ANALYSIS OF CALLING CONTEXT ENCODING AND DECODING ALGORITHMS

An Undergraduate Research Scholars Thesis

by

VICTORIA E. RIVERA CASANOVA

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Eun J. Kim

May 2022

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Victoria E. Rivera Casanova, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Analysis of Calling Context Encoding and Decoding Algorithms

Victoria E. Rivera Casanova
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Eun J. Kim
Department of Computer Science and Engineering
Texas A&M University

The calling context of a program is recorded via a call stack for event logging, debugging, and profiling. There are several calling context encoding and decoding schemes that record the calling context of a program. One such scheme we are introducing is DCCE, Distinguished Calling Context Encoding; it can encode a program's calling context using a single integer ID without the need to decode it later. Without the need to decode, DCCE has less overhead costs than other popular encoding schemes. Another advantage of DCCE is that it can distinguish between different calling contexts that have the same encoded ID and different ending nodes/functions. We want to compare DCCE with other existing algorithms in terms of running time and measure the improved efficiency overall. This research paper discusses the practical uses of calling context encoding, implementation methods for DCCE, and the efficiency improvements of DCCE compared to CCTLib encoding. Through our experiment, DCCE outperformed CCTLib by over 2 times of overall execution time.

DEDICATION

To my family, friends, instructors, and peers who supported me throughout the research process.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Eun Kim, and my research mentor, Sungkeun Kim, for their guidance, patience, and support throughout the course of this research.

Thank you to my friends and colleagues and the department faculty and staff for always encouraging me throughout this process and making my time at Texas A&M University a great experience.

Finally, thanks to my amazing parents and brothers for their encouragement, patience, and love.

The concept and baseline of Distinguished Calling Context Encoding used for Analysis of Calling Context Encoding and Decoding Algorithms were provided by Sungkeun Kim. The integration of ZeroSpy with CCTLib Pin Tool analyzed in Analysis of Calling Context Encoding and Decoding Algorithms was conducted in part by Sungkeun Kim.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

No departmental or outside funding was used in the scope of this research.

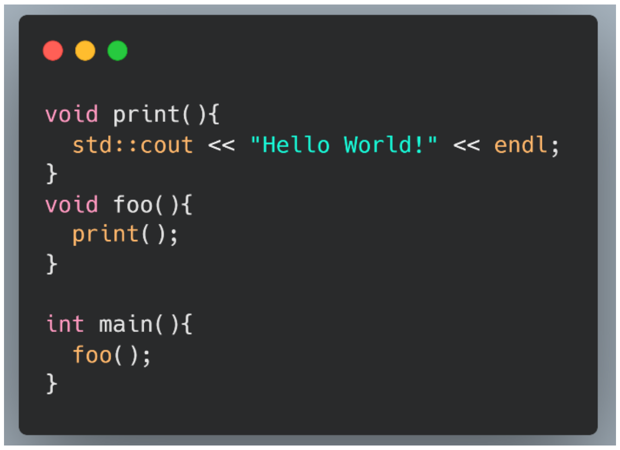
NOMENCLATURE

DCCE	Distinguished Calling Context Encoding
PCCE	Precise Calling Context Encoding

1. INTRODUCTION

Computer programs have many different parts to them that need to run and work together to execute the program correctly. Typically, programmers write their code line-by-line, in a top to bottom manner. However, a computer program typically does not execute in a linear fashion. The execution of instructions starts from the main function and from there it can branch off to different parts of the code depending on the intended behavior. Functions are specific sections of code that perform certain tasks and can be called at any point within the program. Functions can be called multiple times and functions themselves can call other functions. When one function is called and is done executing, the program will return to where the previous function made that call.

During the execution of a computer program, there are many instances of function entries and exits. Whenever a new function is called, it is recorded via a call stack which keeps track of all the active function calls that are running. Once a function has finished, it is removed from the top of the call stack and the program returns to where the call was made and continues its execution from that point. This call stack is also known as the program's calling context. A calling context refers to the contents within the entire call stack itself. It is a series of active function calls that specifies the program's current location during execution [5]. The information within the call stack is used by developers for many different purposes, some of which are event logging, profiling, and debugging [3]. However, the call stack can get expensive in terms of memory allocation when dealing with large programs, and it becomes inefficient to walk through the entire stack when running large programs with many function calls. For these situations, calling context encoding is leveraged to store the call stack information in a more efficient way.



```
void print(){
  std::cout << "Hello World!" << endl;
}
void foo(){
  print();
}

int main(){
  foo();
}
```

Figure 1.1 Simple C++ code snippet with multiple function calls

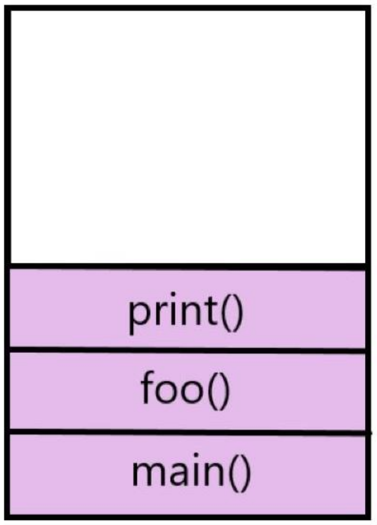


Figure 1.2: Call stack for code snippet

Figure 1.1 demonstrates a simple C++ program with multiple levels of function calls. From the main function, the function foo() is called. While foo() is executing, there is another function call to the function print(). Figure 1.2 represents the program's call stack once it has reached the first instruction in the function print() during execution. Ideally, calling context encoding can be used to store the call stack information from Figure 1.2.

1.1 Existing Calling Context Encoding and Decoding Algorithms

Currently, there are many algorithms that will compress and store a program's calling context information in a more efficient manner than walking through the entire stack itself. Some of the encoding schemes used today are Call Stack Unwinding and Stack Shadowing [2]. These algorithms approach the calling context encoding question using different strategies, data structures and instrumentation frameworks. Binary instrumentation refers to injecting additional instructions into a program's binary based on the existence of certain events. There are multiple ways to instrument a program's binary. Dynamic instrumentation varies from static instrumentation because bits of additional code are added to the program's binary during the execution of said program. Static instrumentation is adding code on the binary that is generated during compile time before the application starts running. A couple of key algorithms that were studied for this research topic are Precise Calling Context Encoding, Valence, and CCTLib. These algorithms approach context encoding in their unique way. They are well-known algorithms that will be tested against our own encoding algorithm.

Valence, a compiler pass based algorithm, encodes calling contexts as a list of bit values. The creators argue that sometimes it is unnecessary to store an entire word to represent a piece of the calling context; a few bits are sufficient [6]. The bit values come from a call graph that is generated using static analysis. Each node is given a level value that represents the maximum number of bits needed to encode any potential call path to said node [6]. The edge weights are the bit values that are appended to the list of bits that represent the calling context. Valence is efficient in terms of reducing the length of encoding compared to other existing algorithms such as Precise Calling Context Encoding [6]. It does, however, still require overhead for decoding.

PCCE is an encoding algorithm that encodes the current call stack of a program with an integer ID using static binary instrumentation [3]. This encoding algorithm generates a call graph with corresponding edge weights that are used to update the calling context ID [3]. The context ID is updated using arithmetic operations before and after each function call. This algorithm works with recursive function calls and requires a decoding algorithm. With the unique context ID and the last known visited function, the context can be decoded by the decoding function provided by PCCE [3]. However, the main drawback of PCCE is that it cannot distinguish between different calling contexts with the same encoded ID that end on different functions (or nodes in the call graph). In other words, the algorithm requires additional information other than the encoded ID to tell it apart from a distinct calling context with the same integer ID. As of now, there is not an established encoding algorithm that can make this distinction that we know of.

1.2 CCTLib

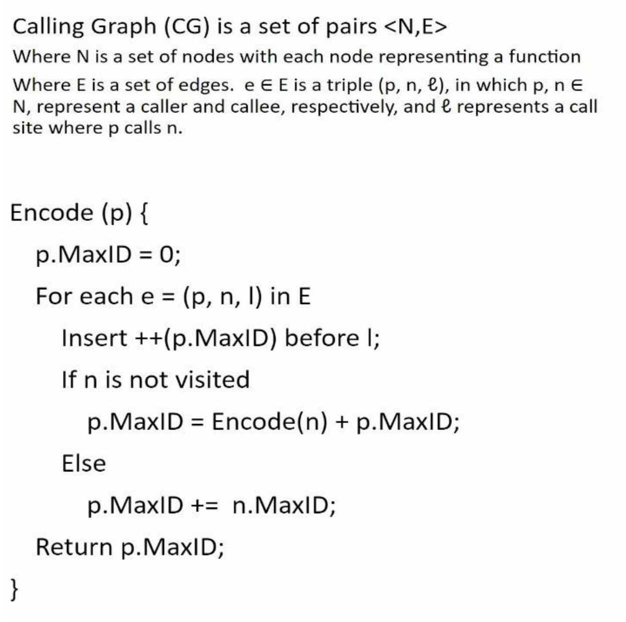
The focus of this research takes inspiration from the most recent encoding work CCTLib which provides pinpoints software inefficiencies with fine-grained software monitoring. CCTLib takes a different approach from Valence and Precise Calling Context Encoding as it is an encoding scheme that leverages Intel’s Pin Tool framework for dynamic binary instrumentation. CCTLib is a tool that can obtain the calling context at any/every machine instruction during execution time [1]. CCTLib uses the Pin Tool API to monitor and instrument every Pin trace entry. It assigns a unique identifier to the current trace and every call and return machine instruction. At each monitored instruction, the client Pin tool calls a callback function that updates the CCT for that particular trace [1]. CCT refers to the data structure that contains the call path information at a given point during execution [1]. CCTLib is used in many event monitoring applications such as ZeroSpy, DeadSpy, and LoadSpy to keep track of the CCT for the target application. It is the most recent work for calling context encoding and will be used for comparison against DCCE.

The rest of the thesis is structured as follows: The background and motivation behind Distinguished Calling Context Encoding in Chapter 2, followed by the methodology of implementation in Chapter 3. Chapter 4 goes over the results of our experiments and the final chapter concludes the thesis.

2. DISTINGUISHED CALLING CONTEXT ENCODING

2.1 Distinguished Calling Context Encoding

We propose the Distinguished Calling Context Encoding algorithm, DCCE, to overcome long encoding/decoding time at runtime. With this new encoding scheme, calling contexts can be differentiated even if they have the same encoded ID and different ending functions/nodes [2]. DCCE updates the calling context ID at every function entry and exit. It is updated using an encoded edge weights from a statically generated call graph.



Calling Graph (CG) is a set of pairs <N,E>

Where N is a set of nodes with each node representing a function

Where E is a set of edges. e ∈ E is a triple (p, n, ℓ), in which p, n ∈ N, represent a caller and callee, respectively, and ℓ represents a call site where p calls n.

```
Encode (p) {
    p.MaxID = 0;
    For each e = (p, n, l) in E
        Insert ++(p.MaxID) before l;
        If n is not visited
            p.MaxID = Encode(n) + p.MaxID;
        Else
            p.MaxID +=  n.MaxID;
    Return p.MaxID;
}
```

Figure 2.1: Pseudocode for encoding DCCE edge weights

Figure 2.1 describes the process for encoding the edge weights for each connection in the call graph. Right before every function call, the edge weight that corresponds with the caller and called function is added to the CCID, calling context ID. Once the active function finishes (returns to its caller function), we subtract the same edge weight we had previously added before the function entry from the CCID. This is a similar process to PCCE. With this encoding scheme, the CCID does not need a decoding component because the nature of DCCE allows it to be able to differentiate calling contexts with different ending nodes given the current CCID. DCCE updates the CCID dynamically by leveraging Intel’s Pin Tool framework, similar to the work of CCTLib. DCCE adds more overhead during encoding but significantly less overhead for decoding when compared to PCCE since it does not require it at all.

There are several other components needed for the overall DCCE scheme. The Pin tool version requires several components to perform offline work in addition to the DCCE Pin tool itself. There is a separate Pin tool that analyzes and records the target program’s function calls in an output file. This file is then passed through to an encoding script that generates and writes the encoded edge weights for the program’s call graph. Once these two offline pieces are complete, that updated file can be passed to the main DCCE Pin tool as input.

DCCE can be used as a substitute in various applications requiring calling context monitoring using stack shadowing or other similar encoding algorithms. We look to substitute existing encoding/decoding schemes with our novel DCCE Pin tool library with the intention of improving the overall efficiency in terms of execution time for the selected analysis programs.

3. METHODS

The Distinguished Calling Context Encoding algorithm was developed to address the limitations of existing calling context encoding schemes. The algorithm is an improvement from PCCE since it allows for distinction between calling contexts with the same ID that have different ending nodes. DCCE is implemented as an analysis tool using Intel's Pin framework for dynamic binary instrumentation. For simplicity, this version of DCCE's Pin tool implementation monitors all machine instructions and performs a callback on every function call and function exit. The reason behind utilizing Pin for DCCE's implementation is motivated by the need to test the algorithm's efficiency when replacing it with other encoding schemes in larger monitoring applications. We look to replace CCTLib's encoding algorithm with DCCE for calling context encoding in monitoring applications such as ZeroSpy to demonstrate an improvement in terms of efficiency in encoding cycles when the switch is made.

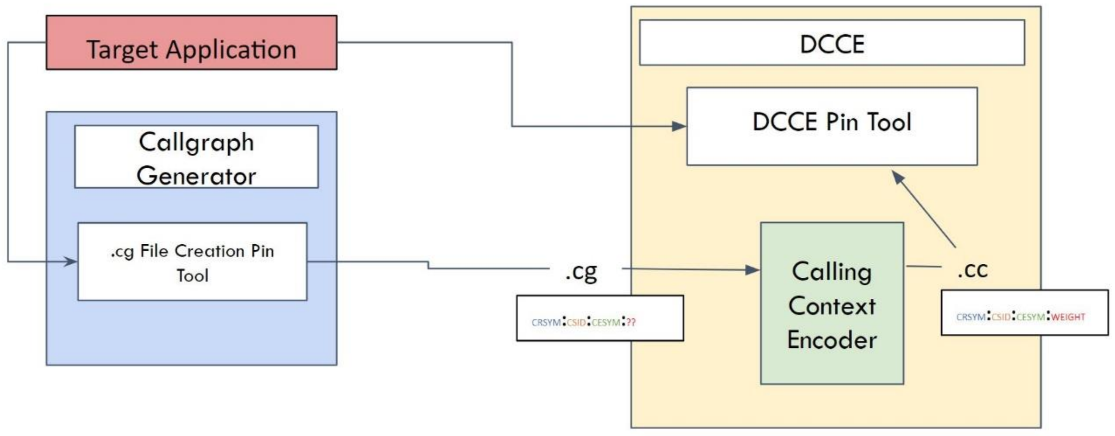
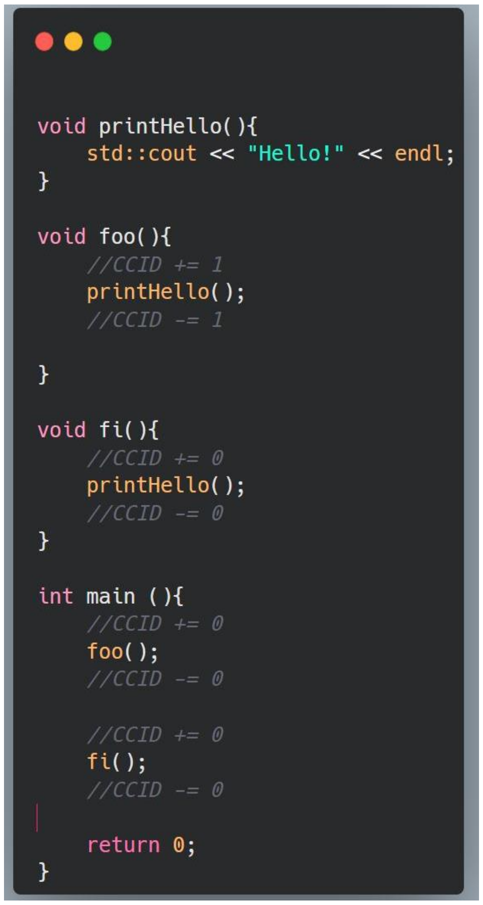


Figure 3.1: Flow Diagram of DCCE Pin Tool Implementation

Figure 3.1 displays the basic project flow for DCCE. As the figure shows, DCCE will take in a target application's binary file and process that through our initial Pin tool, Callgraph Generator. The call graph Pin tool will take in that binary, analyze all the call instructions within the binary and filter out all the system call instructions leaving only the call instructions from the main executable of the target program. Using our callback function, at every valid call instruction its call site, name of caller function, and name of the function being called will be written into the output file in a specific format. The output will be specified as a .cg file. The .cg will be inputted into the "Calling Context Encoder" Python script offline to generate the edge weights for each edge in our call graph. The output of this program is a .cc file which holds the calling context information of each call relationship. The .cc will be used as input to our final DCCE Pin tool along with the initial target application binary file. These two files will be used by the DCCE Pin tool to dynamically encode the context of the target application at every point during execution.

3.1 Overview

Distinguished Calling Context Encoding allows us to encode calling contexts within a single integer ID without needing to decode said ID. Instrumentation is done before and after each function call. We add the encoded edge weight to our CCID before the call and subtract that same edge weight after the function call.



```cpp
void printHello(){
    std::cout << "Hello!" << endl;
}

void foo(){
    //CCID += 1
    printHello();
    //CCID -= 1

}

void fi(){
    //CCID += 0
    printHello();
    //CCID -= 0
}

int main (){
    //CCID += 0
    foo();
    //CCID -= 0

    //CCID += 0
    fi();
    //CCID -= 0

    return 0;
}
```

Figure 3.2: Source code of C++ program with multiple function calls. Contains CCID update examples.

Figure 3.2 contains source code for a slightly more complex version of Figure 1.1. Now, there are two functions, foo() and fi(), that call the print function. The comments before and after the function calls show how, theoretically, DCCE would instrument the binary of this piece of source code. DCCE uses graph theory to generate a corresponding call graph representation for the source code.

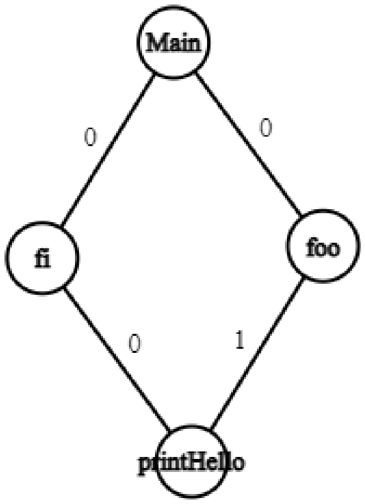
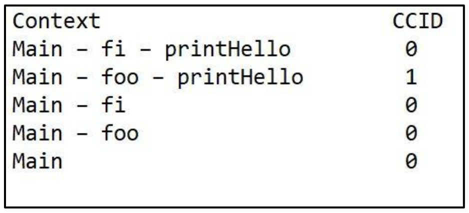


Figure 3.3: Call graph representation of source code in Figure 3.1



```
Context                    CCID
Main – fi – printHello       0
Main – foo – printHello      1
Main – fi                    0
Main – foo                   0
Main                         0
```

Figure 3.4: CCs for 2.3

From Figure 3.3, we can see that each function is represented as a node in the graph, and every function call is represented by the edges connecting the nodes. The numbers on each edge are the encoded edge weights for each function call connection generated by the Calling Context Encoder. The details of how the encoded edge weights are generated are beyond the scope of this research topic. Figure 3.4 shows the different encoded calling contexts for each scenario that can occur for our example.

3.2 Intel's Pin Tool

This version of DCCE is implemented as a dynamic binary instrumentation tool using Intel's Pin tool framework. The Pin framework can be used by IA-32 and x86-64 computer architectures. It is generally used for dynamic binary analysis on any target application. The target program's binary files are instrumented during run time, instead of statically after compiler time. The library provides an extensive API for fetching information while the target program is running. Leveraging the functions provided by the API, DCCE monitors all instructions, checking for function call and function exit instructions. Before the actual application starts executing, Pin will monitor the binary of the target application and look for certain events. In our case, it is looking for function calls and function exits. When one of these events is registered, Pin will make note of it and when the target application is executing, it will trigger a callback function that we have written for each specific case. The callback function changes the behavior of the program and does not actually change or write to the target application's binary file. The callback functions take in certain parameters as input and update the current CCID accordingly.

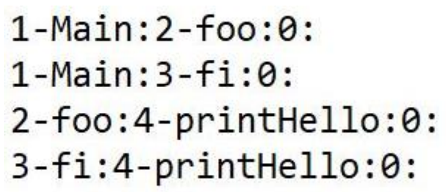
For this research, we downloaded the Linux x86-64 architecture option for Pin. The version of Pin we are using is 3.13 as opposed to the latest release. All development was done in a Linux environment. We used the C++ programming language to write our DCCE Pin tool and Pin's default compiler needs to be the 3.8 version of the GNU C++ compiler (at least for our purposes).

Important API functions that were vital in the DCCE Pin tool implementation were INS_Address(ins), which gave us the memory address of the current instruction. For our purposes, we use it to get the memory address of the current function call (caller) instruction. INS_DirectControlFlowTargetAddress(ins) is used to give is the memory address of the first instruction within the callee function. And finally, INS_InsertPredicatedCall(ins, …) is used to make a call to our callback functions when the event of a function call or exit is triggered. We use INS_InsertPredicatedCall(ins, …) instead of the normal call to INS_InsertCall(ins, …) in order to avoid potential errors in instrumenting instructions that are not actually executed by the program itself.

3.3 Implementation

Once Pin was installed successfully and we verified the example Pin tools were executed correctly, we could begin to implement the necessary parts of the DCCE scheme.

As shown in Figure 2.1, the first piece is the Callgraph Generator. This is a separate Pin tool that generates the .cg file that is needed as input to the DCCE Pin tool. Its purpose is to analyze the target application's binary file and record all the needed call functions from the main executable, excluding automated system function calls. Pin looks for any call instruction and does a check on that instruction's memory address. The memory address is a reference to the assigned memory location in the RAM of that machine instruction or variable. It is in bytes and usually represented in a hexadecimal format. We filter out automatic program start-up system call instructions from being written to the output files but do not worry about the system calls associated with function entries and exits since they have minimal to no consequences on the analysis and outcome of our tests. Once the Callgraph Generator finishes its analysis routine, the output file contains the important call graph information in the format: "*callerID-callerName:calleeID-calleeName:callSite:*". The caller refers to the function within which a new function call is being made. The callee refers to the function that is being called.

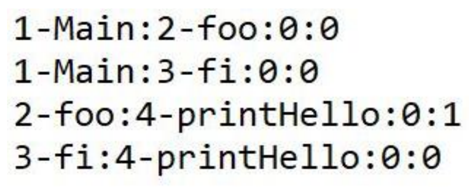


```
1-Main:2-foo:0:
1-Main:3-fi:0:
2-foo:4-printHello:0:
3-fi:4-printHello:0:
```

Figure 3.5: Example of the .cg file for Figure 3.2 code

Figure 3.5 is the corresponding .cg file for the source code found in Figure 3.2. Each function has its own function ID and each line within the file represents a caller-callee connection. The call site is actual the memory address of the next instruction after the called function has finished executing and returned. In this example, we have set it arbitrarily to 0. But, for our implementation, the call site will list the actual memory address value corresponding to where the call instruction was made.

For our implementation, the DCCE Pin tool opens and reads a .cc file alongside the target application's binary. The call graph (.cg) file is run through a separate program, named Calling Context Encoder, that generates an edge weight for each connection between nodes on the call graph. This is done offline as the second step in our overall scheme flow; separate from the Pin tool components. The Python program responsible for generating the encoded edge weights takes in the .cg file and appends the encoded integer ID to the end of each line. The format for each line in the .cc file is "*callerID-callerName:calleeID-calleeName:callSite:weight*".



```
1-Main:2-foo:0:0
1-Main:3-fi:0:0
2-foo:4-printHello:0:1
3-fi:4-printHello:0:0
```

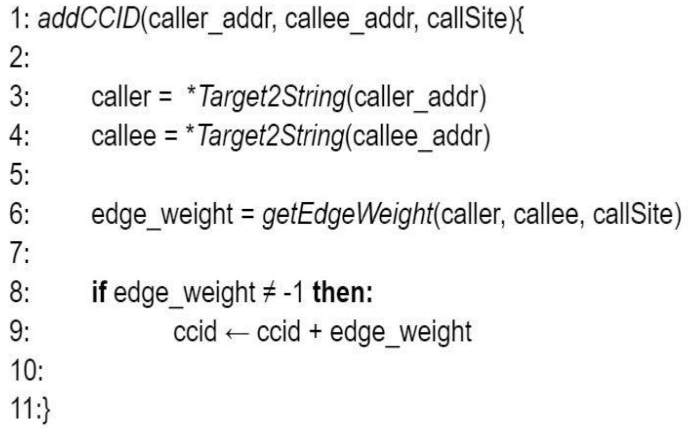
Figure 3.6: Example of the .cc file for Figure 3.2

Figure 3.6 represents the .cc file that is outputted by the Calling Context Encoder Python script. It differs from the .cg file because the encoded edge weight is appended to the end of the .cg file after the colon. The encoded edge weights are used to update the CCID during runtime. Each line within the .cc file contains the caller function name, the callee function name, the memory address associated with the caller function, and the generated edge weight. The file is then parsed by the DCCE Pin tool, and its information is stored in a nested map data structure. This allows for efficient access to the data for when the algorithm needs to look up the edge weight for a particular caller and callee relationship. Using a nested map is more efficient for lookup purposes than parsing the entire .cc file at every function call. This is especially true if the .cc file contains many call relationships for a complex program.

Parsing the call graph (.cc) file occurs before our Pin tool program begins its instrumentation routine. Once the parsing function has returned, the instruction instrumentation routine is called. Within the instruction instrumentation routine, we have the Instruction(INS ins, VOID *v) method that does a check on each machine instruction. The DCCE Pin tool does an initial pass through the target program's binary. This initial pass is done before the execution of the target application. It looks for call function and function exit events. It checks to see if the current instruction is a function call or a function exit. If it is neither, no additional instrumentation occurs. However, there is a couple of function within our Pin tool that are called in the event of a function call or exit. At these events, the DCCE Pin tool makes a note and will trigger one of our prewritten callback functions during execution time. It is at these callback functions where the calling context information stored in the CCID, calling context ID, will be updated accordingly.

Two callback functions were written in the event of a function call or function exit. If there is a call instruction, the Pin tool will execute INS_InsertPredicatedCall(…) with the addCCID() callback function as an argument. If there is a function exit, DCCE needs to subtract the correct encoded edge weight from the CCID. Again, the Pin tool will call INS_InsertPredicatedCall(…), however this time we need to pass subtractCCID() as an argument.

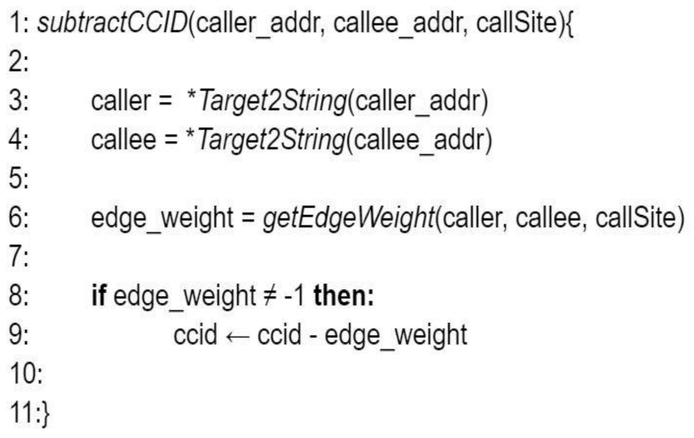
Our unique callback functions take as input the memory address of the caller function and the memory address of the callee function. Inside the callback functions, the memory addresses are passed to a previously implemented function found in Pin's sample program calltrace.cpp, *Target2String(ADDRINT target). This function takes in a memory address and, using a built-in Pin API function, returns the name of the function associated with that memory address. With the string names of the caller and callee functions, in addition to the call site value, we look up the corresponding edge weight from within our populated nested map structure using the function getEdgeWeight(…). If there is a match, then we use that value to update the CCID depending on if it is a function call or function exit. If the value that is returned from getEdgeWeight(…) is -1, there is no match and no changes to the CCID occur.



```
addCCID(caller_addr, callee_addr, callSite){

    caller =  *Target2String(caller_addr)
    callee = *Target2String(callee_addr)

    edge_weight = getEdgeWeight(caller, callee, callSite)

    if edge_weight ≠ -1 then:
        ccid ← ccid + edge_weight

}
```

Figure 3.7: Pseudocode for addCCID() callback function from DCCE

Figure 3.7 shows the pseudocode version of the callback function when DCCE needs to add the edge weight to the CCID in the event of a call instruction. The other function subtractCCID() is written in a similar fashion.



```
1: subtractCCID(caller_addr, callee_addr, callSite){
2:
3:        caller =  *Target2String(caller_addr)
4:        callee = *Target2String(callee_addr)
5:
6:        edge_weight = getEdgeWeight(caller, callee, callSite)
7:
8:        if edge_weight ≠ -1 then:
9:                ccid ← ccid - edge_weight
10:
11:}
```

Figure 3.8: Pseudocode for subtractCCID() callback function from DCCE

Figure 3.8 is the pseudocode version of subtractCCID(). The main difference between the two callback functions would come in line 9, where instead of adding to the CCID the edge weight is subtracted.

3.4 Integration with Monitoring Program Clients

With the completed implementation of Distinguished Calling Context Encoding as a Pin tool, it is necessary to integrate our Pin tool with existing monitoring applications, one of which is ZeroSpy. ZeroSpy is a pure software monitoring tool that identifies and reports where there are redundant zeros stored in memory to reduce inefficiencies and optimize performance for the target programs [4]. ZeroSpy itself is implemented as a Pin tool that monitors every memory load instruction of the given target application and leverages CCTLib calling context encoding in its analysis routines [4]. There are instances where ZeroSpy requires calling context information to identify redundant zeros in memory and it calls functions within the CCTLib library. We are interested in ZeroSpy due to its utilization of the CCTLib Pin tool library.

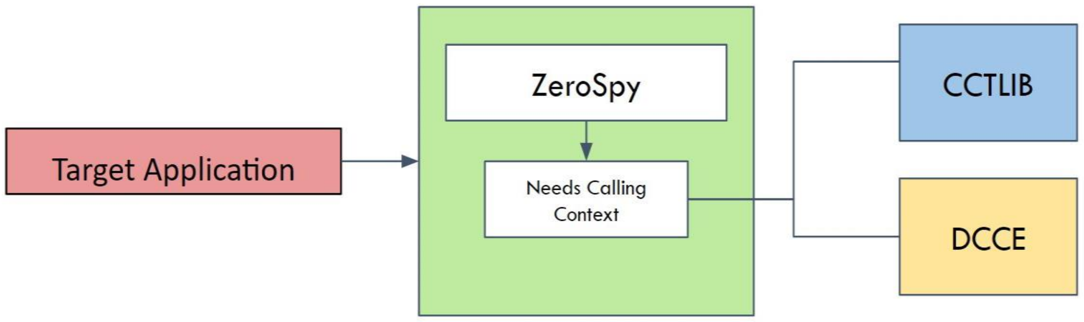


Figure 3.9: Pin tool testing overview

For us to verify DCCE is a more efficient encoding scheme than CCTLib, we need to run benchmark tests on ZeroSpy with CCTLib encoding vs. ZeroSpy with our novel DCCE encoding scheme. Figure 3.9 presents the overall flow of how we will be testing the different encoding schemes. The same target application will be used for both tests to maintain consistency. Wherever ZeroSpy requires the calling context for its respective analysis, we will replace the CCTLib library with DCCE's library.

Our DCCE Pin tool is converted to a Pin tool library. This means that some of DCCE's internal functions, such as returning the value of the encoded CCID, can be called from separate applications such as ZeroSpy. It no longer executes and runs its analysis routine concurrently with the running target application but allows its internal functions to be utilized when needed by other applications.

Due to the scope of this research, a simplified version of the ZeroSpy Pin tool was created. Instead of checking for redundant zeros at each memory access point, our simplified version still monitors for memory read and memory write operations within the target application, but it also instruments call and return instructions. Each of these 4 events triggers a callback function that in turn calls a function from a calling context library (i.e., DCCE library or CCTLib library). Memory reads and memory writes will register the callback function getCCID() which will subsequently call a function within either DCCE or CCTLib's library (depending on which scheme is being tested) that returns the current calling context at that point. Call and return instructions will trigger the callback functions that update the calling context accordingly. These are also functions provided by the calling context libraries.

The integration of the CCTLib library with the ZeroSpy simple version was done outside the scope of this research. Once the integration of both encoding schemes with ZeroSpy simple version was completed, tests were run on the novel ZeroSpy implementation with DCCE and the baseline model with CCTLib. Each benchmark program was passed through every piece of the DCCE workflow that is described in Figure 2.1 to render the correct input files and callgraph information. This is considered offline work so its execution time is not taken into consideration in the testing of the encoding/decoding runtime.

The same steps used for the integration of ZeroSpy with both DCCE and CCTLib libraries were repeated for the simplified versions of LoadSpy and DeadSpy, two other memory access monitoring tools.

3.5 Experiment Design

We will mainly look for differences in runtime in terms of cycles for the whole program and latency of specific functions. CPU cycles in computer terms refer to the time it takes for a processor to execute an operation. Since our tests last only seconds from start to finish, selecting cycles as our main time-keeping unit will provide a more accurate description of the time difference of each encoding scheme for each client tool.

Each client tool monitors for memory access operations throughout the execution of the target program. LoadSpy looks for memory read operations, DeadSpy looks for memory write operations, and ZeroSpy monitors for both these types of memory access operations. At each of their respective memory access instructions, the client tools will call the callback function getCCID() which returns the current calling context information at that point in the program's execution. We are interested in the timing of this function because CCTLib and DCCE have different implementation methods for returning calling context information. This data is useful in showing the speed-up time by DCCE compared to CCTLib.

In addition to monitoring memory access operations, the clients also observe for call and return instructions. These instructions will trigger the action of updating the current calling context information for the program through other callback functions. DCCE's calling context updates are simple addition and subtraction operations that can be easily timed. However, since CCTLib's encoding scheme is more complex in nature it is difficult to get an accurate timing without also timing extra unnecessary overhead operations. Due to this, we do not time each individual callback function for updating the calling contexts but instead record the execution time of the entire client after the required Pin tool start-up functions have finished. This allows for a fair comparison between the two encoding schemes. Running the clients is done through a command line instruction that specifies 1) the target program executable 2) the client tool's name and 3) the encoding scheme to be used. After the tool has finished execution, an output file is created to the current working directory with statistics on total program execution time and the average latency of the getCCID() function.

4. RESULTS

To test the efficiency of Distinguished Calling Context Encoding with other existing calling context encoding schemes, mainly CCTLib, we ran simplified client versions of common program monitoring applications for ZeroSpy, LoadSpy, and DeadSpy. These client programs were each run to monitor a selected test program and called for either DCCE or CCTLib encoding schemes to be utilized in their execution. The times that were recorded were for the entire execution time of the program as well as the average latency of the getCCID() function that returned the current calling context at any point in the execution.

4.1 Experiment Results

CCTLib provides a test program in its GitHub repository titled deadWrites.cpp and we used this same test as the basis for our experiments.

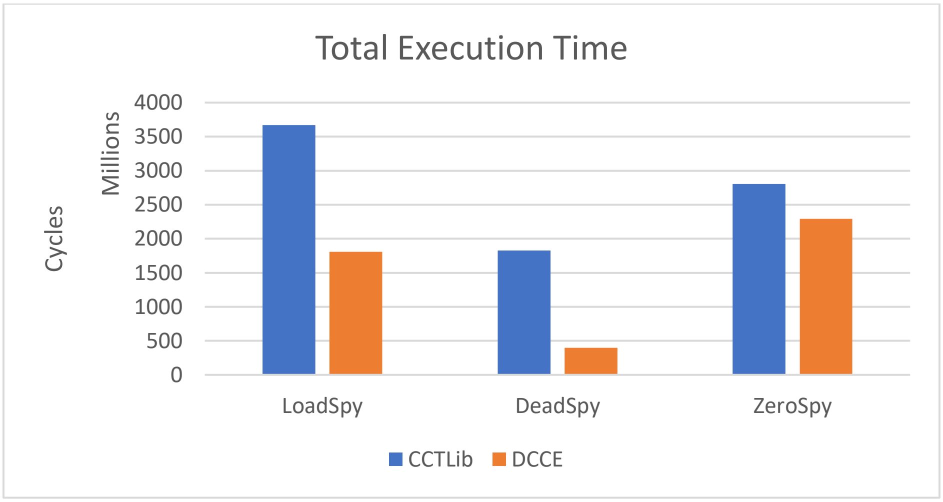


Figure 4.1: Execution Time of different clients in units of cycles.

Figure 4.1 compares the average execution time of the three client tools using either DCCE or CCTLib encoding. Each client tool was run 5 times and the averages of the results are depicted in the chart. Based on the results shown, the DCCE encoding scheme provides significant speed-up to the overall execution time as opposed to CCTLib in all 3 of the client tool tests. This is seen most prominently in the LoadSpy and DeadSpy clients where the difference between the two is in the range of about 1,400,000 – 1,800,000 cycles. The average speed-up for the DCCE clients against the CCTLib clients is over 2 times the execution time.

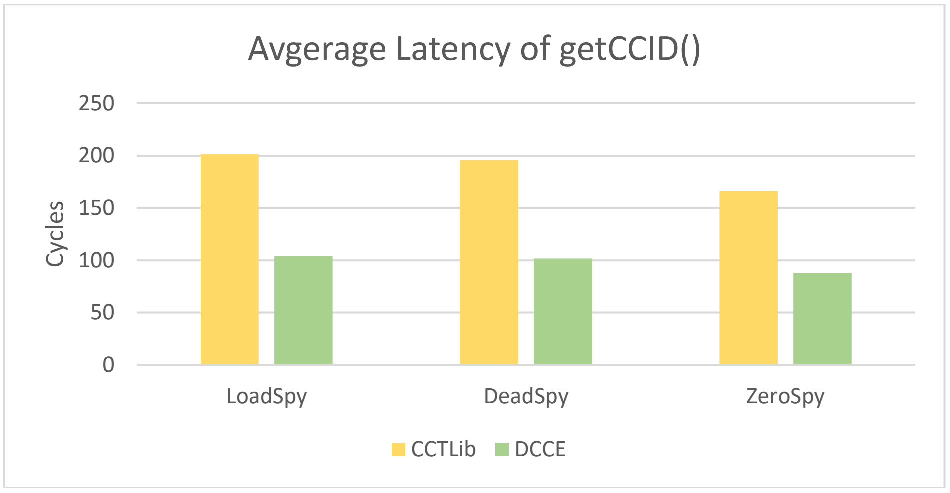


Figure 4.2: Average latency of getCCID() for different clients in units of cycles

Figure 4.2 offers comparisons in the average execution time for the individual function getCCID(). This data was recorded from the same group of tests as for Figure 4.1. Similar to the results in Figure 4.1, it can be seen that getCCID() for DCCE runs faster than the CCTLib version. This is because while DCCE only has to return the integer value of the global variable CCID, CCTLib has to provide a decoding element to return the calling context information. Those operations are much more expensive and time consuming than the simplistic approach of DCCE.

The overall results of the experiment matched our initial hypothesis that DCCE is a faster encoding scheme than CCTLib due to its implementation and use of simple arithmetic operations when updating the calling context. DCCE also has less overhead cost than CCTLib when looking at purely updating current calling context and retrieving calling context information at any given point during execution.

4.2 Things to Consider

The experiments were run on a shared High-Performance server operated by Texas A&M University. While we were allocated space on the server, other users and processes are able to run programs at the same time as our experiments. Program execution time can be affected by the status of CPU usage of the server at testing time. This is important to keep in mind in regard to the execution times that were recorded.

Regarding the test programs used for these experiments, we ran one C++ test program provided by the authors of CCTLib that can be found within the CCTLib GitHub repository. This program was used as the main benchmark instead of common benchmark suites like SPEC due to the complexity of the available test applications. Generating the call graph (.cg) file (as well as running the CC. Encoder for the .cc file) would theoretically take multiple hours and many gigabytes of storage to complete. Time and space restrictions did not allow us to properly run a program in the SPEC benchmark suite.

5. CONCLUSION

5.1 Conclusion

We introduced a novel calling context encoding scheme named Distinguished Calling Context Encoding. It encodes the calling context of any program using an integer ID and dynamic binary instrumentation using Intel's Pin Tool. This encoding scheme addresses several limitations of popular encoding schemes and aims to also have better performance in terms of total running time. DCCE guarantees distinction between calling contexts with the same CCID ending in different nodes within the call graph. There is no decoding overhead associated with DCCE, as opposed to commonly utilized schemes like Precise Calling Context Encoding, CCTLib, and Valence.

DCCE has a specific pipeline for execution that requires a call graph to be generated that contains function call information. This file is then passed through a Python script that assigns encoded integer edge weights to each caller and callee function relationship. After using DCCE as a substitute for the CCTLib Pin tool in the three simplified client monitoring applications, our results demonstrate that the programs leveraging the DCCE scheme had a faster total execution time in terms of cycles overall. DCCE also had a faster retrieval time of the current calling context information for the same tests. Distinguished Calling Context Encoding provides a faster, more efficient calling context encoding alternative to other existing schemes. The advancements DCCE provides can benefit many different sectors within computer science and program development. Calling context information is a vital tool for developers to leverage in certain sophisticated and complex computer programs related to debugging, profiling and stack monitoring. Encoding this information in the least expensive manner in terms of storage and in the fastest manner in terms of execution time can vastly improve the overall performances of these programs, allowing developers to allocate time and space to other aspects of the development process. In the future, DCCE can be a popular encoding scheme due to its improved performance and ability to distinguish calling contexts without decoding overhead or expensive stack walks.

5.2 Moving Forward

With DCCE proven to provide a faster alternative for calling context encoding than CCTLib, the next steps moving forward will be to run additional tests with other popular encoding schemes such as PCCE and Valence. Additionally, the current DCCE Pin tool will need code refactoring to make the code base more efficient and easier to maintain should further changes or updates to implementation occur.

REFERENCES

[1] Chabbi, M., Liu, X., & Mellor-Crummey, J. (2014). Call paths for PIN tools. *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. https://doi.org/10.1145/2544137.2544164

[2] Kim, S. (2021). Survey of Existing Calling Context Collection Techniques and Feasibility Study of DCCE.

[3] Sumner, W. N., Zheng, Y., Weeratunge, D., & Zhang, X. (2010). Precise calling context encoding. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10. https://doi.org/10.1145/1806799.1806875.

[4] X. You, H. Yang, Z. Luan, D. Qian and X. Liu, "ZeroSpy: Exploring Software Inefficiency with Redundant Zeros," SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020, pp. 1-14, doi: 10.1109/SC41405.2020.00033.

[5] Zeng, Q., Rhee, J., Zhang, H., Arora, N., Jiang, G., & Liu, P. (2014). DeltaPath. Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization - CGO '14. https://doi.org/10.1145/2581122.2544150.

[6] Zhou, T., Jantz, M. R., Kulkarni, P. A., Doshi, K. A., & Sarkar, V. (2019). Valence: Variable length calling context encoding. *Proceedings of the 28th International Conference on Compiler Construction - CC 2019*. https://doi.org/10.1145/3302516.3307351