Journal of Systems Architecture 88 (2018) 74–86

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

AdapTracer:Adaptive path profiling using arithmetic coding

Gonglong Chen*, Wei Dong

College of Computer Science, Zhejiang Universit, Hangzhou, China

ARTICLE INFO

Keywords:
Path profiling
Arithmetic coding
Adaptive tracing

ABSTRACT

Path profiling, which aims to trace the execution path of programs, 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 the BLPP (Ball-Larus Path Profiling) algorithm, and PAP (Profiling All Path). 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.

1. 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 BLPP 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 BLPP 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 BLPP 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 corresponding 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 BLPP 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 reverse decoding. Second, it is *adaptive* by explicitly considering the execution frequency of each edge, which is recorded by the edge counters. With the help of edge counter, AdapTracer adjusts each edge's probability to achieve a close-to-optimal path encoding.

We have implemented AdapTracer to profile Android applications. Compared with our previous version [9], we have improved the experimental evaluation a lot by including more programs from the JGF suite and a large set of medium (e.g., Google Play) to big Android

* Corresponding author.
E-mail addresses: chengl@emnets.org, desword@zju.edu.cn (G. Chen), dongw@emnets.org (W. Dong).

https://doi.org/10.1016/j.sysarc.2018.05.011
Received 19 August 2017; Received in revised form 30 March 2018; Accepted 30 May 2018
Available online 31 May 2018

programs. Related work has been extended by a quantitative analysis of existing path profiling algorithms. Moreover, a new chapter with an analysis of common bugs using AdapTracer has been added. 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 2 describes the related work. Section 3 shows two examples that motivate our work. Section 4 presents the encoding and decoding algorithm of AdapTracer. Section 5 and Section 6 present the details of the AdapTracer system. Section 7 shows the evaluation results. Section 8 presents three case studies to illustrate the usage of our approach. Section 9 concludes this paper and gives future research directions.

2. Related work

2.1. Path profiling

Path profiling gives useful information about the execution behavior of the programs [10–13]. 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 the sum of edge weights. To reduce the tracing overhead, several approaches for profiling a subset of paths are proposed. TPP (Targeted Path Profiling) [14] 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.

To further reduce the tracing overhead that is caused by expensive hash operations, Preferential-PP (Preferential-Path Profiling) [15] designs a compact path numbering algorithm for interesting paths. The tracing overhead is further reduced by saving the execution frequency of the interesting paths in an array. Pertinent-PP (Pertinent-Path Profiling) [16] provides a more user-friendly interface for developers to determine interesting paths. Developers only need to specify a set of interested key operations (e.g., read(), write()), Pertinent-PP would then only track the paths that are relevant to the set of key operations. It enables an on-demand path profiling and reduces the tracing overhead. However, profiling selected paths may 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, AdapTracer profiles all paths in a program. AdapTracer can effectively reduce the expectation tracing overhead by assigning frequently executed paths with fewer bits.

Practical-PP (Practical Path Profiling) [17] extends TPP by simplifying path profiling using an edge profile. The amount of instrumentations on cold paths and paths that the edge profile predicts well is reduced. Similar with Practical-PP, PEP (Path and Edge Profiling) [18] also lowers instrumentation overhead using the edge profile collected so far. However, PEP incorporates a thread-switching mechanism that is common to Java virtual machines to further reduce the runtime overhead. P3 (Partitioned Path Profiling) [19] reduces the runtime overhead by running K copies of the program in parallel, each with the same input but on a separate core. P3 collects the profile only for a subset of intra-procedural paths in each copy. Different from above works that aim to reduce the runtime overhead, AdapTracer aims to reduce the expectation tracing overhead. Therefore, AdapTracer is orthogonal with above works.

In BLPP-like profiling approaches (e.g. WPP[6], TPP[14], Practical-PP[17] and k-BLPP[20]), multiple PathIDs are required for representing a cyclic path. A new PathID is added in the sequence once encountering a back-edge. HPP (Hierarchical Program Paths)[21] is a BLPP based path profiling and querying approach. By assigning inter-procedure paths with unique PathIDs, HPP can query any interested paths. Although HPP reduces path encoding overhead by eliminating redundant PathIDs at the call sites comparing with BLPP, it still statically encodes paths without considering execution frequency of paths. Different from HPP, AdapTracer adaptively encodes paths using arithmetic code. Frequently executed paths are assigned with less trace size. Therefore, AdapTracer improves the space efficiency. Roman et al. [22] proposed an program flow tracing method based on the BLPP algorithm. It relies on external devices to record traces, which is not applicable to our scenarios that the applications are distributed to users. In this case, we need to minimize the trace space to ensure the usability on the user side. PAP (Profiling All Path) [7] profiles acyclic and cyclic paths in a unified manner, reducing the overhead of using multiple PathIDs compared to BLPP-like approaches. 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. Different from BLPP and PAP, our approach makes full use of each PathID for the representation of paths without extra recording overhead of CFG nodes.

Comparing with the commercial software RapiTime [23], AdapTracer has the following two advantages. First, AdapTracer traces the program flow in a fine-grained level (i.e., code block level), while the tracing of RapiTime is in the function call level. AdapTracer provides more program execution information. It improves the probability of finding worst-case execution comparing with other approaches that provide course-grained (e.g., function level) information. In this way, the time of finding bugs is saved. Second, AdapTracer traces the control flow of the program, while RapiTime only records the start and end time of functions. Therefore, AdapTracer can be used not only for execution time analysis, but also for analyzing control flow bugs and security issues (e.g., as we detailed in the case study Section 8).

Table 1 compares various approaches with respect to three key desired features:

- *Adaptive.* This is important as to reduce the expectation overhead.
- *Space efficiency.* This is an important feature for the profiling approach to be applies on resource-constraint devices.
- *Whole path profiling.* It means that the corresponding algorithms profile all of the paths in a program, comparing with algorithms that only profile partial paths. Effectively profiling all of the paths brings more benefits than only profiling partial or selected paths.

Table 1
Comparison of existing approaches.

Approach	Adaptive	Space efficiency	Whole path profiling
TPP	√	√	×
Preferential-PP	×	√	×
Pertinent-PP	×	√	×
Practical-PP	×	√	×
PEP	√	√	×
P3	×	√	×
WPP	×	×	√
k-BLPP	×	×	√
PAP	×	×	√
RapiTime	×	√	×
Roma	×	×	√
HPP	×	×	√
AdapTracer	√	√	√

2.2. Arithmetic coding

Arithmetic coding is a well-known universal, lossless compression technique that achieves close-to-optimal compression rates [8,24]. Like other compression mechanisms, arithmetic coding relies on the observation that in any given input stream, only 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}. Now, we wish to send the message *daca*. The encoding and decoding procedures are shown below.

2.2.1. 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).

2.2.2. 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 the multiplication and the addition. The close-to-optimal feature of arithmetic coding attracts us to apply it to path profiling, reducing the tracing overhead.

3. Motivating examples

3.1. 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. Without loss of generality, we assume a 3-bit value is used for one PathID.

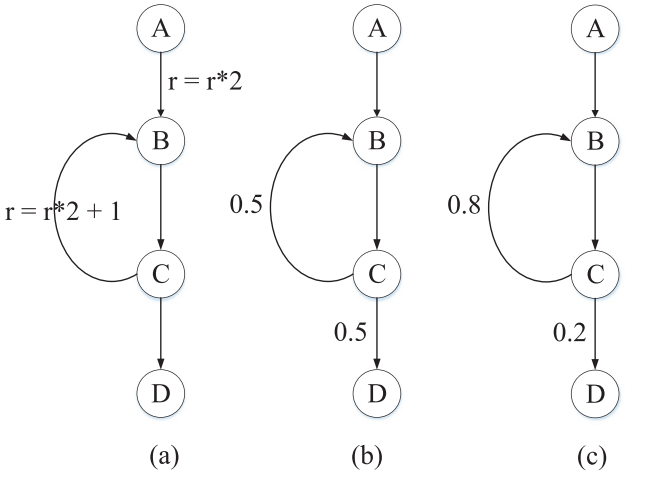

Fig. 1. Examples of PAP and AdapTracer. (a) The instrumentation example of PAP. (b) Assigning equal probability to the edge CB and the edge CD. (c) The example of AdapTracer's edge probability model.

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. The entire procedure is shown below:

- 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 re-initializes 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 a more compact path encoding than PAP. In addition, path decoding starts from the start node in AdapTracer. Hence, the CFG node in the breakpoint can be implicitly inferred from the probe value before overflow, i.e., AdapTracer can effectively eliminate the overhead of CFG node in the breakpoints.

We will show in Section 4 that the recoding overhead of AdapTracer is 3 bits for the execution path ABCBCBCBCBCD, a reduction of 5 bits compared with PAP.

3.2. Benefit of adaptive coding

When applying arithmetic coding to path profiling, a naive 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

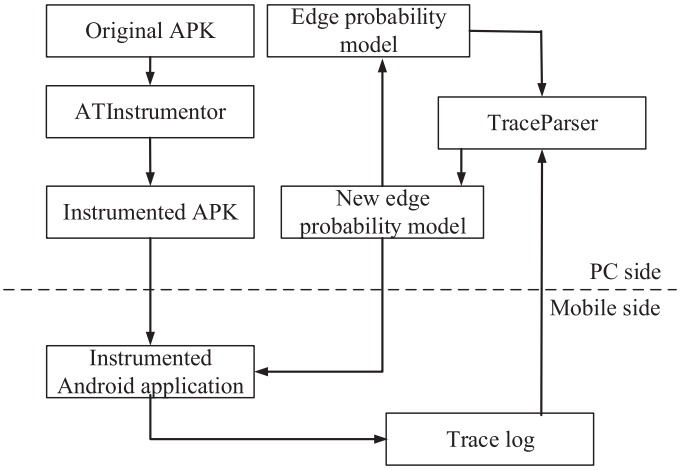

Fig. 2. Overview of AdapTracer system.

multiple executions can be significantly reduced).

A natural improvement is to assign high probabilities to frequently executed edges. 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 BLPP and PAP are *nonadaptive* since they both assign a fixed rule upon executing an edge.

4. The overview and algorithms 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 5 describes how ATInstrumentor instruments the APK (Android Package). Section 6 presents the details of how TraceParser generates the new edge probabilities 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 the mobile phone. At the mobile side, the instrumented APK is installed on the Android system. The instrumented Android application runs on the Android system for some time and produces the trace log. After that, the trace log is pulled from the mobile phone. TraceParser restores the paths from the trace log based on the edge probability model. The new edge probability model can then be 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 edge probability model at the PC side for next path restoring.

In the following subsections, we will present the details of AdapTracer encoding and decoding algorithms, which are the core components for ATInstrumentor and TraceParser respectively.

4.1. The AdapTracer encoding algorithm

Algorithm 1 presents the procedure of the 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 counter of the edge 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()*

```
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. The AdapTracer encoding algorithm.

encodes the parameter E to a new sub-interval according to E's probability. In line 4, *inc* and *bound* are static variables. The parameter *inc* is related to the speed of updating the edge execution probability. Section 7 shows how *inc* impacts on the trace size. The parameter *bound* limits the edge counter to avoid overflow. When the counter 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.

Specifically, as shown in Section 3, 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 doesn't record the code block id (e.g., CFG node id).

In the second execution of the same path, the interval and the PathID are initialized as in 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 by AdapTracer (i.e., $8 - 3 = 5$ bits).

4.2. The AdapTracer decoding algorithm

Algorithm 2 shows the details of the AdapTracer encoding algorithm. In line 1, *PI* is a pointer parameter that points to the address of the PathID. CodeBlock *S* and CodeBlock *E* are the start code block and the exit code block in the current method. From line 6 to line 20, *ATdecoder()* restores the path of the current AndroidLifeMethod (i.e., the Android life-cycle methods) and all related DevpMethod (i.e., the

```
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
```

Algorithm 2. The AdapTracer decoding algorithm.

methods written by developers) together. The details of AndroidLifeMethod and DevpMethod can be found in Section 5. In line 8, *isInvokeDevpMethod()* checks whether there is an invocation for 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 probabilities of multiple out-edges are updated similarly 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.

5. The design of ATInstrumentor

As shown in Section 4, there are two tools in AdapTracer: ATInstrumentor and TraceParser. ATInstrumentor includes four functions: decompiling APK (Android Package) and smali files analysis, generating instrumentation model, instrumenting the smali files, recompiling the smali files to APK.

Because we implement AdapTracer on the Android system, we first introduce necessary background of the Android system to better understand our approach.

5.1. The Background of Android OS

Android component and life cycle. There are four types of components in an Android application: *activity, service, broadcast receiver* and *content provider*. Each component is required to follow a life cycle that defines how this component is created, used and destroyed. For example, Fig. 3 shows the life cycle of an activity. This activity starts with calling three callback functions (e.g., onCreate(), onStart() and onResume()) and changes the state into "Running" (e.g., activity's foreground lifetime). When we press the "Back" button on the phone, the Android application is shutdown and the activity's state is finally changed into "Destroyed" by calling callback functions onPaused(), onStop() and onDestroy() accordingly. When we press the "Home" button, the Android application goes into the background and the activity calls onPaused() and onStop() to change the state into "Stop". If we return to the Android application, the activity is resumed and three callback functions are called as shown in Fig. 3. In exception cases, a stopped or paused activity may be killed for releasing memory.

Smali syntax. Smali is a kind of IL (Intermediate Language) that is decompiled from the Android executable code (e.g., Dalvik machine code) [25]. Fig. 4 shows a typical smali code of the method field. In line 1, it declares the start of the method field. The method's declaration consists of method name, input parameters type and 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 registers used in this method. Smali is a register-based language and all operations are on registers. In order to add instrumentation, we increase the number of registers. In line 3, the label `.prologue` is a keyword that denotes the start of the method content. In line 9, there is another keyword (i.e., `if-eqz`) in this instruction. This instruction means 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 the 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 beneficial for our instrumenting work without calculating the offset addresses.

5.2. Smali files analysis

Step 1: APK decompilation. For the convenience of analyzing the CFG model of an Android application, we first use *apktool* [26] to decompile the APK into smali files. Each smali file denotes a class. There are several fields storing the information about the class. For example, `Head Field` saves the class name, the super class name and the corresponding Java source file name. `Method Field` saves the method name, the input parameters type, the return parameters type and the smali code of this method. The construction of the Android application's CFG model is mainly based on the analysis of the `Method Field`.

Step 2: code block information extraction. In this step, we

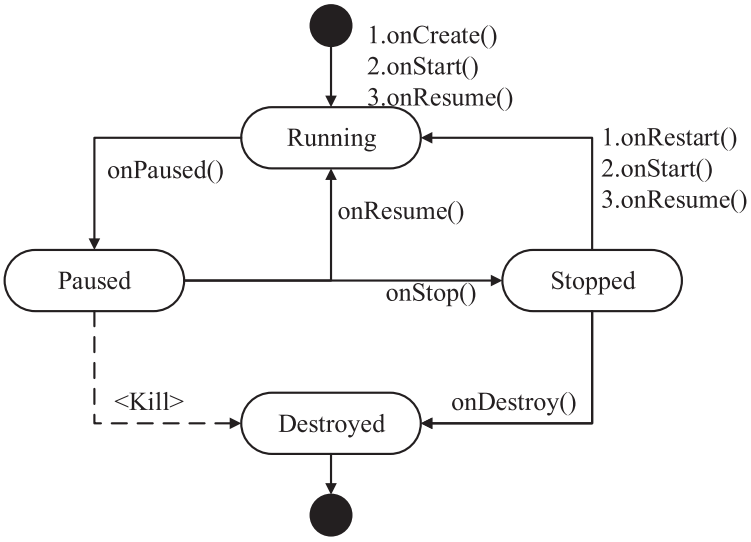

Fig. 3. The life cycle of an Android activity.

analyse every method of smali files and extract the code block information according to the smali syntax [25].

According to the smali syntax mentioned in Section 5.1, we split the smali codes into different code blocks based on the keywords and the jump labels. For the example shown in Fig. 4, the label `.prologue` is a keyword that denotes the start of the method content. From line 2 to line 3 is the first code block. There are three kinds of 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 record the information about the instrumentation position. 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 2. Note that the smali code `.line x` (where x denotes the line number) is not recorded as an instruction since it does not perform any operation.

Step 3: CFG model construction. There are 3 steps for constructing a 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 2, there is an

```
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
```

Fig. 4. Typical smali code of the method field

Table 2
Code block information.

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

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: main function and sub function classification. There are four main components in Android system. Each component reacts to user's interactions using system default methods called Android life-cycle method [27]. For example, when a user starts an Android application, the method *OnCreate()* is executed. Developers overwrite the Android life-cycle methods to response to user's interactions. Thus, the overwritten methods (AndroidLifeMethod for short) can be seen as main functions. Other methods written by developers (DevpMethod for short) invoked by the AndroidLifeMethod are seen as sub functions. In this way, the AndroidLifeMethod and the related DevpMethod can be profiled together.

5.3. Instrumentation model generation

There are two models to be instrumented in Android application: the AdapTracer encoder model and the edge probability model. The above two models 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) is used to record the edge's execution frequency. The AdapTracer encoder model updates the edge's execution frequency using the edge's counter. We assign equal probability to multiple out-edges as follows:

$$p(e) = \frac{1}{|outedge(n)|}, \forall\ e \in outedge(n) \tag{1}$$

Where 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()* and the details can be found in Section 4.1. *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. It is almost the same as the implementation in the arithmetic coding library AMlib. In *Encoder()* and *ValueSelect()*, the PathID may overflow when we profile large programs. *OverFlow()* stores and resets PathIDs which are going to overflow. It is instrumented in the instruction of interval scaling in *Encoder()* and *ValueSelect()* [8] as shown in Algorithm 1. Once the PathID is going to overflow, it is stored in list and reset to zero for the next encoding.

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

5.4. Smali files instrumentation

Algorithm 3 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 because they are profiled together.

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 the current method. *getCodeBlockList()* gets the list of all code blocks of the 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 for 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.

5.5. Recompiling 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 the mobile phone using *adb* [28].

```
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
```

Algorithm 3. The AdapTracer instrumentation algorithm.

Table 3
CFG characteristics of benchmarks.

Application	CFG nodes	CFG edges	Nodes with multi-outedges	Max outedges	Program size
JGFArith	193	341	50	2	0.08MB
JGFAssign	161	281	40	2	0.11MB
JGFCast	69	117	16	2	0.05MB
JGFCreate	507	673	49	2	0.14MB
JGFException	77	123	9	2	0.05MB
JGFInstrumentor	170	242	13	4	0.07MB
JGFLoop	53	89	12	2	0.05MB
JGFMath	963	1323	120	2	0.21MB
JGFMethod	258	338	24	2	0.07MB
JGFSerial	225	361	22	2	0.09MB
JGFTimer	136	160	5	4	0.07MB
filemanager	46793	68973	6909	55	4.3MB
tinyclip-boardmanager	3558	4738	335	9	1.4MB
duckduckgo	14853	20813	1905	12	3.6MB
QKSMS	58788	88074	8636	64	4.2MB
swiftnotes	1464	2060	187	2	1.1MB
mirakelandroid	69759	98079	8603	17	5.5MB
antennapod	20958	31724	3517	49	4.4MB
audiobook	6756	10268	1069	49	2.1MB
dashclock	7649	11133	1079	9	0.6MB
muzei	14227	21143	2294	14	1.8MB
net.osmand.plus	106183	178279	22777	32	51.8MB

6. 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, because we have no idea of each 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.

6.1. Trace log analysis

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 tuple of `(methodname_classname, 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.

6.2. Execution paths restoration

After finding the corresponding method according to 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 Section 4.2.

6.3. Edge probability model calculation

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.

7. Evaluation

The experiment is running on Google Nexus 4, which is equipped with a 1.5GHz CPU and the 2GB of RAM. As shown in Table 3, we conduct experiments in two sets of benchmarks: Java Grande Benchmarks (JGF) [29] and a collection of Android applications from Google Play. We select most downloaded Google applications with various program structures. We have following two reasons for doing so: First, testing most downloaded applications can evaluate whether AdapTracer is practical and effective for profiling real-world applications. Second, testing applications with various program structures can evaluate the performance of AdapTracer on various control flow characteristics. The detailed CFG characteristics are shown in Table 3. Java Grande Benchmarks have been 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 CFG construction is based on the analysis of smali files which can be obtained through *apktool* [26]. We include the top-eleven most popular Android applications to evaluate the performance of AdapTracer. There are different CFG structures in these Android applications due to their different functions and the implementation style of developers. For example, "filemanager" is a file manager Android application and there is a relatively large number of occurrences of the "switch-case" construct in this Android application to response to user's different actions toward the files. The user's interactions with the mobile phone are simulated by recording a sequence of specific interactive events and replaying on the mobile phone.

7.1. Algorithm parameter selection

Fig. 5 shows the relationship between the *inc* and the trace size. Note that the *inc* is related to the speed of matching the edge execution probability as shown in Section 5. 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 and 1323 edges. Each loop performs different mathematical operations, such as modulus, maximize, minimize, logarithm and etc. We set *inc* to four different values from 1 to 4, which are labeled with 1-step to 4-step in Fig. 5. From the results, we can find that the change of trace size is the same as the 3-step. The bigger *inc* is, the more times the counter would need to be narrowed. So the 3-step is used in our AdapTracer. The experimental results also show that the trace size produced by AdapTracer becomes smaller following the program's execution. In PAP, the trace size is still large no matter how many times the program executes.

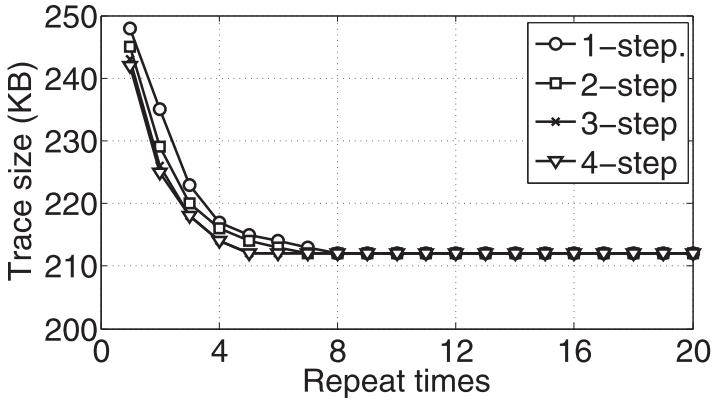

Fig. 5. Algorithm parameter selection.

7.2. Space overhead

We evaluate the space overhead between AdapTrace and PAP on modified JGF benchmarks and Android applications. As shown in Fig. 6 and Fig. 7, AdapTracer can achieve a consistently space-efficient comparison with PAP on both sets of benchmarks. Specifically, AdapTracer significantly reduces trace size by 44% and 57.1% on average compared with PAP in JGF benchmarks and Android applications respectively. Because of the two characteristics, AdapTracer can effectively improve the space efficiency: (1) Eliminating the overhead of recording CFG node in breakpoint. (2) Adaptively compressing the trace size using arithmetic code. Note that AdapTracer improves the space efficiency in terms of the algorithm characteristics but not the language characteristics. Therefore the space efficiency of 44% can carry over to other programs using different languages but not only Java. AdapTracer can save more space compared with PAP when the program size is larger (e.g., improving larger space efficiency when profiling applications from Google Play). Because in large size programs, PAP needs more breakpoints and larger space for storing the CFG node ID. The 44% and 57.1% space efficiency is the averaged result over different program size of JGF benchmarks and Google applications respectively. The trace size is relatively small because of the short execution time (from 20s to 400s according to different applications), We run each application for a sufficient time duration to validate the efficiency of AdapTracer. With longer execution time the trace size will increase, however, the improved space efficiency would stay almost same, because PAP is a static approach. For AdapTracer, it adaptively assigns less trace size for frequently executed paths.

7.3. Time cost

We also conduct an experiment to evaluate the time cost on two datasets (e.g., modified JGF benchmarks and Android applications). As shown in Figs. 8 and 9, AdapTracer only incurs execution overhead by 10% and 18% at most compared with PAP in modified JGF benchmarks and Android applications respectively. The overhead mainly stems from updating edge probabilities and the encoding interval. It is acceptable for currently wide-used mobile phone.

8. Case study of common bugs

In this section, we discuss how to identify three common bugs that occur in Android applications using AdapTracer: wasted computation for invisible GUI bugs, frequently invoked callback bugs and SMS (Short Message Service) security bugs. These bugs have been reported several times in the literature [30]. They play a crucial role for the energy saving and security of the Android system.

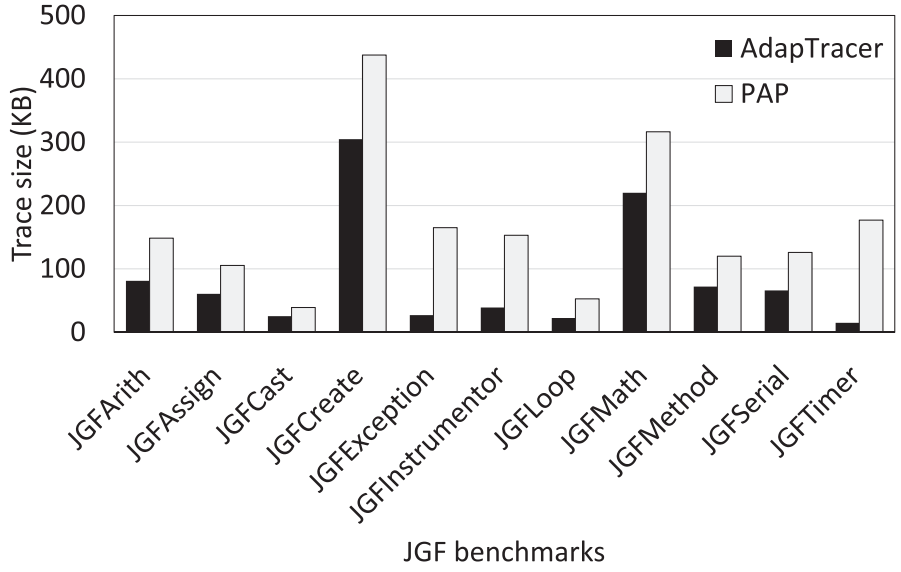

Fig. 6. Space overhead in JGF benchmarks.

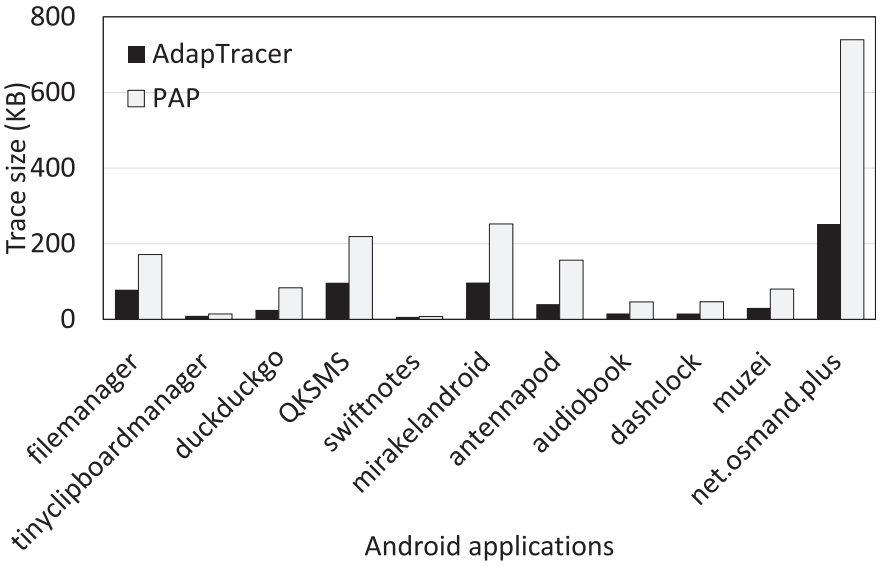

Fig. 7. Space overhead in Android applications.

8.1. Case 1: wasted computation for invisible GUI bugs

Bug description. This bug mainly comes from the carelessness of developing Android applications without considering the Android life-cycle. When an Android application switches to the background, the invisible GUI may still be updated. Fig. 10 presents the example smali code of a localization Android application. It lists the wasted computation bug and the corresponding bug-fixing patch. When this Android application launches, it registers a location listener to receive the location information to update its GUI (lines 6-8). The location listener is normally unregistered when the activity is destroyed (line 31). However, when a user launches this Android application and then switches it to background (Android OS will call *onPause()* or *onStop()*, but not *onDestroy()*), this Android application will keep listening the location information and updating the invisible GUI. As a consequence, the battery power will be drained. To fix this bug, we modify two Android life-cycle functions (i.e., *OnResume()* in lines 14-17 and *OnPause()* in line 25) to eliminate the wasted computation for the invisible GUI.

Tracing. We can efficiently capture the fine-grained control flow using AdapTracer, then further analyze works can be done by writing simple scripts. For example, recording the execution time of every code block. According to the specific behavior that is extracted from the control flow trace, we can find the frequently invoked Android life-cycle functions such as *onPause()* or *onResume()* and this suggests that there maybe some wasted computations inside these control flow actions.

8.2. Case 2: Frequently invoked callbacks bugs

Bug description. To timely response to the user's interactions, these callbacks are frequently invoked by Android OS and need to be light-weight. However, many such callbacks are ill-implemented in real-world applications. They are heavy-weight and can significantly slow down the Android applications. The typical example is the list view handler callback (as shown in Figs. 11 and 12).

For the example shown in Figs. 11 and 12, there are two elements in the listed item: a text label and an icon. Fig. 11 shows an inefficient version which performs the two aforementioned operations (lines 11–14). Fig. 12 applies a "view holder" design pattern to schedule the new and old list items. The basic idea is to reuse previously recycled list

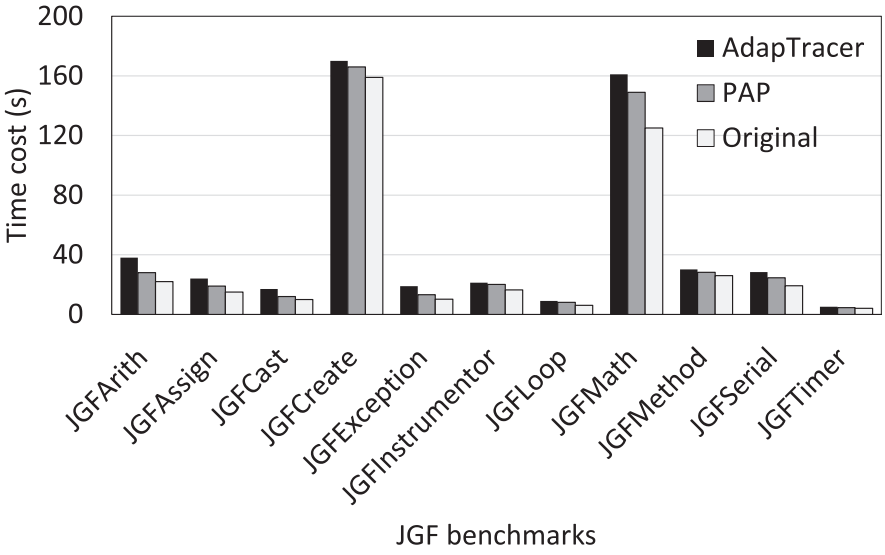

Fig. 8. Execution overhead in JGF benchmarks.

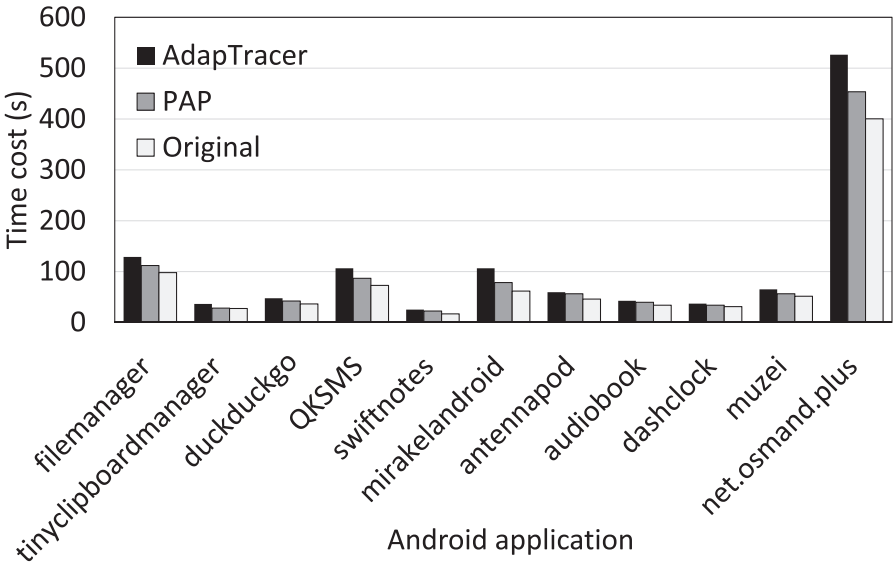

Fig. 9. Execution overhead in Android applications.

```
1    .field private lm:LocationManager;
2    .field private lc:Listener;
3
4    .method public onCreate()V
5       .prologue
6       new-instance v0, LocationManager;
7       new-instance v1, Listener;
8       invoke-direct {v0, v1},  register()V
9       return-void
10   .end method
11
12   .method public onResume()V
13      .prologue
14 +    invoke-direct {v0}, buildingUInotFinished()I
15 +    move-result v2
16 +    if-eqz v2, :cond_0
17 +    invoke-direct {v0}, requestLocationUpdates()V
18
19      :cond_0
20      return-void
21   .end method
22
23   .method public onPause()V
24      .prologue
25 +    invoke-direct {v1}, removeListener()V
26      return-void
27   .end method
28
29   .method public onDestory()V
30      .prologue
31      invoke-direct {v1}, removeListener()V
32      return-void
33   .end method
```

Fig. 10. Example code of wasted computation for invisible GUI.

items. It avoids list item layout inflation when there are recycled items for reuse (line 11). Besides, when a list is constructed for the first time, the references to its inner elements are identified and stored in a special data structure (lines 13–19, data structure defined at lines 1–6). Later, when reusing recycled items, these stored references can be used directly for updating content (lines 21–23), avoiding inner elements retrieval operations. By doing so, the computation overhead for inner elements retrieval is reduced and the memory for constructing new list

```
1   .method public getView()V
2      .prologue
3
4      const  v0, 0x7f030001
5      const  v1, 0x7f050001
6      invoke-virtual {v0}, findViewById()View
7      move-result-object v2
8      invoke-virtual {v1}, findViewById()View
9      move-result-object v3
10
11     const-string v4, "iOS"
12     const v5, 0x7f020001
13     invoke-virtual {v2, v4}, setText()V
14     invoke-virtual {v3, v5}, setImageResource()V
15     return-void
16  .end method
```

Fig. 11. Inefficient version of list view.

```
1  + .annotation system ViewHolder;
2  + value = {
3  +    const-string;
4  +    const;
5  + }
6  + .end annotation
7
8     .method public getView(View)V
9        .prologue
10
11 +     if-neqz p0, :cond_0
12 +
13 +     new-instance v0, ViewHolder;
14 +     const  v1, 0x7f030001
15 +     const  v2, 0x7f050001
16 +     invoke-virtual {v0, v1}, VfindViewById()View
17 +     move-result-object v3
18 +     invoke-virtual {v0, v2}, VfindViewById()View
19 +     move-result-object v4
20 +
21 +     :cond_0
22 +     invoke-virtual {p0}, getHolder()ViewHolder;
23 +     move-result-object v0
24
25       const-string v5, "iOS"
26       const v6, 0x7f020001
27       invoke-virtual {v0, v5}, VsetText()V
28       invoke-virtual {v0, v6}, VsetImageResource()V
29       return-void
30    .end method
```

Fig. 12. Modified version of list view.

```
1  .method public navigation()V
2    .prologue
3
4    new-instance v0, Date;
5    new-instance v1, Date;
6    const v2, 0xc
7    const v3, 0x14
8    const v4, 0x7e0
9    invoke-direct {v1, v2, v3, v4}, <inti>()V
10
11   invoke-virtual {v0, v1}, after()Z
12   move-result v2
13
14   if-eqz v2, :cond_0
15
16   invoke-virtual {p0}, generateLongRoute()V
17   goto :goto_0
18
19   :cond_0
20   invoke-virtual {p0}, generateShortRoute()V
21
22   :goto_0
23   return-void
24 .end method
```

Fig. 13. Example code of SMS security bug.

items is saved.

Tracing. When testing an Android application that utilizes the list view component, the control flow trace records the operational logic for the list view. If the logic of callback function *getView()* is simple, it implies that this application may have a frequently invoked callback bug. There are many metrics to quantify the complexity of the implemented function such as the execution time or the number of code lines in this function. We can thus target the code line corresponding to the control flow trace, and fix this bug using the efficient version as shown in Fig. 12.

8.3. Case 3: SMS (Short Message Service) security bugs

Bug description. SMS (Short Message Service) security bugs are not a vulnerability of the short message service. Instead, they are a typical logic bomb trigger. Logic bomb is a kind of malicious application logic that is executed or triggered only under certain circumstances.

As an example of the SMS security bugs, we present a navigation application as shown in Fig. 13. It is meant to assist a soldier in the battlefield to determine the shortest route to a given destination. It is legitimate for this application to collect the GPS-related information and send the information to the server. Then, results are received by the application and displayed to the user. Assuming this application contains another functionality that checks whether the current day is past a specific date (line 14). If the current day is not past this date, this application would give a shortest route as the user would expect (line 20). Otherwise, this application would queries the network for a long route (line 16). Thus, there is a SMS security bug, which may potentially trigger the malicious behavior, in the code line of date check.

Tracing. With the traced control flow, we can find an unexpected control flow when a specific specific SMS message is received at the mobile phone. For example, setting a series of predefined expected normal actions (e.g., a train of events) and comparing the difference between the captured event train and the predefined normal event train. The abnormal control flow in the trace reveals that there maybe a potential malicious behavior. We can thus further exam the suspicious code corresponding to the control flow branches to verify wether it is an actual SMS security bug.

9. 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 frequent long execution paths significantly, but the actual space usage may be the same as PAP in profiling frequent short execution paths. Namely, for frequent short execution paths, 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, No. 61772465, and the Fundamental Research Funds for the Central Universities (No. 2017FZA5013).

References

[1] M. Tancreti, V. Sundaram, S. Bagchi, P. Eugster, TARDIS: software-only system-level record and replay in wireless sensor networks, Proc. of ACM/IEEE IPSN, (2015), http://dx.doi.org/10.1145/2737095.2737096.
[2] S. Hao, D. Li, W.G.J. Halfond, R. Govindan, Estimating mobile application energy consumption using program analysis, Proc. of ACM/IEEE ICSE, (2013).
[3] S. Hao, D. Li, W.G. Halfond, R. Govindan, SIF: a selective instrumentation framework for mobile applications, Proc. of ACM MobiSys, (2013), http://dx.doi.org/10.1145/2462456.2465430.
[4] Y. Liu, C. Xu, S.C. Cheung, Characterizing and detecting performance bugs for Smartphone applications, Proc. of ACM/IEEE ICSE, (2014), http://dx.doi.org/10.1145/2568225.2568229.
[5] T. Ball, J.R. Larus, Efficient path profiling, Proc. of ACM MICRO, (1996).
[6] J.R. Larus, Whole program paths, Proc. of ACM/IEEE PLDI, (1999), http://dx.doi.org/10.1145/301618.301678.
[7] B. Li, L. Wang, H. Leung, F. Liu, Profiling all paths: a new profiling technique for both cyclic and acyclic paths, J. Syst. Softw. (2012) 1558–1576.
[8] A. Said, Introduction to arithmetic coding theory and practice, Technical Report HPLC2004C76, Hewlett-Packard Laboratories Report, 2004.
[9] G. Chen, W. Dong, Adaptive path profiling using arithmetic coding, Prof. of IEEE ICPADS, (2015), pp. 164–171.
[10] D.G. Melski, Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation, Technical Report, (2002).
[11] B. Kasikci, T. Ball, G. Candea, J. Erickson, M. Musuvathi, Efficient tracing of cold code via bias-free sampling, USENIX ATC, (2014).
[12] T. Apiwattanapong, M.J. Harrold, Selective path profiling, Proc. of ACM PASTE, (2002), http://dx.doi.org/10.1145/586094.586104.
[13] M. Arnold, B.G. Ryder, A framework for reducing the cost of instrumented code, Proc. of ACM PLDI, (2001), http://dx.doi.org/10.1145/378795.378832.
[14] R. Joshi, M.D. Bond, C. Zilles, Targeted path profiling: lower overhead path profiling for staged dynamic optimization systems, Proc. of ACM/IEEE CGO, (2004).
[15] A.V.N. Kapil Vaswani, T.M. Chilimbi, Preferential path profiling: compactly numbering interesting paths, Proc. of ACM POPL, (2007).
[16] R. Chouhan, S. Roy, S. Baswana, Pertinent path profiling: tracking interactions among relevant statements, Proc. of ACM/IEEE CGO, (2013).
[17] M.D. Bond, K.S. McKinley, Practical path profiling for dynamic optimizers, Proc. of ACM/IEEE CGO, (2005), http://dx.doi.org/10.1109/CGO.2005.28.
[18] M.D. Bond, K.S. McKinley, Continuous path and edge profiling, Proc. of ACM MICRO, (2005).
[19] D.S. Mohammed Afraz, A. Kanade, P3: partitioned path profiling, Proc. of ACM FSE/ESEC, (2015).
[20] D.C. DElia, C. Demetrescu, Ball-larus path profiling across multiple loop iterations, Proc. of ACM OOPSLA, (2013).
[21] C. Yang, S. Wu, W.K. Chan, Hierarchical program paths, ACM Trans. Softw. Eng. Methodol. 25 (3) (2016) 27:1–27:44.
[22] R. Lim, L. Thiele, Testbed assisted control flow tracing for wireless embedded systems, Proc. of EWSN, (2017), pp. 180–191.
[23] Rapitime, https://www.rapitasystems.com/products/rapitime.
[24] T.M. Cover, J.A. Thomas, Elements of Information Theory, (2012).
[25] Smali syntax, https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.
[26] Apktool, http://ibotpeaches.github.io/apktool/.
[27] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau,

P. McDaniel, FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android Apps, Proc. of ACM PLDI, (2014), http://dx.doi.org/10.1145/2594291.2594299.
[28] Android debug bridge, http://developer.android.com/tools/help/adb.html.
[29] Jgf benchmarks, http://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking.
[30] Android bug list, https://code.google.com/p/android/issues/list.

Gonglong Chen(S'15) received his B.S. degree from the College of Criminal Justice at East China University of Political Science and Law.He is currently a second year Ph.D student at the College of Computer Science in Zhejiang University. His current research interests include wireless and mobile computing. He is a student member of IEEE.

Wei Dong (S'08-M'11) received the B.S. and Ph.D. degrees in computer science from Zhejiang University, Hangzhou, China, in 2005 and 2011, respectively. He is currently a Professor with the College of Computer Science, Zhejiang University. His research interests include network measurement, wireless and mobile computing, and sensor networks. He is a member of IEEE and ACM, and a senior member of CCF.