Enhancing Program Analysis with Deterministic Distinguishable Calling Context

Sungkeun Kim
Samsung Research
Seoul, Republic of Korea
sk84.kim@samsung.com

Khanh Nguyen
Texas A&M University
College Station, USA
khanhtn@tamu.edu

Chia-Che Tsai
Texas A&M University
College Station, USA
chiache@tamu.edu

Jaewoo Lee
Texas A&M University
College Station, USA
jaewoo2@tamu.edu

Abdullah Muzahid
Texas A&M University
College Station, USA
abdullah.muzahid@tamu.edu

Eun Jung Kim
Texas A&M University
College Station, USA
ejkim@tamu.edu

Abstract

Calling context is crucial for improving the precision of program analyses in various use cases (clients), such as profiling, debugging, optimization, and security checking. Often the calling context is encoded using a numerical value. We have observed that many clients benefit not only from a *deterministic* but also *globally distinguishable* value across runs to simplify bookkeeping and guarantee complete uniqueness. However, existing work only guarantees determinism, not global distinguishability. Clients need to develop auxiliary helpers, which incurs considerable overhead to distinguish encoded values among all calling contexts.

In this paper, we propose Deterministic Distinguishable Calling Context Encoding (DCCE) that can enable both properties of calling context encoding *natively*. The key idea of DCCE is leveraging the static call graph and encoding each calling context as the running call path count. Thereby, a mapping is established statically and can be readily used by the clients. Our experiments with two client tools show that DCCE has a comparable overhead compared to two state-of-the-art encoding schemes, PCCE and PCC, and further avoids the expensive overheads of collision detection, up to 2.1× and 50%, for Splash-3 and SPEC CPU 2017, respectively.

CCS Concepts: **• Theory of computation → Program analysis**; **• Software and its engineering → Automated static analysis**.

Keywords: Calling context, Context-sensitive profiling and optimization

ACM Reference Format:
Sungkeun Kim, Khanh Nguyen, Chia-Che Tsai, Jaewoo Lee, Abdullah Muzahid, and Eun Jung Kim. 2025. Enhancing Program Analysis with Deterministic Distinguishable Calling Context. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC '25), March 1–2, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3708493.3712679

ACM ISBN 979-8-4007-1407-8/25/03
https://doi.org/10.1145/3708493.3712679

1 Introduction

Program analysis is paramount in helping developers grapple with large code bases and complex logic. To improve precision, an analysis is enhanced with calling context information [34]. A calling context represents the sequence of function calls starting from the main function to the current point of interest. The enhanced analysis thus has a finer-grained view of the application's behavior based on different execution paths. There is a large body of work on context-sensitive analyses for program optimization [18, 35, 41], debugging and testing [2, 11, 13, 42, 48, 49], security checking [16, 17, 23, 38], and others [9, 10, 14, 15, 21, 22, 27, 31, 52].

A calling context is commonly encoded using a numerical value, which we refer to as CCID (Calling Context IDentifier). Encoding saves memory compared to alternatives such as strings. Encoding also reduces the time cost of the analysis (e.g., two calling contexts can be easily checked by two different numerical values instead of performing a string comparison). However, encoding obfuscates the call chain, and thus requires a decoder to recover the list of call sites to make the analysis human-explainable. A naïve solution of recording a mapping between the numerical value and the concrete call chain in each program run incurs prohibitive space and time overheads.

The majority of existing works focus on encoding, i.e., generating a CCID [9, 12, 25, 44, 50, 51, 53]. We have observed that many critical client tools may benefit from CCIDs to be not only *deterministic* but also *globally distinguishable* across all functions, as shown in Table 1. The benefit of determinism is clear. Deterministic CCIDs allow the same calling context to be recognized without ambiguity across different runs of the analysis (i.e., inter-run). Distinguishable CCIDs are beneficial within a specific run (i.e., intra-run). CCIDs that are globally distinguishable help identify

Table 1. Use of calling contexts in various client tools.

	May Need Global Distinguishability?	May Need Determinism?
Fuzzing [10, 15]	Yes	Yes
Profiling [21, 22, 29, 52]	Yes	Yes
Malware Analysis [14]	Yes	Yes
Control-Flow Integrity [31]	No	No
Pointer Analysis [27]	Yes	No

Table 2. Summary of encoding schemes.

	Globally Distinguishable?	Deterministic?	Client Impact
PCCE [25, 44, 50, 53]	No	Yes	High
PCC [9]	Probabilistically	Yes	Low
Dynamic Call Path Profiler [12, 51]	Yes	No	High
DCCE (this work)	**Yes** (for non-DAG)	**Yes**	**Low**

unique calling contexts, thereby enabling fast lookup and eliminating the need for decoding (to disambiguate). For instance, fuzzing [10, 15] aims to enhance the exploration of new test cases without the complex constraint solving of symbolic execution [24]. A deterministic and globally distinguishable CCID helps these tools avoid unnecessarily revisiting program paths and hence, improving test coverage. Security-checking tools [14, 23, 31] often use a white-list-based approach for violation detection. Specifically, in profiling runs, fine-grained memory accesses (e.g., calling context and memory access patterns) are recorded, and are associated with deterministic and globally distinguishable CCIDs to be checked at production time. As another example, in profiling tools [21, 22, 29, 52], inflated profiled data collected from test inputs that trigger similar behaviors can be reduced with the help of deterministic and globally distinguishable CCIDs. Existing works such as PCCE [44] only provide determinism and local distinguishability. There is no work that can provide both natively. The clients must take extra steps to create a globally distinguishable CCID, including performing collision detection during runtime to detect the cases where the same CCID is reused for different calling contexts. These steps, however, add considerable overhead, ranging up to 23.6× compared to the native performance.

In this paper, we propose an encoding scheme ensuring determinism and globally distinguishable CCIDs, called *Deterministic/Distinguishable Calling Context Encoding* (DCCE), which in turn, enables a lightweight decoding algorithm. To our knowledge, this is the first work that identifies the requirement of deterministic and globally distinguishable calling context encoding and that achieves (almost) both properties with a static scheme. The key challenge of this work is that, in order to make the calling context encoding distinguishable, the call graph is mapped into a *non-overlapping numerical space*. We solve the challenge by, conceptually, iterating all calling contexts ordered by a pre-order traversal of the call graph. Each context is then given an incremental value, which ensures no two contexts have the same CCID. For call graphs with recursion, one CCID is used for an entire loop regardless of how many times the loop is executed. As such, our CCID is almost globally distinguishable. To address the determinism requirement, our solution is similar to existing works. Specifically, each call site on a call path is assigned statically a fixed edge weight. The CCIDs can be obtained by summing all of such edge weights. The calculation of the edge weight will be explained in §3.1. Because the weights are fixed, these CCIDs are guaranteed to be deterministic across different runs.

In summary, the contributions of this paper are as follows:

- We propose a calling context encoding algorithm whose encoded values are deterministic across runs and (almost) globally distinguishable across all possible calling contexts.
- We present a formal proof of the global distinguishability of the encoding scheme.
- We provide a thorough evaluation of the overhead of the encoding scheme and compare it to two state-of-the-art schemes, PCCE and PCC, and show significant performance improvement due to avoidance of collision detection at runtime.

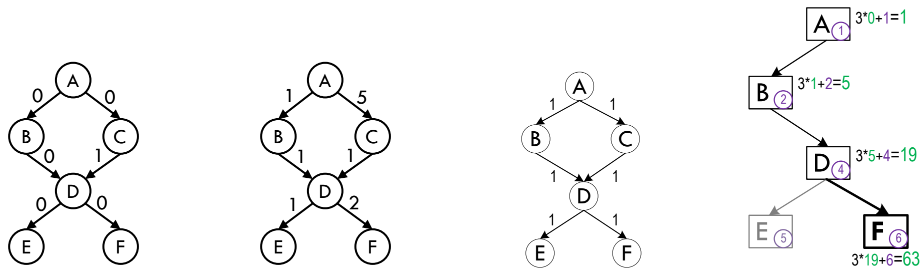
2 Existing Encoding Schemes

In this Section, we introduce state-of-the-art encoding schemes and discuss whether they can provide global distinguishability and determinism in Table 2.

2.1 Precise Calling Context Encoding (PCCE)

PCCE [25, 44, 50, 53] extends the Ball-Larus algorithm [6] that encodes control flows in a program to encoding calling contexts. Each callee n's context ID is the sum of caller p's context ID and a factor which is the position i of such caller p in the sequence of callers p_i in the static call graph. i is stored in the graph edge $p \rightarrow n$ as weight. The CCIDs encoded by PCCE are distinct among callees of the same caller (i.e., locally distinguishable).

Figure 1a shows an example of a weighted call graph. The path ABD has a context ID of 0 and the path ACD's context ID is 1. By construction, the encoded context by PCCE is not distinguishable among all paths. For instance, ABDE and ABDF have an ID of 0 and paths ACDE and ACDF 1. As such, PCCE will return to users all possible calling contexts when decoding a value. However, the encoding is deterministic thanks to the static weighted call graph.



(a) PCCE [44] **(b)** DCCE (this work) **(c)** DrCCTProf [51] **(d)** PCC [9] (on dynamic graph)

Call Path	Encoded Calling Context		
	PCCE	DCCE	PCC
A	0	0	1
AB	0	1	5
ABD	0	2	19
ABDE	0	3	62
ABDF	0	4	63
AC	0	5	6
ACD	1	6	22
ACDE	1	7	71
ACDF	1	8	72

(e) Comparisons of different consistent encodings.

Figure 1. Example of calling context encodings. PCCE and DCCE increment calling context value by edge weights on the call path. **DrCCTProf** increments by one for every new function call. PCC uses function $3 * V + call\_site\_ID$.

2.2 Probabilistic Calling Context (PCC)

PCC [9] comes close to achieve global distinguishability and determinism. It dynamically encodes the context at a call site using a non-commutative, composable linear function $3 * V + cs$ where V is the context ID at the caller and cs is a statically-assigned ID of the current call site. Figure 1d shows an example of encoding the path ABDF as $3 * 19 + 6 = 63$, with 19 being the context of the caller D and 6 the static ID of the call site where D calls F.

While deterministic, PCC value is not entirely distinguishable because the encoding function is not conflict-free. However, it has been shown that, with up to 10 million values, the expected number of collisions is negligible (less than 0.1% for 64-bit values). PCC leverages open-address hashing and double hashing to resolve collisions at run time. Due to the combination of the encoding function and the use of hashing, decoding PCC is more computationally demanding compared to other approaches [8].

2.3 Dynamic Call Path Profiling

Also known as stack shadowing, these schemes such as CCTLib [12] and DrCCTProf [51] monotonically assign a numeric value to new call paths during execution. This is equivalent to implicitly assigning each graph edge a weight of 1. For example, in Figure 1c, starting from A, the path AC can be assigned the ID of 1 if C is the first function to be called at run time. Otherwise, if C is last, the context's ID will be assigned a value 5. Because the numeric values and the calling contexts do not correlate, while the value is globally distinguishable, there is no determinism. As such, decoding is expensive as one has to store concrete call chains, eliminating the benefits of context encoding altogether.

3 Deterministic/Distinguishable Calling Context Encoding (DCCE)

In this Section, we describe Deterministic/Distinguishable Calling Context Encoding (DCCE). First, we introduce the algorithm that assigns weights to each edge in a call graph to guarantee globally distinguishable CCID calculation. Then we show proofs of the global distinguishability of DCCE, followed by the decoding algorithm.

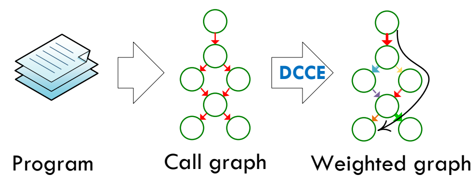


Figure 2. DCCE Overview. A call graph is constructed statically from a program. The encoding algorithm computes a weight to each edge. A call path to the bottom left node is shown in the weighted graph. A CCID is the sum of edge weights on the path.

Overview. Figure 2 shows an overview. Given a program, a call graph is constructed statically. The encoding algorithm runs on this static graph to compute a weight for each edge, shown in Figure 2 by different colors. Encoded context, i.e., CCID, is obtained by adding the weight of edges on the path starting from the root node. The key property of DCCE is to compute these weights in a way that CCID is guaranteed to be *globally distinguishable*. Specifically, CCID is the running path count in the call graph following a preorder traversal. Because edge weights are fixed, CCIDs are *deterministic*. Because DCCE relies on the static call graph, distinguishing different iteration of a loop is not feasible, our heuristic is to use one CCID for an entire loop, regardless of the number of iterations at run time. The details of this handling will be discussed in §3.4.

We start the discussion of the encoding scheme by formally defining terminologies.

Definition 3.1. A call graph (CG) is a directed multigraph with a pair of sets $(\mathbf{N}, \mathbf{E})$ where node $n \in \mathbf{N}$ represents a function in the program, and an edge $e \in \mathbf{E}$ exists between nodes p and n if p calls n.

Figure 3 shows an example of a call graph. Table 3 contains auxiliary functions to query the graph. Where convenient,

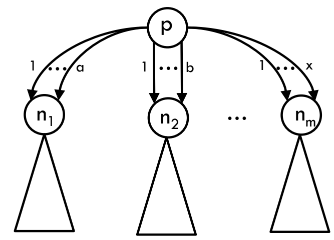


Figure 3. A call graph example. Triangles under nodes $n_1, n_2, \ldots, n_m$ are subgraphs rooted at such nodes. a, b, and x are numbers of edges from p to $n_1, n_2, \ldots, n_m$, respectively.

Table 3. Auxiliary functions used in the paper.

src(e) / *caller(e)*	source of an edge e, i.e., the caller p
dest(e) / *callee(e)*	destination of an edge e, i.e., the callee n
callees(p)	a total order set of all callees n_i called by p
edges(p)	a total order set of all edges e_i from caller node p
edges(p,n)	a subset of $edges(p)$, i.e., a total order set of all edges e_i from caller node p to callee node n
index(e)	position i of edge e in the set $edges(src(e), dst(e))$

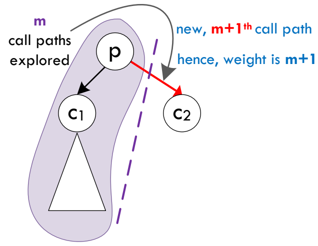


Figure 4. Intuition of assigning edge weight.

we refer to an edge e as a quadruple (p, n, l, w), with l is the l^{th} site (in program order) where p calls n, i.e., $l = index(e)$; and w is the weight of the edge e.

Definition 3.2. A call path from a node p to a node n, CP(p,n) is a chain of edges $e_1, e_2, \ldots, e_m$ through which n is reachable from p, i.e., src(e_1)=p, dest(e_m)=n, and dest(e_i) = src(e_{i+1}), $1 \leq i < m$.

Definition 3.3. A calling context of a node n, CC(n) is a call path from the root node r to n, i.e., CC(n) = CP(r, n).

3.1 Computing Edge Weights

In this work, an encoded calling context, **CCID**(n) is the sum of edge weights on a call path from the root node r to n, as done in other works [25, 44, 50, 53]. By statically assigning each call edge a weight, a CCID is deterministically (and cheaply) obtained by summing up all edge weights on a call path. The key challenge of DCCE is determining the edge weights so that CCIDs are *globally distinguishable*.

Figure 4 illustrates the intuition of assigning edge weights to guarantee the distinguishability of CCIDs. An edge e can be seen as a cut of a subgraph rooted at the caller, separating the visited subgraph (shaded) from the unvisited one. If there is no previously visited edge, e is assigned a weight of 1. If there are some call paths visited before, the weight of e must be larger than that number. This guarantees the CCID is incremental and reflects the running path count of the graph. In Figure 4, edge $p \longrightarrow c_2$ is a new path, after exploring the left subtree of p through c_1 with **m** paths. Correspondingly, the weight of edge $p \longrightarrow c_2$ is assigned **m+1**.

Formally, an edge e's weight, CCW(e) is defined as:

$$\begin{aligned}\mathbf{CCW}(e) &= \mathbf{CCW}(p, n, l, \_) \\ &= \sum_k \left\{\mathbf{NCP}(n'_k) \times \mathbf{NE}(p, n'_k)\right\}, \forall n'_k \in callees(p), n'_k < n, \\ &+ \left\{\mathbf{NCP}(n) \times (l-1)\right\} + 1\end{aligned}$$

where:

- **NE**(p, n) = $|edges(p, n)|$, i.e., the number of edges from caller p to callee n
- **NCP**(n) is the number of call paths from node n to all reachable nodes plus one. If n has no outgoing edge, **NCP**(n) is one.

The first term ($\sum$) of the definition represents all call paths visited through n's siblings. The next term accounts for cases where there are multiple edges from p to n, in which case, this term is the number of call paths visited through n itself. The (+1) ensures CCIDs are always increasing.

Algorithm 1 shows how to compute edge weight via computing *NCP*. Each node x maintains a variable *NCP* to store the total number of call paths starting from x. *NCP* is initialized with one and will not be updated if x does not have any outgoing edge. Given a call graph, we start a Depth-First Search (DFS) traversal from the root node. The recursive function `visit()` returns the updated *NCP* and that is accumulated to *NCP* of the caller p (Lines 11 and 10, respectively). If a callee n is visited, we do not have to visit n again because *NCP*(n) is already computed; we accumulate *NCP*(n) to that of p (Line 8). Before following any new edge, the weight w is assigned the current value of *NCP*(p) (Line 6).

Algorithm 1: Computation of *CCW* and *NCP*

```
1  def visit (p):
2      E′ ← edges(p)
3      NCP(p) ← 1
4      p.visited ← true
5      foreach e = (p, n, _, w) ∈ E′ do
6          w ← NCP(p)
7          if n.visited then
8              NCP(p) += NCP(n)
9          else
10             NCP(p) += visit(n)
11     return NCP(p)
```

Figure 5 shows an example of DCCE at a caller p, computing weight of outgoing edges to callees n_1, n_2, and n_3. With the first edge: e_1 ($p \longrightarrow n_1$), we assign the weight of one (the current value of $NCP(p)$) because there is no call path being explored yet. With edge e_2, because we already visited the subgraph rooted at n_1 through e_1, $NCP(n_1)$ is known, we assign $NCP(p) = NCP(n_1) + 1$ to e_2's weight (Figure 5a). Next, for edge e_3 ($p \longrightarrow n_2$), value of $2 \times NCP(n_1) + 1$ is assigned where $2 \times NCP(n_1)$ is the total number of call paths visited so far (Figure 5b). Similarly, as shown in Figure 5c, we assign e_4's weight $2 \times NCP(n_1) + NCP(n_2) + 1$.

Calculating CCID at run time. CCID can be calculated by add/subtract edge weight w to/from a global value CCV before and after each call site. This CCV is the encoded context at the caller. Algorithm 1 guarantees that the CCID assigned to a new path is always unique and at least one greater than the running CCV.

Figure 5d to Figure 5f show a concrete example of a call graph with CCID updated. 11 different CCIDs are assigned to distinguish 11 calling contexts.

3.2 Proof of Global Distinguishability of DCCE

Lemma 3.4. *$\forall u, v \in \mathbf{N}$ in CG ($\mathbf{N}$,$\mathbf{E}$), $u \neq v$, if $CCID(u)$ is assigned later than $CCID(v)$, then $CCID(u) > CCID(v)$.*

Proof. We prove Lemma 3.4 by exhaustion, based on two cases when visiting a node in the call graph:

Case 1: The first child of node p. Algorithm 1 will assign one as weight (Line 6) and thus, $CCID(n_1) = CCID(p) + 1$

Case 2: Another child n_2 of node p. After visiting the first child node n_1, $NCP(n_1)$ is computed, and accumulated into $NCP(p)$, which is assigned as weight of the edge $p \rightarrow n_2$. That is, $CCID(n_2) = CCID(p) + NCP(p) = CCID(p) + NCP(n_1) + 1$ with $NCP(p) = NCP(n_1) + 1$ being the weight computed by Algorithm 1. Hence, $CCID(n_2) > CCID(n_1)$.

Because the assigned weight is proportional to $NCP(p)$, which accumulates the number of call paths from p, it is guaranteed to be non-decreasing. Therefore, the CCID of any child n_j is larger than that of any child n_i visited earlier (cf. Figure 5a to Figure 5c).

We now prove that for all call paths from node n_1 to another node y in the subgraph rooted at n_1, because y is visited before n_2, $CCID(y) < CCID(n_2)$. Consider an edge in this subgraph $e = (n_1, x, l, w)$, its weight w (cf. §3.1) is:

$$
\begin{aligned}
CCW(e) &= \sum_k \left\{ NCP(n'_k) \times NE(n_1, n'_k) \right\} + NCP(x) \times (l-1) + 1 \\
&= \sum_k \left\{ NCP(n'_k) \times NE(n_1, n'_k) \right\} + NCP(x) \times l - NCP(x) + 1 \\
&= \boxed{\textstyle\sum_k \left\{ NCP(n'_k) \times NE(n_1, n'_k) \right\} + NCP(x) \times NE(n_1, x) + 1} \\
&\quad - NCP(x)
\end{aligned}
$$

The boxed terms are no more than $NCP(n_1)$, due to, by definition, $NCP(n_1)$ includes the number of paths from visited subtree(s), subtree at x, and unvisited subtree(s). Hence,

$$CCW(e) \leq NCP(n_1) - NCP(x)$$

That is, an edge e's weight is no more than $NCP(caller(e)) - NCP(callee(e))$. As such, for any call path of length m from n_1 to y:

$$
\begin{aligned}
\sum_{e_i \in CP(n_1,y)} CCW(e_i) \leq \sum_{e_i \in CP(n_1,y)} & NCP(caller(e_i)) - NCP(callee(e_i)) \\
&\leq NCP(caller(e_1)) - NCP(callee(e_m)) \\
&\leq NCP(n_1) - NCP(y) \\
&< NCP(n_1) \qquad \text{because } NCP(y) \geq 1
\end{aligned}
$$

Therefore, $\forall y$

$$
\begin{aligned}
&CCID(y) = CCID(n_1) + \sum_{e_i} CCW(e_i) && \text{how CCID is computed} \\
&CCID(y) < CCID(n_1) + NCP(n_1) && \sum_{e_i} CCW(e_i) < NCP(n_1) \\
&CCID(y) < CCID(p) + 1 + NCP(n_1) && \text{Case 1}: CCID(n_1) = \\
& && \quad CCID(p) + 1 \\
&CCID(y) < CCID(n_2) && \text{Case 2}: CCID(n_2) = \\
& && \quad CCID(p) + NCP(n_1) + 1
\end{aligned}
$$

Hence proved that $CCID(n_2)$ is larger than the CCID of any visited node. □

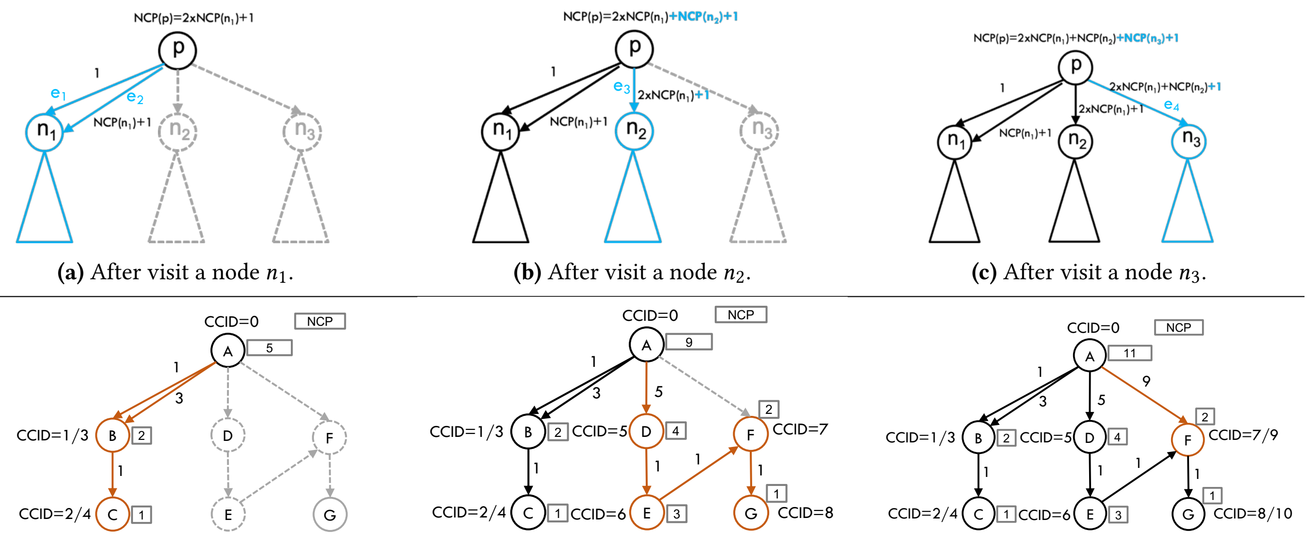
Theorem 3.5. *(Distinguishability) $\forall u, v \in \mathbf{N}$ in CG ($\mathbf{N}$,$\mathbf{E}$), if $u \neq v$, $CCID(u) \neq CCID(v)$.*

Proof. We prove this by contradiction. Let us assume that $CCID(u)$ and $CCID(v)$ are not distinguishable. That is, $\exists u, v \in \mathbf{N}$ where $CCID(u) = CCID(v)$ when $u \neq v$. However, if $u \neq v$, either $CCID(u)$ is assigned earlier than $CCID(v)$ or $CCID(v)$ is assigned earlier than $CCID(u)$. By Lemma 3.4, if $CCID(u)$ is assigned earlier than $CCID(v)$, then $CCID(u) < CCID(v)$ (and vice versa). In either case, $CCID(u) \neq CCID(v)$. Therefore, there is no such u and v, thus contradicting the assumption. As such, the Theorem is true. □

3.3 Decoding Algorithm

Unlike other approaches [9, 44], thanks to DCCE, decoding is lightweight and a greedy algorithm, as shown in Algorithm 2. From the root, it iterates through outgoing edges, sorted by weight. It adds callee's name (Line 9) to the result and decrements the queried CCID by the first edge whose weight w < CCID. Decoding recursively follows the chosen edge and keeps decrementing until the CCID equals zero.

It is worth noting that analyses with DCCE have no needs for this decoder. Because DCCE's CCIDs are globally distinguishable, analyses have the guarantee that any two calling contexts will result in two different CCID values. We provide



(a) After visit a node n_1. **(b)** After visit a node n_2. **(c)** After visit a node n_3.

(d) After visiting the left subtree. **(e)** After visiting the mid subtree. **(f)** After visiting the right subtree.

Figure 5. A walk-through example of DCCE. Top row: algorithm steps. Bottom row: concrete CCID values in corresponding steps. Dotted graphs are not visited yet. Callees reachable via multiple paths have CCIDs separated by "/".

Algorithm 2: Decoding Algorithm

```
def decode (CCID):
    p ← r                              ▷ r is root node
    context ← "main"
    while CCID ≠ 0 do
        E'' ← edges(p) // sorted by weight, largest first
        foreach e = (p, n, _, w) ∈ E'' do
            if CCID >= w then
                CCID −= w
                context ← context • FuncName(n)
                p ← n
                break
    return context
```

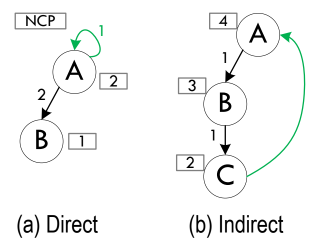
this decoding algorithm as a utility to developers to recover the call path of any CCID if needed.

Example. Consider Figure 5f with a CCID=5. From node A, there are 4 outgoing edges with weights 1, 3, 5, and 9, respectively. The heaviest edge (9) is clearly incorrect. Among the other edges, we select the edge with the weight of 5. Because of our encoding, every CCID is unique and is guaranteed to have one and only one path whose cumulative edge's weight equals 5; hence picking other edges (with weight 1 or 3) is not correct. The returned call path is AD.

3.4 Recursive Calls

Encoding recursive calls is challenging because the call graph is no longer a DAG (directed acyclic graph). We consider two

Figure 6. Weighted call graphs after applying Algorithm 1 with (a) direct and (b) indirect recursive call



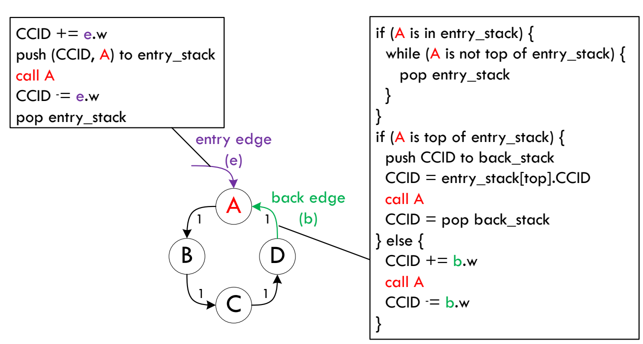


Figure 7. Instrumentation to entry edge e and back edge b for recursion. The current CCID is saved at the entry edge in `entry_stack`). At the back edge, the saved CCID is reused.

types of recursive calls: one is when a function has a self-loop edge (*direct recursion*), and the other is when a function calls another function that is in the current calling context, forming a group of cyclic edges (*indirect recursion*). In static analysis, we can identify those "back edges" that connect back to previously visited nodes. For PCCE, a recursive call is treated as the entry point of a separate sub-path, and the encoding is reset to zero after the previous encoding is pushed to the stack. The previous encoding is then popped and restored when returning from the recursive call. This is okay because PCCE only support local distinguishability and can still differentiate the calling contexts in a specific function even if the function is in a recursive sub-path.

DCCE takes a similar approach of pushing and popping the CCIDs when entering a recursive loop but maintains global distinguishability even between non-recursive call paths and recursive sub-paths. Consider the example in Figure 6a where A has a self-loop edge (direct recursive). According to Algorithm 1, edge $A \rightarrow A$ has a weight of 1, and $A \rightarrow B$ has a weight of 2. If we simply add the edge weight when a function is called, there is no bound to the CCID values since the edge $A \rightarrow A$ can be traversed for an infinite amount of times. This is a limitation for both PCCE and DCCE because static analysis cannot estimate the number of times a recursive call is taken. For indirect recursive, we consider the example in Figure 6b, where a back-calling edge $C \rightarrow A$ creates a loop between A, B, and C. According to Algorithm 1, the edge $A \rightarrow C$ has a weight of 1, so the calling context `ABCA` will have CCID=3 and will further increase if the edge $A \rightarrow C$ is traversed repeatedly.

To resolve this issue, we identify two types of special edges within a call graph. A *back edge* is an edge that leads to a previously visited node. Correspondingly, an *entry edge* is an incoming edge that shares the same destination as a back edge. We identify these back edges and entry edges by searching cyclic sub-graphs within the generated call graph. When an entry edge is called, the current CCID is pushed into an `entry_stack` and is later popped when returning from the function. When a back edge is called, the current CCID is pushed into a `back_stack` and then we set the current CCID as the CCID at the top of the `entry_stack`. Figure 7 shows the instrumentation needed at entry and back edges.

3.5 CCID Overflow

Because DCCE enumerates all paths in the call graph, it is more prone to value overflow than techniques such as PCCE because PCCE leaves some edges with zero weight if there is only one call path to a function. To mitigate the huge size of the graph, there are complementary techniques to reduce the size of call graphs while not impacting the effectiveness of the encoding. For instance, strongly connected components can be collapsed into one node in the graph [47]. In addition, through dynamic profiling, we may detect the most popular call paths within a program. By sorting the call paths by likelihood of occurrence and assigning smaller edge weights to more popular call paths, we can minimize most of the CCIDs observed during run time and can actively avoid overflowing the CCID variables.

3.6 Indirect Calls

For indirect calls, we use precise Andersen-style interprocedural points-to analyses [3] to transform an indirect call into multiple direct calls with conditional branches when building the call graph. If an unexpected call target is observed for an indirect call, we provide the feature of dynamically updating the edge weights at the runtime. To do so, we maintain a table of maximum numbers of call paths for each function, and the back trace of each instrumentation location where the edge weight needs to be updated for adding a new edge to the function. Then, when a new call target is detected, we recursively trace back to previously visited functions and update all edge weights accordingly. We consider this a rare occasion because the Andersen analysis can determine most of the indirect call targets under normal circumstances.

3.7 Dynamically Linked Library (DLL)

Because DCCE relies on the static call graph, it cannot handle dynamically linked libraries or code compiled just-in-time (JIT). We defer to future work to adopt Li et al. [25] to perform dynamic encoding.

4 Implementation

We use LLVM (version 12.0.0) as our implementation platform. Call graphs are constructed using the SVF tool [43, 45] with optimized LLVM IR as input. The weighted graph is computed (cf. §3) offline. The result weighted graph is loaded by LLVM for static instrumentation. A thread-private variable is used to store the CCID so that no lock is needed for updating the CCID. For DCCE, the SVF tool instruments each call instruction by inserting add_weight: CCID += w before the call instructions and inserting remove_weight: CCID -= w after the call instructions. For PCCE, only the call instructions that have edge weight larger than zero are instrumented. For PCC, a randomly-generated weight w is assigned to each call instruction, without the consideration of the call graph. add_weight: CCID = CCID * 3 + w is added before a call instruction and remove_weight: CCID = (CCID - w) / 3 is added after a call instruction.

5 Evaluations

We run all experiments on a Dell OptiPlex 3060 Tower desktop machine with Intel® Core™ i7-8700 CPU @ 3.20 GHz, 32 GB memory, running Ubuntu 20.04.6 LTS with the Linux kernel 6.2.0. The hard drive is Seagate 2TB SATA 7.2K RPM.

We compare DCCE against PCCE [44] and PCC [9] using nine and thirteen benchmarks from SPEC CPU 2017 [39] and Splash-3 [36], respectively, written in C/C++. The excluded benchmarks (e.g., *perlbench, gcc, imagick*) are due to the CCID overflow, which occurs with PCCE as well. Benchmark binaries are built using LLVM with *–plugin-opt=-lto-embed-bitcode=optimized* and *-O3*. We use the provided inputs from SPEC CPU 2017 and Splash-3 and run each application ten times and report the median value. Table 4 summarizes compile-time and run-time statistics.

Table 4. Compile-time and run-time statistics of SPEC 2017 and Splash-3 benchmark suites. †: The percentage of edges with zero weight.

	Compile time statistics				Run time statistics
Benchmark	Nodes	Edges (†)	#Indirect Calls	Max CCID (DCCE)	#Call Paths visited
lbm	25	56 (0.00%)	0	74	10
leela	115	751 (2.40%)	0	4,925	7,055,467
mcf	25	137 (0.73%)	48	188	790
namd	102	1,952 (0.00%)	66	16,077	96
omnetpp	4,007	7,974 (3.92%)	2,977	4,484,946,004,759	8,446,032
parest	2,226	15,269 (0.29%)	6,729	65,117,646	992,272
x264s	398	21,321 (0.00%)	17,911	18,929,528	15,787
xalancbmk	5,444	4,902 (8.29%)	60	69,077,904,007	1,535,244
xz	151	1,900 (11.26%)	1,561	304,699,440,136	119
barnes	73	270 (0.74%)	0	495	248
cholesky	155	544 (0.74%)	0	4,835	4,205
fft	39	201 (0.00%)	0	421	31
fmm	115	562 (1.43%)	38	14,513	2,263
lu-cb	39	170 (0.00%)	0	241	38
lu-ncb	36	158 (0.00%)	0	228	36
ocean-cp	43	339 (0.00%)	0	732	106
ocean-ncp	35	303 (0.00%)	0	603	55
radiosity	205	619 (5.01%)	16	315,116	3,120
radix	30	170 (0.00%)	0	239	22
raytrace	143	659 (0.91%)	90	2,450	4,500
water-nsquared	42	239 (0.00%)	0	434	49
water-spatial	42	252 (0.00%)	0	437	55

Making PCCE and PCC Globally Distinguishable. To have a fair comparison, we extend the original PCCE scheme, which only generates CCIDs that are locally distinguishable. Instead, we encode the original PCCE encoded value with a call graph node ID unique to each function. Such a call graph node ID is known at LLVM transformation passes. However, because both the PCCE encoded value and call graph node ID are 64-bit values, we need to combine the two values into one 64-bit integer, using a hash function. To ensure global distinguishability, *collision detection* is performed at the runtime by inserting the values into an unordered, hashed set. We use the hash function from the C++ boost library to combine the original encoded value and node ID.

For PCC, the call_site_ID used in the encoding function is randomized for each call site. However, PCC suffers the same limitations as PCCE. PCC cannot guarantee global distinguishability unless collision detection is performed on the encoded values. We use the same implementation from PCCE to detect collision of PCC encoded values.

5.1 Instrumentation Overhead on Execution Time

We start by evaluating the instrumentation overhead on the execution time of SPEC CPU 2017 and Splash-3. In this experiment, we only add and remove the edge weight at each entry and exit of a function (for PCCE and DCCE — for PCC, we use the encoding function with V=1), but do not output the encoding results to any variable. This is to evaluate the base overhead of each encoding scheme without the impact of encoding. Figure 8 shows this instrumentation overhead. Among the three schemes, PCC incurs the highest overheads, up to 184% for `radiosity`, and 55.1% on average across all programs. Meanwhile, DCCE and PCCE have similar overheads (42.6% and 41.4% on average, respectively), and PCC incurs 9-10% time overheads than DCCE and PCCE.

Table 5. Analysis and instrumentation cost (shown as `minutes:seconds`) for DCCE, broken down into various components. We only show the cost for benchmarks with more than 1,000 call paths (see Table 4). Other benchmarks each take no more than 5 seconds in total.

Benchmark	CG Extract. & PTA	CC Analysis & Gen.	Bitcode Inst.	Binary Gen.	Total
leela	00:00.7	00:00.1	00:01.6	00:00.2	00:02.6
namd	00:04.4	00:00.1	00:25.6	00:01.0	00:31.1
omnetpp	05:55.7	00:28.9	12:30.4	00:02.3	**18:57.3**
parest	02:40.7	00:02.1	07:20.9	00:03.2	**10:06.9**
x264	00:23.5	00:00.9	00:52.6	00:00.8	01:17.8
xalancbmk	02:39.9	00:02.1	20:42.1	00:05.7	**23:29.8**
xz	00:01.3	00:03.9	00:03.5	00:00.2	00:08.9
cholesky	00:00.5	00:00.1	00:00.7	00:00.1	00:01.4
fmm	00:00.3	00:00.1	00:00.4	00:00.1	00:00.9
radiosity	00:00.5	00:00.1	00:00.7	00:00.1	00:01.4
raytrace	00:00.9	00:00.1	00:01.2	00:00.1	00:02.3

5.2 Analysis and Instrumentation Cost

We collect the analysis and instrumentation cost for DCCE using the `time` command to measure the wall time of each step of the static phase. We show the time cost in Table 5. For most applications in SPEC CPU 2017 and Splash-3, the entire analysis and instrumentation process is completed within 10 seconds. Only three applications (`parest`, `omnetpp`, and `xalancbmk`) take more than 10 minutes to complete the entire process, most of which is spent on (1) call graph (CG) extraction and point-to analysis (PTA) using SVF, and (2) instrumenting the LLVM bitcode to inject instructions for adding the edge weights. The aforementioned three workloads have the highest static-time cost due to having some of the largest numbers of edges in their call graphs. Other costs such as the cost for Calling Context Analysis for assigning the edge weight to each pair of caller and callee (cf. §3) is insignificant; e.g., `omnetpp` has the highest cost with 28.9 seconds to assign the edge weights for the entire call graph.

Not shown here, the analysis and instrumentation of PCCE incur a similar cost as DCCE, due to the same process of call graph extraction, point-to-analysis, and calling context analysis. For PCC, there is no time spent on such components because the assignment of edge weights does not rely on the call graph. The only static-time cost for PCC is the instrumentation of the LLVM bitcode.

5.3 Memory Overhead

DCCE incur negligible memory overhead with 1.7 MB (1%) and 1.8 MB (8%) for SPEC CPU 2017 and Splash-3, respectively, on average because it only allocates one CCID variable

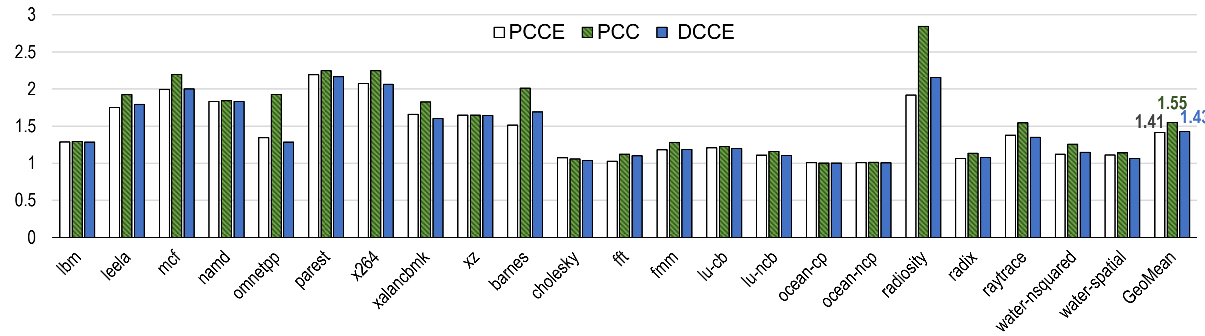


Figure 8. Execution time of SPEC CPU 2017 and Splash-3, with the instrumentation of adding and removing the edge weights encoded with PCCE, PCC, and DCCE, normalized to the native execution without any instrumentation. Lower is better.

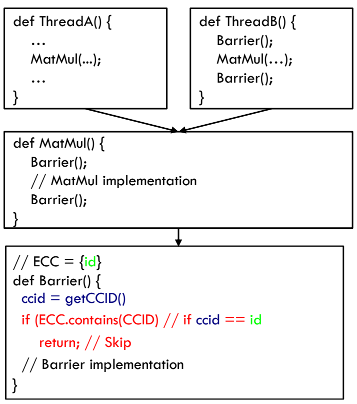


Figure 9. Barrier Elision example. id is a profiled CCID of the call path reaching *Barrier* from *ThreadB*

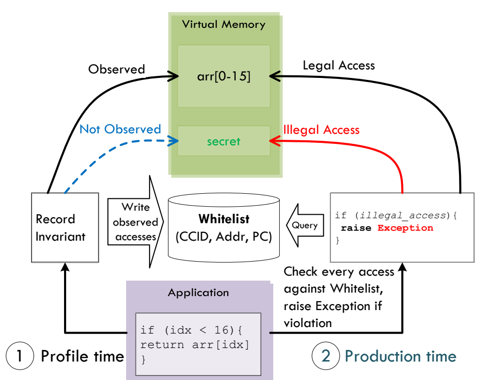


Figure 10. Overview of WHISTLE [23].

per thread. PCCE is comparable, consuming 3.86 MB (2%) and 2 MB (9%) more memory for SPEC CPU 2017 and Splash-3, respectively, on average.

6 Client Tools

We use 2 clients to evaluate the encoding schemes:

Barrier Elision [11] aims to eliminate unnecessary barrier operations in parallel programs. Figure 9 shows an example. In this example, *ThreadB* calls a function *Barrier* before and after a function *MatMul*, but it is redundant because *MatMul* itself will call *Barrier*. Redundant barriers are commonly due to developers' conservatism and can negatively impact the application's performance. At profile time, the tool performs data race detection, and collects a set of *Eligible Candidate Contexts (ECC)* which contains data-race-free contexts. The ECC is queried in production runs, during which the barrier operation is skipped if the current CCID at the barrier call is in ECC. Distinguishable CCIDs help the tool confidently use the encoded values in ECC without decoding to disambiguate 2 equal CCID values.

WHISTLE [23] provides protection against software and hardware memory safety violations. Figure 10 shows an overview. WHISTLE first profiles programs in attack-free runs. It whitelists context-sensitive memory access: which instruction (PC) accesses which data (Addr) and under which context (CCID). These accesses are considered legitimate. After profiling, WHISTLE starts monitoring all memory access, checking for violation: if a tuple (CCID, Addr, PC) is not in the whitelist, such access is illegal and an exception will be raised. The benefit of CCIDs that are globally distinguishable is similar to that in case of Barrier Elision.

6.1 Impact on Barrier Elision

Table 6 shows the normalized execution time of Barrier Elision with each application in SPEC CPU 2017 and Splash-3, and the geometric mean (geomean). Overall, DCCE does not incur more overheads. Because Barrier Elision only requires reading CCIDs at thread creation (`pthread_create`), barrier (`pthread_barrier_wait`), and use of conditional variables (`pthread_cond_wait` and `pthread_cond_broadcast`), the overheads of extraction and collision detection of CCIDs are marginal. For SPEC CPU 2017, both PCCE and PCC are sightly slower than DCCE slower (3% and 6%, respectively). For Splash-3, PCCE is a bit faster (1%) while PCC is marginally worse (2%) than DCCE.

As an interesting data point, DrCCTProf [51] incurs significant overheads, up to 17×, highlighting the expensiveness of the stack shadowing approach. Such a high cost is due to dynamically computing and assigning CCIDs to the current

Table 6. Execution time of SPEC CPU 2017 (top) and Splash-3 (bottom) with BarrierElision, encoded with PCCE, PCC, and DCCE, normalized to native execution without any instrumentation. Smaller is better. For PCCE and PCC, we separately show the overhead of collision detection (+CD).

Benchmark	PCCE+CD	PCC+CD	DCCE
lbm	1.252+0.003	1.253+0.012	1.261
leela	1.722+0.000	1.785+0.000	1.772
mcf	2.009+0.000	2.023+0.003	1.999
namd	1.824+0.000	1.829+0.006	1.818
omnetpp	1.482+0.000	1.900+0.000	1.395
parest	2.227+0.000	2.254+0.000	2.195
x264	1.964+0.009	1.985+0.023	1.967
xalancbmk	1.690+0.017	1.715+0.000	1.660
xz	1.623+0.000	1.626+0.000	1.638
geomean	**1.732+0.002**	**1.798+0.003**	**1.722**
barnes	1.391+0.018	1.626+0.008	1.562
cholesky	1.000+0.143	1.036+0.007	1.000
fft	1.019+0.000	1.093+0.000	1.089
fmm	1.000+0.001	1.016+0.000	1.060
lu-cb	1.185+0.019	1.191+0.000	1.181
lu-ncb	1.121+0.000	1.148+0.000	1.122
ocean-cp	1.003+0.002	1.001+0.001	1.001
ocean-ncp	1.002+0.000	1.003+0.004	1.001
radiosity	1.643+0.000	1.908+0.013	1.831
radix	1.045+0.000	1.065+0.001	1.066
raytrace	1.359+0.000	1.369+0.000	1.352
water-nsquared	1.150+0.000	1.184+0.000	1.172
water-spatial	1.095+0.006	1.084+0.001	1.087
geomean	**1.141+0.014**	**1.186+0.006**	**1.174**

Table 7. Execution time of SPEC CPU 2017 (top) and Splash-3 (bottom) with WHISTLE, encoded with PCCE, PCC, and DCCE, normalized to native execution without any instrumentation. Smaller is better. For PCCE and PCC, we separately show the overhead of collision detection (+CD).

Benchmark	PCCE+CD	PCC+CD	DCCE
lbm	1.258+0.000	1.254+0.000	1.253
leela	1.885+0.598	1.871+0.499	1.875
mcf	2.283+1.495	2.201+1.190	2.229
namd	1.837+0.067	1.835+0.043	1.836
omnetpp	1.782+1.142	2.121+0.974	1.644
parest	2.263+0.142	2.287+0.137	2.218
x264	2.168+1.021	2.128+0.530	2.122
xalancbmk	1.873+0.889	1.888+0.647	1.799
xz	1.633+0.041	1.629+0.043	1.632
geomean	**1.860+0.503**	**1.885+0.379**	**1.819**
barnes	1.726+23.642	1.836+23.151	1.800
cholesky	1.036+0.035	1.018+0.053	1.054
fft	1.046+0.572	1.102+0.226	1.110
fmm	1.012+0.585	1.119+0.576	1.018
lu-cb	1.251+3.532	1.197+2.684	1.190
lu-ncb	1.170+2.790	1.139+2.394	1.123
ocean-cp	1.000+0.005	1.000+0.000	1.002
ocean-ncp	1.001+0.000	1.000+0.000	1.002
radiosity	2.352+42.459	2.374+37.666	2.356
radix	1.125+2.314	1.109+2.370	1.113
raytrace	1.546+6.637	1.534+6.178	1.528
water-nsquared	1.256+2.058	1.256+1.992	1.231
water-spatial	1.110+0.156	1.130+0.182	1.103
geomean	**1.236+2.127**	**1.249+1.957**	**1.235**

context and maintaining a calling context tree. Furthermore, the analysis with DCCE did not generate incorrect results.

6.2 Impact on WHISTLE

Table 7 shows the normalized execution time of WHISTLE with each application in SPEC CPU 2017 and Splash-3, and the geometric mean (geomean). For WHISTLE, we inject the code for reading CCIDs into every function to check memory access against the respective calling contexts recorded during profiling. If we compare the base overhead of PCCE and PCC, without the consideration of collision detection, DCCE is on par with PCCE while PCC is slightly worse. However, if we consider the cost of collision detection in order to make PCCE and PCC globally distinguishable, the overhead becomes quite significant. For SPEC CPU 2017, PCCE is 50% slower than DCCE, and PCC is 45% slower than DCCE. For Splash-3, both PCCE and PCC at least incur 2× more overheads than DCCE. Especially for applications which perform a large amount of function calls, such as `barnes` and `radiosity` in Splash-3, the overheads can be up to 42×.

7 Related Work

Stack walking. The straightforward method to provide calling context is by walking the call stack, unwinding function frames using either compiler-recorded information (e.g., libunwind [26]) or results from binary analysis (e.g., Valgrind [33], HPCToolkit [1]). This technique does not require instrumentation at each call site. Hence, it adds no execution overhead except when calling context is needed (e.g., a bug or crash at run time [37]). Walking the stack is time-consuming and thus is only suitable for clients who rarely need to know the context such as a bug-reporting system, unlike any context-sensitive analyses which frequently require calling context information.

Stack shadowing. An alternative to stack walking is by constructing a calling context tree (CCT). The tree represents the dynamic call path and is suitable for fine-grained call path profilers such as CCTLib [12], DrCCTProf [51], DeadSpy [13], Runtime Value Numbering [49], RedSpy [48], and LoadSpy [42]. Combining call stack unwinding and stack shadowing yields hybrid call-path collection [19, 28]. As mentioned earlier in §2, dynamic path profilers implicitly assume an edge weight of one. DCCE can be adapted by these works to make the calling context deterministic and distinguishable. The weights computed by DCCE can be packaged as a library that can be queried at run time. However, this approach suffer from huge execution time overheads [12, 51].

Calling context profiling. There is a large body of work relying on calling contexts profiling for various software engineering tasks [5, 7, 32, 40, 46, 50, 51, 53, 54]. Ausiello et. al. introduce k-calling context forest and provide performance metrics for paths of length at most k [5]. Zhuang et. al. present an adaptive bursting technique to build accurate calling context trees at run time [54]. Valence leverages the compiler to use shorter encodings for hot paths and longer encodings for infrequent paths to reduce the encoding size [53]. DCCE complements these works by cheaply encoding a unique identifier for each calling context.

Path profiling. Paths and calling contexts are orthogonal: paths provide intraprocedural control flow while calling contexts provide interprocedural control flow. Paths are much cheaper than calling context to compute. There is a line of work in path profiling [4, 6, 19, 20, 32, 46]. DCCE complements works in path profiling and can be combined in a hybrid approach similar to Melski and Reps [30].

8 Conclusion

This paper presents DCCE, a new calling context encoding scheme. Our evaluation with two real clients shows that DCCE incurs negligible additional space overhead and reduces time overhead compared to other state-of-the-art encoding schemes (up to 2.1× and 50%, for Splash-3 and SPEC CPU 2017, respectively) while providing the benefits of *determinism* and *global distinguishability* of the encoded values.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback in improving this paper. This work is supported by NSF CNS-2107010.

References

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*. 298–313.

[3] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph. D. Dissertation. DIKU, University of Copenhagen.

[4] Taweesup Apiwattanapong and Mary Jean Harrold. 2002. Selective path profiling. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Charleston, South Carolina, USA) *(PASTE '02)*. 35–42. https://doi.org/10.1145/586094.586104

[5] Giorgio Ausiello, Camil Demetrescu, Irene Finocchi, and Donatella Firmani. 2012. k-Calling context profiling. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. 867–878. https://doi.org/10.1145/2384616.2384679

[6] Thomas Ball and James R. Larus. 1996. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 46–57.

[7] Andrew R. Bernat and Barton P. Miller. 2007. Incremental call-path profiling. *Concurrency and Computation: Practice and Experience* 19, 11 (2007), 1533–1547. https://doi.org/10.1002/cpe.1125

[8] Michael D. Bond, Graham Z. Baker, and Samuel Z. Guyer. 2010. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. 13–24. https://doi.org/10.1145/1806596.1806599

[9] Michael D. Bond and Kathryn S. McKinley. 2007. Probabilistic calling context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. 97–112. https://doi.org/10.1145/1297027.1297035

[10] Pietro Borrello, Andrea Fioraldi, Daniele Cono D'Elia, Davide Balzarotti, Leonardo Querzoni, and Cristiano Giuffrida. 2024. Predictive Context-sensitive Fuzzing. In *NDSS*.

[11] Milind Chabbi, Wim Lavrijsen, Wibe de Jong, Koushik Sen, John Mellor-Crummey, and Costin Iancu. 2015. Barrier elision for production parallel programs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. 109–119. https://doi.org/10.1145/2688500.2688502

[12] Milind Chabbi, Xu Liu, and John Mellor-Crummey. 2014. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 76–86.

[13] Milind Chabbi and John Mellor-Crummey. 2012. Deadspy: a tool to pinpoint program inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 124–134.

[14] Monika Chandrasekaran, Anca Ralescu, David Kapp, and Temesgen Kebede. 2021. Malware Detection using the Context of API Calls. In *NAECON 2021 - IEEE National Aerospace and Electronics Conference*. 92–97. https://doi.org/10.1109/NAECON49338.2021.9696310

[15] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[16] Long Fei and Samuel P. Midkiff. 2006. Artemis: practical runtime monitoring of applications for execution anomalies. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. 84–95. https://doi.org/10.1145/1133981.1133992

[17] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. 2003. Anomaly detection using call stack information. In *2003 Symposium on Security and Privacy, 2003*. IEEE, 62–75.

[18] Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek. 2020. Contextual dispatch for function specialization. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–24.

[19] Todd Gamblin, Martin Schulz, Bronis R de Supinski, Felix Wolf, Brian JN Wylie, et al. 2011. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 640–651.

[20] Jungwoo Ha, Christopher J. Rossbach, Jason V. Davis, Indrajit Roy, Hany E. Ramadan, Donald E. Porter, David L. Chen, and Emmett Witchel. 2007. Improved error reporting for software that uses black-box components. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. 101–111. https://doi.org/10.1145/1250734.1250747

[21] Kim Hazelwood and David Grove. 2003. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003*. 253–264. https://doi.org/10.1109/CGO.2003.1191550

[22] Chung Hwan Kim, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Xiangyu Zhang, and Dongyan Xu. 2014. IntroPerf: transparent context-sensitive multi-layer performance inference using system stack traces. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems* (Austin, Texas, USA) *(SIGMETRICS '14)*. 235–247. https://doi.org/10.1145/2591971.2592008

[23] Sungkeun Kim, Farabi Mahmud, Jiayi Huang, Pritam Majumder, Chia-Che Tsai, Abdullah Muzahid, and Eun Jung Kim. 2022. WHISTLE: CPU

Abstractions for Hardware and Software Memory Safety Invariants. *IEEE Trans. Comput.* (2022).

[24] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (jul 1976), 385–394. https://doi.org/10.1145/360248.360252

[25] Jianjun Li, Zhenjiang Wang, Chenggang Wu, Wei-Chung Hsu, and Di Xu. 2014. Dynamic and adaptive calling context encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 120–131.

[26] libunwind. 2022. https://www.nongnu.org/libunwind

[27] Bozhen Liu and Jeff Huang. 2022. SHARP: fast incremental context-sensitive pointer analysis for Java. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 88 (apr 2022), 28 pages. https://doi.org/10.1145/3527332

[28] Xu Liu and John Mellor-Crummey. 2011. Pinpointing data locality problems using data-centric analysis. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 171–180.

[29] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based memory allocation for C++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 541–556.

[30] David Melski and Thomas W. Reps. 1999. Interprocedural Path Profiling. In *Proceedings of the 8th International Conference on Compiler Construction (CC '99)*. Springer-Verlag, Berlin, Heidelberg, 47–62.

[31] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive Function Specialization against Code Reuse Attacks. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. 17–33. https://doi.org/10.1109/EuroSP48549.2020.00010

[32] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. 2009. Inferred call path profiling. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. 175–190. https://doi.org/10.1145/1640089.1640102

[33] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. 89–100. https://doi.org/10.1145/1250734.1250746

[34] Flemming Nielson, Hanne Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. springer. https://doi.org/10.1007/978-3-662-03811-6

[35] Gabriel Poesia and Fernando Magno Quintão Pereira. 2020. Dynamic dispatch of context-sensitive optimizations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[36] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 101–111.

[37] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/using-valgrind-detect-undefined-value-errors-bit

[38] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. 2015. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 401–413.

[39] SPEC2017. 2017. SPEC releases major new CPU benchmark suite. https://www.spec.org/cpu2017/press/release.html.

[40] J. M. Spivey. 2004. Fast, accurate call graph profiling. *Softw. Pract. Exper.* 34, 3 (mar 2004), 249–264. https://doi.org/10.1002/spe.562

[41] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS 2016)*. 84–95. https://doi.org/10.1145/2989225.2989236

[42] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant loads: A software inefficiency indicator. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 982–993.

[43] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

[44] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. 2011. Precise calling context encoding. *IEEE Transactions on Software Engineering* 38, 5 (2011), 1160–1177.

[45] SVFGit. 2022. Static Value-Flow Analysis Framework. https://github.com/SVF-tools/SVF

[46] Kapil Vaswani, Aditya V. Nori, and Trishul M. Chilimbi. 2007. Preferential path profiling: compactly numbering interesting paths. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. 351–362. https://doi.org/10.1145/1190216.1190268

[47] Ben Weidermann. 2007. *Know your place: Selectively executing statements based on context*. Technical Report. University of Texas at Austin.

[48] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. Redspy: Exploring value locality in software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 47–61.

[49] Shasha Wen, Xu Liu, and Milind Chabbi. 2015. Runtime value numbering: A profiling technique to pinpoint redundant computations. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 254–265.

[50] Qiang Zeng, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, and Peng Liu. 2014. DeltaPath: Precise and Scalable Calling Context Encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. 109–119. https://doi.org/10.1145/2544137.2544150

[51] Qidong Zhao, Xu Liu, and Milind Chabbi. 2020. DrCCTProf: A fine-grained call path profiler for ARM-based clusters. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[52] Keren Zhou, Jonathon Anderson, Xiaozhu Meng, and John Mellor-Crummey. 2022. Low overhead and context sensitive profiling of GPU-accelerated applications. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Article 1, 13 pages. https://doi.org/10.1145/3524059.3532388

[53] Tong Zhou, Michael R. Jantz, Prasad A Kulkarni, Kshitij A. Doshi, and Vivek Sarkar. 2019. Valence: variable length calling context encoding. In *Proceedings of the 28th International Conference on Compiler Construction*. 147–158.

[54] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. 2006. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. 263–271. https://doi.org/10.1145/1133981.1134012

Received 2024-11-13; accepted 2024-12-21