiency with the advantage of precise decoding.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive suggestions and comments. Many thanks to authors of PCCE [41], William N. Sumner and Xiangyu Zhang, for their valuable feedback on the paper.

10. REFERENCES

[1] gdb: The GNU Project Debugger. http://sources.redhat.com/gdb/.

[2] Java Virtual Machine Tool Interface (JVM TI). http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/.

[3] Linux Kernel Oops. https://www.kernel.org/doc/Documentation/oops-tracing.txt/.

[4] SPECjvm2008: Java Virtual Machine Benchmark. http://www.spec.org/jvm2008/.

[5] T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/.

[6] Windows Error Reporting (WER). http://msdn.microsoft.com/en-us/library/windows/desktop/bb513641(v=vs.85).aspx.

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[8] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1997.

[9] D. C. Arnold, D. H. Ahn, B. R. Supinski, G. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, 2007.

[10] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, 1996.

[11] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle. Finding similar failures using callstack similarity. In *Proceedings of the 3rd Conference on Tackling Computer Systems Problems with Machine Learning Techniques*, pages 1–1, 2008.

[12] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[13] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 2010.

[14] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 97–112, 2007.

[15] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 405–422, 2007.

[16] A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In *Proceedings of ACM & IEEE International Conference on Embedded Software*, pages 262–271, 2006.

[17] S. Chiba. Javassist - a reflection-based programming wizard for java. In *Proceedings of the ACM OOPSLA Workshop on Reflective Programming in C++ and Java*, 1998.

[18] T. M. Chilimbi and V. Ganapathy. Heapmd: Identifying heap-based bugs using anomaly detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–228, 2006.

[19] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44, 2009.

[20] J. Clause and A. Orso. Leakpoint: Pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 515–524, 2010.

[21] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 62–75, 2003.

[22] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-based methods for classifying software failures. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 451–462, 2004.

[23] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 103–116, 2009.

[24] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[25] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–111, 2007.

[26] H. Inoue. *Anomaly Detection in Dynamic Execution Environments*. PhD thesis, 2005.

[27] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, 2007.

[28] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.

[29] R. E. Jones and C. Ryder. A study of java object demographics. In *Proceedings of the 7th International Symposium on Memory Management*, pages 121–130, 2008.

[30] R. Joshi, M. D. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 239–250, 2004.

[31] S. Kim, T. Zimmermann, and N. Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, pages 486–493, 2011.

[32] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, pages 301–310, 2008.

[33] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 1999.

[34] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308 – 318, 2003.

[35] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *Proceedings of SIAM International Conference on Data Mining*, pages 286–297, 2005.

[36] D. Melski and T. W. Reps. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software*, pages 47–62, 1999.

[37] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[38] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, pages 289–304, 2005.

[39] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 2–2, 2005.

[40] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, 1988.

[41] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 525–534, 2010.

[42] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. *IEEE Transactions on Software Engineering*, 38(5):1160–1177, 2012.

[43] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–362, 2007.

[44] T. Zhang, X. Zhuang, S. Pande, and W. Lee. Anomalous path detection with hardware support. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 43–54, 2005.

[45] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 81–91, 2006.

[46] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–271, 2006.