2015 IEEE 21st International Conference on Parallel and Distributed Systems

Adaptive Path Profiling Using Arithmetic Coding

Gonglong Chen and Wei Dong
College of Computer Science, Zhejiang University
Email: {chengl, dongw}@emnets.org

***Abstract*—Path profiling, which aims to trace a program's execution path, has been widely adopted in various areas such as record and replay, program optimizations, performance diagnosis, and etc. Many path profiling approaches have been proposed in the literature, including B.L. algorithm, and PAP. Unfortunately, both approaches suffer from large tracing overhead for representing long execution paths. In this paper, we propose AdapTracer, a path profiling approach based on arithmetic coding. There are two salient features in AdapTracer. First, it is *space efficient* by adopting a path profiling algorithm based on arithmetic coding. Second, it is *adaptive* by explicitly considering the execution frequency of each edge. We have implemented AdapTracer to profile Android applications. Our experimental evaluation uses modified JGF benchmarks to show AdapTracer's efficiency. Experimental results show that AdapTracer reduces the trace size by 44% on average and incurs execution overhead by 10% at most compared to PAP.**

***Keywords*-Path profiling, arithmetic coding, adaptive.**

I. Introduction

Path profiling refers to the technique for tracing a program's execution path. A path profile gives information about the execution behavior of the program. It has been widely adopted in various areas such as record and replay [1], program optimizations [2, 3], performance diagnosis [4], and etc.

In their seminal work [5], Ball and Larus have described an efficient path profiling algorithm (called B.L. algorithm) using a compact numbering scheme to differentiate different paths in a program. Specifically, the program is first modeled as a control flow graph (CFG). When the CFG is a directed acyclic graph (DAG), the B.L. algorithm assigns a unique PathID in the range of $[0, n-1]$ (where n is the total number of paths in the DAG) to one execution path. When the CFG is not a DAG, the B.L. algorithm first transforms the graph into DAG by removing the back-edges. Multiple PathIDs are used to represent a cyclic path (path that has loops), which inevitably introduces a large overhead [6].

Recently, Li *et al.* propose PAP [7], an efficient path profiling algorithm for tracing all paths including acyclic and cyclic paths. It instruments probes on the multiple in-edges of each CFG node and uses addition and multiplication operations in the calculation of probe values. In this way, it can effectively profile all finite-length paths within a procedure. Then the PathID is used to restore the corresponpding path by doing division and modulo operations reversely. When long paths are executed, the probe value keeps growing and may overflow. The breakpoints mechanism is introduced in PAP to deal with this problem. A breakpoint consists of two elements: the CFG node and the probe value before overflow.

Unfortunately, both approaches suffer from large space overhead for representing a long execution path. For the B.L. algorithm, multiple PathIDs may be required for the representation. For PAP, multiple breakpoints may be required to solve the problem of PathID overflow. We also notice that both approaches are not *adaptive*, i.e., they use a fixed numbering scheme for tracing multiple executions of the same program. Hence, they lose the opportunity to optimize the space overhead for frequently executed paths.

To address the two problems mentioned above, we propose AdapTracer, an adaptive path profiling using arithmetic coding. There are two salient features in AdapTracer. First, it is *space efficient* by adopting a path profiling algorithm based on arithmetic coding. Different from PAP, AdapTracer instruments probes on the multiple outedges of each CFG node, and uses operations involved in the integer implementation of arithmetic coding [8] for calculating the probe values. Breakpoints for labelling a node in the CFG are not required since AdapTracer decodes the PathID from start to exit, unlike PAP which relies on reserve decoding. Second, it is *adaptive* by explicitly considering the execution frequency of each edge, which is recorded by the edge counter. With the help of edge counter, AdapTracer adjusts each edge's probability to achieve the close-to-optimal path encoding.

We have implemented AdapTracer to profile Android applications. Our experimental evaluation uses modified JGF benchmarks to show AdapTracer's efficiency. Experimental results show that AdapTracer reduces the trace size by 44% on average and incurs execution overhead by 10% at most compared to PAP.

The contributions of this paper are summarized as follows:

- We identify two significant problems using existing path profiling techniques.
- We propose a *space-efficient* and *adaptive* path profiling technique AdapTracer to reduce the trace size.
- We implement AdapTracer and use modified JGF benchmarks to show the effectiveness of our system.

The rest of this paper is structured as follows. Section II describes the related work. Section III shows two examples that motivate our work. Section IV gives an overview of

1521-9097/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPADS.2015.29

AdapTracer. Section V and Section VI present the details of the AdapTracer system. Section VII shows the evaluation results. Section VIII concludes this paper and gives future research directions.

II. Related Work

A. Path profiling

Path profiling gives useful information about the execution behavior of the program [9–12]. It attracts much research attention. In [5], each edge in a program's DAG is assigned with a weight. The PathID of an executed path is simply the sum of edge weights. The tracing overhead of a large program is expensive if we profile all paths. To reduce the tracing overhead, several approaches for profiling a subset of paths are proposed. TPP (Targeted Path Profiling) [13] eliminates unselected paths by assigning large negative weights to the edges that belong to the unselected paths but not belong to the selected paths. The selected paths are assigned with unique positive PathIDs and the unselected paths are assigned with non-unique negative PathIDs. The negative PathIDs will not be recorded and the tracing overhead is thus reduced. PPP (Practical Path Profiling) [14] extends TPP by combining the edge weights and eliminates the unneeded instrumentation. Profiling selected paths will miss the opportunity of finding the bugs residing in unselected paths. Therefore, profiling all paths is necessary for effective bug diagnosis. Different from profiling partial paths such as TPP and PPP, AdapTracer profiles all paths in a program. AdapTracer can effectively reduce the tracing overhead by assigning frequently executed paths with fewer bits.

In B.L.-like profiling approaches (e.g. TPP, PPP), multiple PathIDs are required for representing a cyclic path. A new PathID is added in the sequence once encountering a back-edge. PAP (Profiling All Path) [7] profiles the acyclic path and the cyclic path in a unified manner. A breakpoint (i.e., the CFG node and the PathID before overflow) is added in the sequence once the current PathID is going to overflow, reducing the tracing overhead compared to B.L.-like approaches. Different from the B.L. and PAP, our approach makes full use of each PathID for the representing of paths without extra recording overhead of CFG node.

B. Arithmetic Coding

Arithmetic coding is a well-known universal, lossless compression technique that achieves close-to-optimal compression rates [8, 15]. Like other compression mechanisms, arithmetic coding relies on the observation that in any given input stream, a fraction of symbols are likely to occur frequently. Arithmetic coding achieves compression by encoding these frequently occurring symbols using a smaller number of bits.

Suppose we have an alphabet N = {a, b, c, d}, and the corresponding probability model is {0.4, 0.2, 0.1, 0.3}.

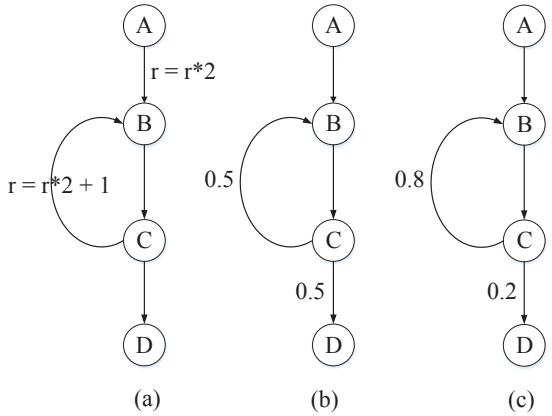

Figure 1: Examples of PAP and AdapTracer. (a) The instrumentation example of PAP. (b) Assigning equal probability to the edge CB and the edge CD. (c) The example of AdapTracer's edge probability model.

Now, we wish to send the message *daca*. The encoding and decoding procedures are shown below.

1) Encoding: Initially, both the encoder and the decoder know that the range is [0, 1). After seeing the first symbol *d*, the encoder narrows it down to [0.7, 1) (this is the range that the model allocates to symbol *d*). For the second symbol *a*, the interval is further narrowed, since *a* has been allocated to [0, 0.4). Thus the new interval is [0.7, 0.82). For the third symbol *c*, the new interval is [0.808, 0.82). For the last symbol *a*, the interval is [0.808, 0.8128). The final procedure is value selection, and a single number in the range can be chosen for the encoding result (0.809 in our example).

2) Decoding: In order to restore the sending message, we use the single number from the encoding. After knowing the single number 0.809, the decoder can immediately deduce that the first character was *d*. Now the decoder simulates the action of the encoder, and the range is expanded to [0.7, 1). In further processing, the decoder computes each subrange using the corresponding symbol probability. Next, the decoder can get subrange of [0.7, 0.82). So the decoder knows that the second character was *a*. In this way, the decoder can completely decode the transmitted message.

We note that it is easy for PAP to cause overflow because it encodes path using multiplication and addition. The close-to-optimal feature of arithmetic coding attracts us to apply it to path profiling, reducing the tracing overhead.

III. Motivating Examples

A. Benefit of using arithmetic coding

Fig. 1 (a) shows the CFG of a program where a node denotes a code block and a directed edge denotes an execution flow. There is a back-edge between code block B and C. Suppose that the execution path is ABCBCBCBCBCD.

With loss of generality, we assume a 3-bit value is used for one PathID.

PAP adds probes on multiple in-edges of a CFG node and uses multiplication and addition to calculate the PathID. For the example shown in Fig. 1 (a), it first initializes the probe value r to 0, and then changes the value of r according to the operation associated with each edge following the execution path.

- After the edge AB is executed, r = 0.
- After the edge BC is executed, the probe value r is unchanged.
- After the edge CB is executed, r = 1.
- After the subsequent edges are executed until the 3rd CB, r = 7.

Next, just before executing the 4th CB, the probe value will overflow if multiplication and addition are applied. PathID overflow will cause path decoding failures. To address this problem, PAP records the current probe value (i.e., 7) and the current CFG node (i.e., C). It reinitializes the probe to 0 and continues the above path encoding process.

Finally, PAP records the execution path as 7, C, 1 with 7 indicating the probe value before overflow, C indicating the CFG node before overflow, and 1 representing the probe value after overflow. If we use a 2-bit value to represent a CFG node (since there are 4 nodes in Fig. 1 (a)), the overall cost of PAP is $3 + 2 + 3 = 8$ bits.

In essence, PAP uses multiplication and addition to differentiate different in-edges of a CFG node. The path decoding process starts from the exit node (i.e., D). The previous node is iteratively inferred from the current probe value by division and modulo operations. Since the decoding is in reverse order, a breakpoint (containing the CFG node and probe value before overflow) is required to infer the executed path before overflow. Otherwise, the decoding process would have no idea where to start for decoding the path before overflow.

We note that two problems cause large recording overhead in PAP. First, path encoding using multiplication and addition will easily cause overflow. Second, the CFG node in the breakpoint causes extra overhead. Different from PAP, AdapTracer adds probes on multiple out-edges of a CFG node and uses arithmetic coding to address the above two problems. Arithmetic coding can achieve more compact path encoding than PAP. In addition, path decoding starts from the start node with AdapTracer. Hence, the CFG node in the breakpoint can be implicitly inferred from the probe value before overflow, i.e., AdapTracer can effectively eliminate overhead of CFG node in the breakpoints.

We will show in Section V that the recoding overhead of AdapTracer is 3 bits for the execution path ABCBCBCBCBCD, a reduction of 5 bits compared with PAP.

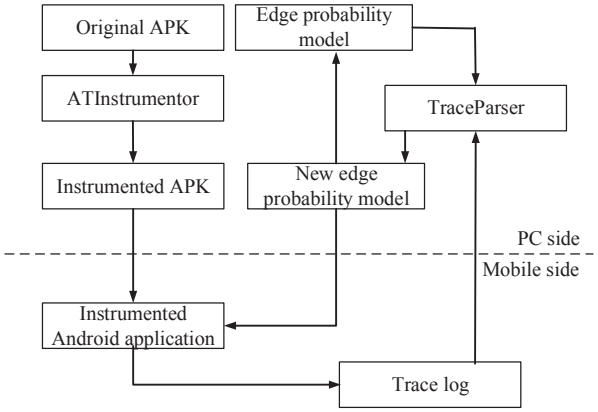

Figure 2: Overview of AdapTracer system

B. Benefit of adaptive coding

When applying arithmetic coding to path profiling, a native approach is to assign equal probabilities to multiple out-edges of a CFG node since it is possible to execute each edge. For the example shown in Fig. 1 (b), we assign equal probabilities to the two out-edges of node C, i.e., 0.5. For the execution path ABCBCBCBCBCD, the tracing overhead is 6 bits (the same as in the previous subsection).

We note that the performance of arithmetic coding highly depends on the probability model which assigns a probability to each of the various symbols [8]. These probabilities correspond to the edge probabilities for our path profiling problem. The assignment of equal probabilities leads to poor performance since it loses the opportunity to reduce the overhead for frequently executed paths. For example, if the path ABCBCBCBCBCD is frequently executed, it is beneficial to reduce the tracing overhead for this path so that the expected tracing overhead can be significantly reduced (in other words, the overall cost for tracing multiple executions can be significantly reduced).

A natural improvement is to assign high probability to frequently executed edge. For the example shown in Fig. 1 (c), we assign a high probability 0.8 to the frequently executed edge CB and a low probability 0.2 to the infrequently executed edge CD, the tracing overhead can be reduced to 3 bits for the execution path ABCBCBCBCBCD. AdapTracer records the edge execution frequencies during multiple program executions and uses this information to *adaptively* change edge probabilities for arithmetic coding.

It is worth noting that both B.L. and PAP are *nonadaptive* since they both assign a fixed rule upon executing an edge.

IV. Overview of AdapTracer

Fig. 2 shows an overview of AdapTracer. We have implemented AdapTracer to profile Android applications. There

are two tools in AdapTracer: ATInstrumentor and TraceParser. Section V describes how ATInstrumentor instruments the APK. Section VI presents the details of how TraceParser generates the new edge probability according to the restored paths.

At the PC side, ATInstrumentor analyses the original APK and generates the instrumented APK. Then, the instrumented APK is pushed to mobile side. At the mobile side, the instrumented APK is installed on Android system. The instrumented Android application runs on Android system for some time and produces the trace log. After that, the trace log is pulled from the mobile. TraceParser restores the paths from the trace log and the edge probability model. The new edge probability model is generated according to the restored paths. Finally, there are two copies of the new edge probability model. One is pushed to the mobile to replace the instrumented Android application's edge probability model file. Another is used to replace the PC side edge probability model for next path restoring.

V. The Design of ATInstrumentor

ATInstrumentor includes four functions: decompiling APK and smali files analysis, generating instrumentation model, instrumenting the smali files, recompiling the smali files to APK.

A. Decompiling APK and analyzing smali files

Step 1: decompiling APK. For the convenience of analyzing the CFG model of Android application, we first use *apktool* [16] to transform the APK into smali files. Each smali file denotes a class. There are several fields saving the information about the class. For example, `Head Field` saves information of the class name, the super class name and the corresponding java source file name. `Method Field` saves information of the method name, the input parameters' type, the return parameters' type and the smali code of this method. The construction of Android application's CFG model is mainly based on the analysis of the `Method Field`.

Step 2: extracting code block information. In this step, we analyse every method of smali files and extract the code block information according to the smali syntax [17].

Fig. 3 shows a typical smali code of method field. In line 1, it declares the start of method field. The method's declaration consists of method name, input parameters type, return parameters type. In this example, method name is `IFSense`, input parameter is null and the return parameter is `Z` which means boolean type. In line 2, it declares the number of register used in this method. Smali is a register-based language and all operations are on registers. In order to do instrumenting, we increase the number of register. In line 3, the label `.prologue` is a keyword that denotes the start of method content. In line 9, there is another keyword (i.e., `if-eqz`) in this instruction. This instruction means

```
1  .method private ifSense()Z   #method start label
2      .locals 2                #number of register
3      .prologue                #method content start label
4
5      .line 22                 #line number in jave source
6      const/4 v0, 0x1
7      .line 24
8      .local v0, tempFlag:Z
9      if-eqz v0, :cond_0
10
11     .line 25
12     const/4 v1, 0x1
13
14     .line 27
15     :goto_0                  #jump label
16     return v1
17
18     :cond_0                   #UpLine
19     const/4 v1, 0x0
20     goto :goto_0             #DownLine
21 .end method                  #method end label
```

Figure 3: Typical smali code of method field

that if the register v0 equals to zero then the execution flow jumps to the address labeled with `:cond_0`. Otherwise the execution flow moves to next line. Note that the jump among different executions can be changed by moving the position of the jump label (i.e., `:cond_0`). It is benefit for our instrumenting work without calculating the offset addresses.

According to the smali syntax mentioned above, we can split the smali codes into different code blocks using the keywords and the jump labels. For the example shown in Fig. 3, the label `.prologue` is a keyword that denotes the start of method content. From line 2 to line 3 is the first code block. There are three kinds of code block information to be recorded. First, the code block id that differentiates different code blocks. In this example, the code block id is assigned with zero. Second, the jump label in code block start line (UpLine for short) and the keyword in code block end line (DownLine for short). The jump label of the code block is analyzed from the UpLine and the code blocks keyword operation is analyzed from the DownLine. If there is no keyword or jump label in the start line or end line of the code blocks, the other instructions are recorded. For the code block zero, the UpLine is recorded as `.locals 2` since there is no jump label in UpLine. The DownLine is `.prologue`. Third, the lines of UpLine and DownLine in the smali file. The lines are used for instrumentation. For the code block zero, the line of UpLine is 2 and the line of DownLine is 3. After that, we continue splitting the code block from line 4. The rest code blocks' information are shown in Table I. Note that the smali code `.line x` (where x denotes the line number) is not recorded as an instruction since it has no operation.

Step 3: constructing CFG model. There are 3 steps for

Table I: Code block information

blockid	UpLine (line)	DownLine (line)
0	.locals 2 (2)	.prologue (3)
1	const/4 v0, 0x1 (6)	if-eqz v0, :cond_0 (9)
2	const/4 v1, 0x1 (12)	const/4 v1, 0x1 (12)
3	:goto_0 (15)	return v1 (16)
4	:cond_0 (18)	goto :goto_0 (20)

constructing CFG model. First, code blocks are modeled as CFG nodes. Then, we construct the execution flows among different code blocks according the UpLine and the DownLine. As shown in Table I, there is an execution flow from code block 1 to code block 4. Finally, execution flows among different code blocks are modeled as edges in CFG.

Step 4: classifying main functions and sub functions. There are four main components in Android system. Each component responses to user's interactions using system default methods called Android life-cycle method [18]. For example, when user starts an Android application, the Android Activity life-cycle method *OnCreate()* is executed. Developers overwrite the Android life-cycle methods to response to user's interactions. Thus, the overwritten Android life-cycle methods (the AndroidLifeMethod for short) can be seen as main functions and all other developers written methods (the DevpMethod for short) invoked by the AndroidLifeMethod can be seen as sub functions. In this way, the AndroidLifeMethod and the related DevpMethod can be profiled together.

B. Generating instrumentation model

There are two kinds of models to be instrumented in Android application: the AdapTracer encoder model and the edge probability model. The AdapTracer encoder model and the edge probability model are transformed into smali codes such that they can be invoked by Android application directly.

Edge probability model. The edge probability model is a list that has 5 elements: the method name combined with the class name (i.e., MainActivity_IFSense()Z), the edge's start code block id, the edge's end code block id, the edge's probability and the edge's counter. The edge's counter (initialized to one usually) is used to record the edge's execution frequency. AdapTracer encoder model updates the edge's execution frequency using the edge's counter. We assign equal probability to multiple out-edges. $\forall e \in outedge(n)$, $p(e) = \frac{1}{|outedge(n)|}$. n is a CFG node that has multiple out-edges. $|outedge(n)|$ means the number of the multiple out-edges of the CFG node n.

AdapTracer encoder model. There are five methods in AdapTracer encoder model: *Initial()*, *SetStartCodeBlock()*, *ATencoder()*, *ValueSelect()*, *OverFlow()*. The main method is *ATencoder()* that encodes each executed edge. *Initial()* initializes the interval variables and declares the current executed method's name. *SetStartCodeBlock()* sets the edge's start code block id which can be used to construct the executed edge for *ATencoder()*. *ValueSelect()* selects the minimal value within the interval to denote the executed path. *OverFlow()* stores and resets the PathID which is going to overflow.

Algorithm 1 The AdapTracer encoding algorithm

```
1: function ATENCODER(Edge E)
2:     CodeBlock n = E.getStartCodeBlock()
3:     AMlib.Encoder(E)
4:     if E.getCounter() + _inc > _bound then
5:         n.ShrinkCounter()
6:     end if
7:     E.setCounter(E.getCounter() + _inc)
8:     n.updateOutedgesProbability()
9: end function
```

Algorithm 1 presents the procedure of AdapTracer encoding algorithm. In line 1, the input parameter *E* is a class type. *E* finds the edge related elements from the edge probability model. *getStartCodeBlock()* gets the edge's start code block. *setCounter()* and *getCounter()* are used to assign and get the edge's counter respectively. In line 2, the local variable *n* is a CodeBlock class type. In line 3, AMlib is the arithmetic coding library implemented in integer [8]. *Encoder()* encodes the parameter *E* to a new sub-interval according to *E*'s probability. In line 4, $\_inc$ and $\_bound$ are static variables. $\_inc$ is related to the speed of matching the edge execution probability. Section VII shows the relationship between $\_inc$ and trace size. $\_bound$ limits the edge counter's value to avoid overflow. When the counter's value is going to overflow, the method *ShrinkCounter()* is called to diminish the value of multiple out-edges' counter of the code block *n*. In line 7, the edge *E*'s counter is updated. In line 8, multiple out-edges' probability of the code block *n* are updated.

ValueSelect() is almost the same as the implementation in arithmetic coding library AMlib. In *Encoder()* and *ValueSelect()*, the PathID may overflow when we profile large program. *OverFlow()* is instrumented in the instruction of interval scaling in *Encoder()* and *ValueSelect()* [8]. Once the PathID is going to overflow, current PathID is stored in list and reset to zero for next encoding. *Initial()* and *SetStartCodeBlock()* are intuitive so we don't show here.

C. Instrumenting the smali files

Algorithm 2 shows the basic idea of AdapTracer instrumentation. The instrumentation algorithm for AndroidLifeMethod and DevpMethod is similar. The difference is that there is no instrumentation of line 3 and line 14 for the DevpMethod.

In line 1, the input parameter F is a CFG class type. *getEntry()* and *getExit()* get the start code block and the exit code block of current method. *getCodeBlockList()* gets the list of

Algorithm 2 The AdapTracer instrumentation algorithm

```
1: function INSTRU(CFG F)
2:     CodeBlock entry = F.getEntry()
3:     instrumentCodeBlock(entry,"Initial")
4:     for CodeBlock n in F.getCodeBlockList() do
5:         int s = n.outEdgeCount()
6:         if s > 1 then
7:             for Edge e in n.getOutEdgeList() do
8:                 instrumentEdge(e.getStartCodeBlock(),
   "SetStartCodeBlock")
9:                 instrumentEdge(e.getEndCodeBlock(),
   "ATencoder")
10:            end for
11:        end if
12:    end for
13:    CodeBlock exit = F.getExit()
14:    instrumentCodeBlock(exit, "ValueSelect")
15: end function
```

all code blocks of current method. From line 2 to line 3, *instrumentCodeBlock()* instruments the smali instruction of invoking *Initial()* at the address of entry's DownLine. From line 4 to line 12, we instrument the *SetStartCodeBlock()* and *ATencoder()* at multiple out-edges of each code block. In line 8, the smali instruction of invoking *SetStartCodeBlock()* is instrumented at the address of start code block's DownLine. In line 9, the smali instruction of invoking *ATencoder()* is instrumented at the address of end code block's UpLine. In line 14, the smali instruction of invoking *ValueSelect()* is instrumented at the address of exit's DownLine.

Specifically, for the example shown in Section III. The interval is first initialized to [0, 1) and the PathID is set to 0. Each edge is assigned with equal probability (i.e., 0.5). After the edge CB is executed, CB's counter is added with one. Thus, the new CB's probability is 0.67 and the new CD's probability is 0.33. Finally, the value 0.1875 (0.00011_2) within [0.1621, 0.1953) is selected as the PathID. The final probability of CB and CD are 0.75 and 0.25 respectively. In the first execution, AdapTracer still uses fewer bits than PAP (i.e., $8 - 6 = 2$ bits) since AdapTracer doesnt record the code block id.

In the second execution of the same path, the interval and the PathID are initialized the same as the first execution. The edge probability model is obtained from the first execution's results. Finally, we select the value 0.375 (0.011_2) within [0.3321, 0.3925) to denote the same path. Owing to the benefit of *adaptive* coding, the tracing overhead is significantly reduced in AdapTracer (i.e., $8 - 3 = 5$ bits).

D. Recompiling the smali files to APK

After all smali files are instrumented, we use *apktool* to recompile the smali files to APK. Then, we can push and install the instrumented APK on mobile phone using *adb* [19].

Algorithm 3 The AdapTracer decoding algorithm

Output: corresponding path

```
1: function ATDECODER(PathId* PI, CodeBlock S,
   CodeBlock E)
2:     /*p is used to record the path*/
3:     List<CodeBlock> p = new List<CodeBlock>()
4:     CodeBlock cur = S
5:     Edge eg
6:     while cur != E do
7:         p.append(cur)
8:         if cur.isInvokeDevpMethod() then
9:             CFG f = cur.getDMCFG()
10:            p.append(ATdecode(PI, f.entry, f.exit))
11:        end if
12:        if cur.outEdgeCount() > 1 then
13:            eg = AMlib.Decoder(PI, cur)
14:            eg.setCounter(eg.getCounter() + _inc)
15:            cur.updateOutedgesProbability()
16:            cur = eg.getEndCodeBlock()
17:        else
18:            cur = cur.getOutEdgeList()[0].getEndCodeBlock()
19:        end if
20:    end while
21:    p.append(E)
22:    return p
23: end function
```

VI. THE DESIGN OF TRACEPARSER

TraceParser restores the paths from the trace log and the edge probability model. In the first execution, the edge probability model assigns equal probability to each edge since we have no idea of the edge's execution probability. Then the new edge probability model is generated according to the restored paths. Two copies of the new edge probability model are generated. One is pushed to the mobile phone to replace the instrumented Android application's edge probability model. Another is used to replace the PC side edge probability model for the next path restoring.

A. Analyzing trace log

After using the instrumented Android application for some time, the trace log is produced. The file format of the trace log is a list and the tuples of (the method name combined with the class name, PathIDlist) are stored in the trace log. The CFG information is obtained from the ATInstrumentor. We can find the method's CFG information according to the the method name combined with the class name.

B. Restoring the paths

After we finding the corresponding method according to the the method name combined with the class name, the

method's CFG and the PathIDlist are used in ATdecode to restore the execution path as shown in Algorithm 3.

In line 1, *PI* is a pointer parameter that points to address of the PathID. CodeBlock *S* and CodeBlock *E* are the start code block and the exit code block in current method. From line 6 to line 20, *ATdecoder()* restores the path of current AndroidLifeMethod and all related DevpMethod together. In line 8, *isInvokeDevpMethod()* checks whether current code block's DownLine invokes the DevpMethod. If so, *ATdecoder()* jumps to restore the DevpMethod. The restored DevpMethod path is appended in *p*. In line 13, *Decoder()* of the arithmetic coding library AMlib is used to decode the PathID. From line 14 to line 15, the multiple out-edges' probabilities are updated same as *ATencoder()*. Then, in line 16, *ATencoder()* moves to the next code block to continue the decoding procedure. Finally, the decoding procedure is finished when it encounters the exit code block. The restored path is saved in *p* and returned.

C. Calculating the edge probability model

According to the restored paths, the edge probability model can be calculated easily. Two copies of the new edge probability model are generated. One is pushed to the path of the instrumented Android application. Another is saved at the PC side for the next path restoring.

VII. Evaluation

The experiment is running on Google Nexus 4, which is equipped with the 1.5GHz CPU and the 2GB RAM. A part of Java Grande Benchmarks (JGF) [20] are transformed into Android applications for our experimental tests. Different kinds of functions and CFG structures are contained in different JGF benchmarks. For example, JGFLoop has lots of loops and JGFMath has different mathematical operations. The construction of CFG is based on the analysis of smali codes which can be obtained through *apktool* [16]. The user's interactions with mobile phone are simulated by recording a sequence of specific interactive events and replay on mobile phone.

A. The relationship between the _inc and the trace size

Fig. 4 shows the relationship between the $\_inc$ and the trace size. The horizontal axis refers to the repeat times of interactive events. We choose the JGFMath application which is the most complex application of JGF in this experiment. According to our analysis of smali codes, JGFMath has 963 code blocks, 120 branches and 30 loops. Each loop has different mathematical operations, such as absolution, maximize, minimize, logarithm and etc. We set the $\_inc$ to four different values from 1 to 4, which are labeled with 1-step to 4-step in Fig. 4. From the results, we can find that the change of trace size is the same as the 3-step. When $\_inc$ is bigger, the more times the counter would need to be narrowed. So the 3-step is used in our

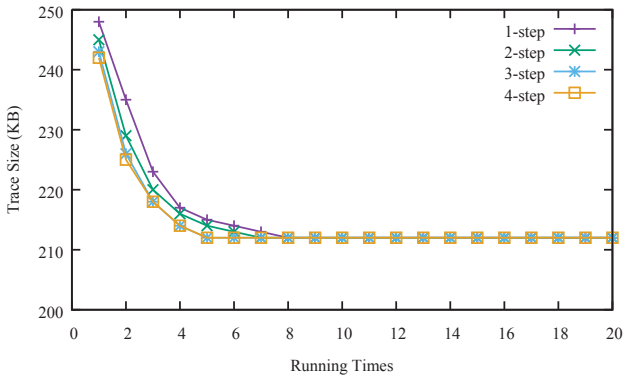

Figure 4: The relationship between the $\_inc$ and the trace size

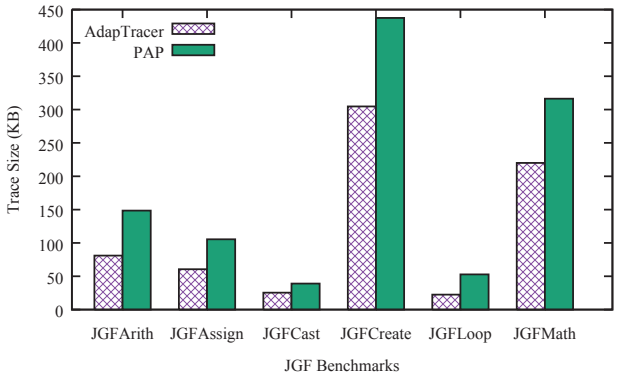

Figure 5: Space cost in JGF benchmarks

AdapTracer. The experimental results also show that the trace size produced by AdapTracer becomes small following the program's execution. In PAP, the trace size is still large no matter how many times the program executes.

B. Complete tests using modified JGF benchmarks

Fig. 5 - Fig. 7 present the complete experimental results using JGF benchmarks. Fig. 5 shows the different trace size

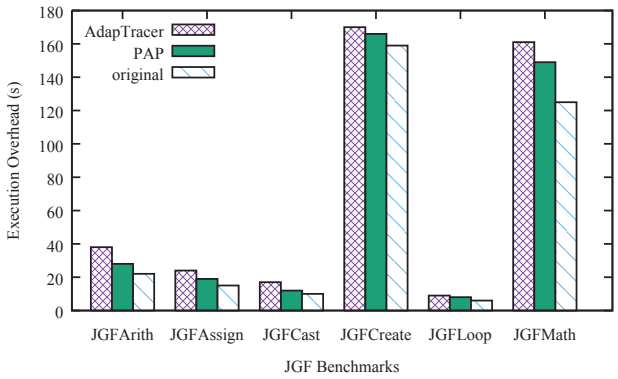

Figure 6: Execution overhead in JGF benchmarks

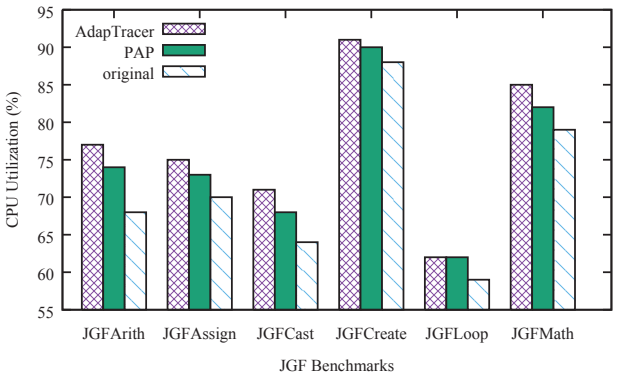

Figure 7: CPU utilization in JGF benchmarks

produced in AdapTracer and PAP while profiling the JGF benchmarks. The AdapTracer reduces the trace size by 44% on average compared with PAP.

Fig. 6 and Fig. 7 present the different execution overhead and CPU utilization among AdapTracer, PAP and the original Android applications. The AdapTracer incurs execution overhead by 10% at most compared to PAP.

VIII. Conclusion

This paper presents AdapTracer, a path profiling approach based on arithmetic coding. There are two salient features in AdapTracer. First, it is *space efficient* by adopting a path profiling algorithm based on arithmetic coding. Second, it is *adaptive* by explicitly considering the execution frequency of each edge. Experimental results show that AdapTracer reduces the trace size by 44% on average and incurs execution overhead by 10% at most compared to PAP.

AdapTracer can reduce the tracing overhead of frequently long execution path significantly, but the actual space usage may be the same as the PAP in profiling frequently short execution path. Namely, for frequently short execution path, the PathID encoded in PAP may not overflow. No matter how few bits AdapTracer produces, the actual space usage is same with PAP. Therefore, our future work will focus on designing an optimal algorithm to reduce the tracing overhead of profiling the frequently short execution path.

Acknowledgment

This work is supported by the National Science Foundation of China under Grant No. 61472360, Zhejiang Provincial Platform of IoT Technology under Grant No. 2013E60005, and Zhejiang Commonwealth Project under Grant No. 2015C33077.

References

[1] M. Tancreti, V. Sundaram, S. Bagchi, and P. Eugster, "TARDIS: Software-only System-level Record and Replay in Wireless Sensor Networks," in *Proc. of ACM/IEEE IPSN*, 2015.

[2] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating Mobile Application Energy Consumption Using Program Analysis," in *Proc. of ACM/IEEE ICSE*, 2013.

[3] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "SIF: A Selective Instrumentation Framework for Mobile Applications," in *Proc. of ACM MobiSys*, 2013.

[4] Y. Liu, C. Xu, and S. C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," in *Proc. of ACM/IEEE ICSE*, 2014.

[5] T. Ball and J. R. Larus, "Efficient Path Profiling," in *Proc. of ACM MICRO*, 1996.

[6] J. R. Larus, "Whole Program Paths," in *Proc. of ACM/IEEE PLDI*, 1999.

[7] B. Li, L. Wang, H. Leung, and F. Liu, "Profiling All Paths: A New Profiling Technique for Both Cyclic And Acyclic Paths," *Journal of Systems & Software*, pp. 1558–1576, 2012.

[8] A. Said, "Introduction to Arithmetic Coding Theory and Practice," Hewlett-Packard Laboratories Report, Tech. Rep. HPLC2004C76, April 2004.

[9] D. G. Melski, "Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation," Tech. Rep., 2002.

[10] B. Kasikci, T. Ball, G. Candea, J. Erickson, and M. Musuvathi, "Efficient Tracing of Cold Code via Bias-Free Sampling," in *USENIX ATC*, 2014.

[11] T. Apiwattanapong and M. J. Harrold, "Selective Path Profiling," in *Proc. of ACM PASTE*, 2002.

[12] M. Arnold and B. G. Ryder, "A Framework for Reducing the Cost of Instrumented Code," in *Proc. of ACM PLDI*, 2001.

[13] R. Joshi, M. D. Bond, and C. Zilles, "Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems," in *Proc. of ACM/IEEE CGO*, 2004.

[14] M. D. Bond and K. S. McKinley, "Practical Path Profiling for Dynamic Optimizers," in *Proc. of ACM/IEEE CGO*, 2005.

[15] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2012.

[16] "http://ibotpeaches.github.io/apktool/."

[17] "https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html."

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proc. of ACM PLDI*, 2014.

[19] "http://developer.android.com/tools/help/adb.html."

[20] "http://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking."