Received June 13, 2020, accepted July 12, 2020, date of publication July 17, 2020, date of current version July 29, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3009990

A Gray Box for Visualizing Instruction Sequence Based on Improved Suffix Tree

DONGLIN WANG, (Member, IEEE), AND JIANDONG FANG
Department of Information Engineering, Inner Mongolia University of Technology, Hohhot 010051, China

Corresponding author: Donglin Wang (1104098996@qq.com)

This work was supported in part by the Inner Mongolia Major Science and Technology Project under Grant MZD20180301, in part by the Inner Mongolia Scientific and Technological Project under Grant N220001000409311 and Grant N220001000409278, and in part by the Inner Mongolia Natural Science Foundation Project under Grant 2019MS06023.

ABSTRACT Gray box is a kind of device in which the working process of a program or system is locally recognized. Gray box testing, also known as gray box analysis, is a software debugging method based on the limited cognition of the internal details of the program. Testers may know how system components interact with each other, but they lack a detailed understanding of internal program functions and operation. So the construction of gray box is particularly important. The most original gray boxes are static debugger and dynamic debugger. And then reflexion model, which reduces the manual work greatly, is developed and applied. The latest gray boxes are focus on regarding instructions as a natural language using the mature mathematical model to mine their internal value. Adhering to the idea of latest researches, our paper improves the original suffix tree and use the improved suffix tree as a mathematical models to analyse and visualize the internal logic of instructions. Our gray box aims at solving three problems in practical application. In addition, we explain the complexity of instruction sequence and put forward a prediction formula for the building part. By experiment, we prove the time complexity of each part and the correctness of the prediction formula, and show the effect of visualizing part.

INDEX TERMS Gray box, reverse engineering, suffix tree, visualize, instruction preprocess.

I. INTRODUCTION

Gray box testing [1] is between white box testing and black box testing [2], which not only pays attention to the correctness of output and input, but also focuses on the internal situation of the program. Gray box testing is not as detailed and complete as white box testing, but it pays more attention to the internal logic of the program than black box testing. Nowadays, the program size is bigger and bigger, and the encryption method is more and more complex. Therefore, there are few opportunities to use white box testing in practice, and the effect of black box testing is often not up to standard. The improvement of gray box testing technology will greatly improve the security quality of programs before officially released.

Although gray box testing is very advanced, there are three problems in practical application.

The associate editor coordinating the review of this manuscript and approving it for publication was Tu Ngoc Nguyen.

1) How to collect the source instructions when facing the compression encryption?
   The compression encryption [3] is different from obfuscated code [4] essentially. The obfuscated code blocks the normal understanding by disturbing instructions' logic [5], however, we still can get the encrypted source code whether by static analysis or dynamic analysis. But when facing the compression encryption such as UPX [6], the static analysis loses effect immediately, and the dynamic analysis can be stopped by anti debug techniques such as timeout detection [7] or attached detection [8]. So it is necessary to innovate a new method to collect the source instruction effectively.
2) How to compress the source instructions into a suitable volume for the chosen model?
   For example, if a researcher wants to use N-grams model to analyse the instructions, he has to map the instructions into the form of vector domain description [9]. But if the chosen model need $O(n^2)$ or more time complexity to build or traverse such as suffix tree [10], mere mapping

will be not enough for the high-volume instructions, because the process will cost too much time. So it is necessary to compress the source instructions into suitable volume without losing the key information.

3) How to reuse the built model quickly and mine the hierarchical inclusion relationship?
For example, a researcher can use DFS or BFS to traverse one built suffix tree many times for different aims [11]. That does reuse the model but costs too much time, because each DFS or BFS needs O(n^2) time complexity. And such traverse results are lack of hierarchical inclusion relationship [12], because how many times traversing are needed and where once traversing finishes in are both unknown until the traversing does finish. In other words, if we want to get the hierarchical inclusion relationship among two or more parts of instructions like the red-bordered ones in figure 1 and figure 2, by the traditional suffix tree, we have to do n times O(n^2)-time-complexityed traversing, that is, the total process for hierarchical inclusion relationship needs O(n^3) time complexity. So it is necessary to find a suitable model and invent a low-time-complexityed traversing algorithm.

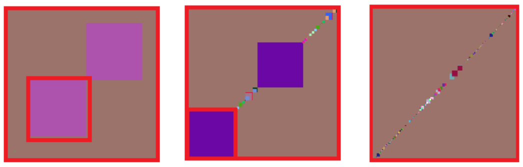


FIGURE 1. Three parts of instruction.

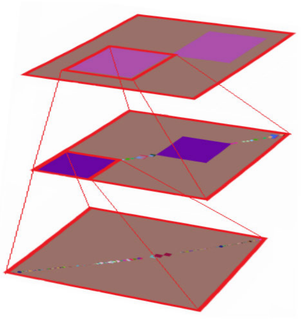
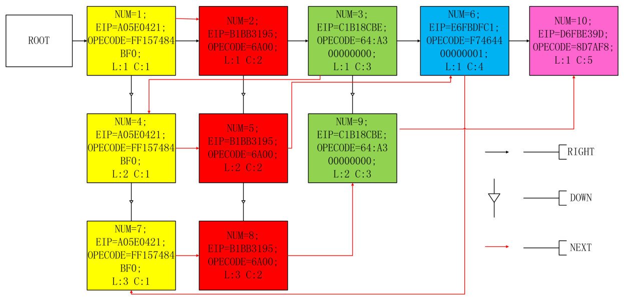


FIGURE 2. Hierarchical inclusion relationship among three parts of instruction.

There have been lots of related works aiming to solve these three problems. And by comparing with the related works, we explain the innovative points and the corresponding improvement effect of our paper:

1) For collecting the source instruction when facing the compression encryption.
The most traditional way, which is still used currently, uses static analysis or dynamic analysis to crake the compression encryption [13], although it consumes a lot of human labor. The latest way can be summed as "ignoring the encryption completely, using dynamic analysis or dump [14] to record the running instructions simply, mining the instruction's inner logic by mathematical way after all the recording is finished". The latest way reduces human labor greatly and makes the whole process finish within predictable time.
Our research inherits the idea of the latest way, but makes innovations in setting breakpoints. Most of the existing researches are attaching and recording the instructions from OEP [15], which records lots of irrelevant contents. In order to make the recorded instructions closer to the theme, we propose a method which searches the aim information's memory addresses, sets Dr0 register (8 bits) [16] into the first-searched address's value, and sets Dr7 into 0×30101. When the first STATUS_SINGLE_STEP exception is triggered, repeat to set the TF flag [16] into 1 and record the instructions like what the dynamic analysis does.
Compared with the latest way which records instruction from OEP, our debugger can record instruction more purposefully and avoid timeout detection [7] effectively.

2) For compressing the source instructions into a suitable volume for the chosen model.
Basicly, the instruction compression here must satisfy that the compression result can be analysed by the corresponding model without decompression. So it is different from the common compression algorithm [17], [18] used in our daily life. And we can find many related works satisfying this point [19], [20]. The innovation of our research is that we compress not only the instructions but also their corresponding running addresses. Only both-the-samed recordings can be compressed into the-same-coloured structure object like figure 3 showing. And we use three kinds of pointers to save different relationships between each recording.


ROOT
NUM=1; EIP=A05E0421; OPECODE=FF157484BF0; L:1 C:1
NUM=2; EIP=B1BB3195; OPECODE=6A00; L:1 C:2
NUM=3; EIP=C1B18CBE; OPECODE=64:A300000000; L:1 C:3
NUM=6; EIP=E6FBDFC1; OPECODE=F7464400000001; L:1 C:4
NUM=10; EIP=D6FBE39D; OPECODE=8D7AF8; L:1 C:5
NUM=4; EIP=A05E0421; OPECODE=FF157484BF0; L:2 C:1
NUM=5; EIP=B1BB3195; OPECODE=6A00; L:2 C:2
NUM=9; EIP=C1B18CBE; OPECODE=64:A300000000; L:2 C:3
NUM=7; EIP=A05E0421; OPECODE=FF157484BF0; L:3 C:1
NUM=8; EIP=B1BB3195; OPECODE=6A00; L:3 C:2
RIGHT
DOWN
NEXT
FIGURE 3. The compression result in the memory.

Compared with the related works, our algorithm transforms the basic unit from "char" to "sentence" by compressing. Concretely speaking, we compress 100 chars into a structure object which contains all the key values of the corresponding sentence. This compression will make ten thousand times speed improvement for the latter building process of suffix tree. For example, in our experiment, our algorithm can finish building process of 290809 instructions in 213000 ticks (nearly 213 seconds). If we use the "char" as the basic unit, it will cost about 591 hours (24 days), so the speed improvement caused by compressing is obvious.

3) For reusing the built model quickly and mine the hierarchical inclusion relationship.
The current works surely have noticed the large potential value of this research. For example, Rainer Koschke's research demonstrates how suffix tree can be used to obtain a scalable comparison in a faster way [21]. And Kai Huang's team develops an intelligent instruction sequence based malware categorization system (ISMCS) using a novel weighted subspace clustering method [22]. The most obvious feature of current works is reusing and mining through mathematical calculation, which do improve model utilization effectively and calculate out some hierarchical indexes for the instructions, but the calculation results are not able to be expanded any more. In other words, the researches still have to do lots of redundant calculations for getting the hierarchical inclusion relationship like figure 2 shown.
The innovation of our research is that we put forward an "improved suffix tree" on the base of traditional suffix tree [10]. By the improved suffix tree, we can expand the hierarchical inclusion relationship among different parts of instructions by traversing the built tree not by mathematical calculation. And we can visualize the traversing result like the figure 1 and figure 2 showing.
Compared with the traditional suffix tree, the improved suffix tree can finish traversing the visualizing both in the linear time complexity strictly.

Our gray box is composed of collecting part, compressing part, improved suffix tree and visualizing part.The relationship among each part is like figure 4 shown:

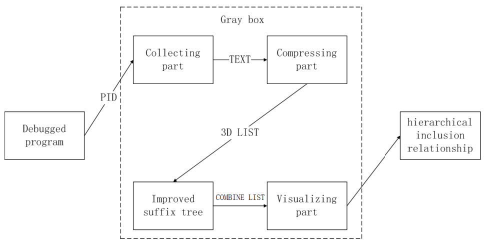


FIGURE 4. The composition of our gray box.

This paper is organized as follows. Section I introduces the background, existing problems, related works, our innovation points and corresponding improvement effects. Section II illustrates how our gray box collects the source instructions when facing the compression encryption. Section III illustrates how our gray box compresses the source instructions into a suitable volume. Section IV illustrates the similarities and differences between improved suffix tree and traditional suffix tree, discuss the constructing time consume for suffix tree, and give out a fast prediction formula for constructing time consume. Section V illustrates how gray box reuses the improved suffix tree quickly in order to mine the hierarchical inclusion relationship of the source instructions. Section VI summarizes the conclusions of this paper. Finally, the future works are on Section VII.

II. COLLECTING

This section aims at collecting the source instructions. By setting breakpoints innovatively, we realize a dynamic debugger which can record the target-specific assembly instructions and format them into a txt file. Three types of breakpoint called MBP, DRX and TF are used in this section.

A. DEFINITION

The definitions of related variables in collecting part are like table 1 shown:

TABLE 1. The variables of collecting part.

Num	Name	Type	Definition
1	AP	int	the aim process which process ID is PID.
2	TI	byte	the byte code of target information.
3	TIA	byte *	the array maintaining addresses of TI.
4	MBP	BREAK	the breakpoint setting memory property into PAGE_NOACCESS.
5	WI	int	the current running time of WaitForDebugEvent().
6	WB	int	the allowed maximum value of WI.
7	DE	EVENT	the instance of DEBUG_EVENT.
8	DRX	BREAK	the breakpoint setting registers dr0-dr7 into corresponding values.
9	AI	CONTENT	the breakpoint setting registers dr0-dr7 into corresponding values.
10	TEXT	FILE	the txt file recording module result.
11	TF	FLAG	the 9th flag bit of register PSW.

The pseudo code of collecting part is described in Algorithm 1:

B. EXPLANATION

1.Why must MBP be set before DRX?

The decision is based on practise. If we set DRX immediately after attaching, the debugged application may reset them by safeguard function before triggering, so that we can not accept exception at all. Compared with DRX, MBP can be hardly used in safeguard function, because determining every instruction in a memory-paged range costs too much time. That is also the reason why we do not just use "MBP+TF".

2.How many times is MBP set and triggered?

Only the TIA[0] is set as MBP once time, and it is also just triggered once in loop 1, in which we set DRX at the same time. The triggering position between MBP and DRX can not be over one memory-paged rage, so the DRX can be hardly reset.

3.Why does collecting() just have two loops for three types of breakpoint?

Loop 1 is for MBP obviously. Because DRX and TF trigger the same exception, the loop 2 triggers once DRX first and sets 9-th flag bit into 1 at the same time. Except the first DRX, the rest of loop 2 triggers TF (WB-1) times simply or shuts down with the crash of debugged program.

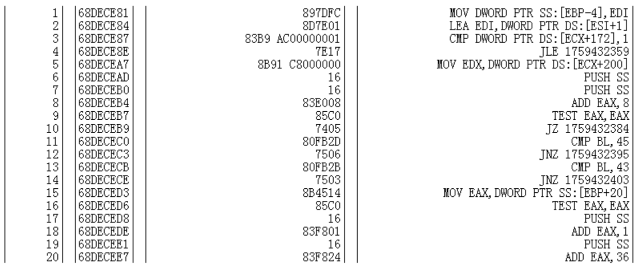
Algorithm 1 Collecting()

```
 1:attach to AP
 2:search TI and save TI by TIA
 3:set MBP by TIA[0]
 4:WI=0
 5:while WI<WB
 6:    wait and record DE
 7:    if DE==EXCEPTION_ACCESS_VIOLATION
 8:        set DRX(dr0=TIA[0] dr7= 0 × 30101) in all
             thread
 9:        WI=WB
10:    continue AP
11:    WI++
12:delete MBP
13:WI=0
14:while WI<WB
15:    wait and record DE
16:    if DE==EXCEPTION_SINGLE_STEP
17:        record AI by TEXT
18:        TF=1
19:    continue AP
20:    WI++
21:detach from AP
```

C. EXPERIMENT

We chose a large-scale online game to test whether collecting() can effectively obtain the instruction set. The game program consists of two parts: the starting end (1.09 MB) and the client end (1.05 GB). When the game starts, the starting end decompresses the client end. We set the ASCII code of the game currency as TI. Then we attach our gray box to the game. When we do currency related operations in the game, our gray box can record the currency related instruction set directly. We set WB as one millon. When we do the different currency related operations, we can record TEXTs of different length from 2292 to 1000000.

The example of TEXT is like figure 5 shown:



```
 1 |68DECE81|            897DFC|    MOV DWORD PTR SS:[EBP-4],EDI
 2 |68DECE84|            8D7E01|    LEA EDI,DWORD PTR DS:[ESI+1]
 3 |68DECE87|   83B9 AC00000001|    CMP DWORD PTR DS:[ECX+172],1
 4 |68DECE8E|              7E17|                  JLE 1759432359
 5 |68DECEA7|     8B91 C8000000|   MOV EDX,DWORD PTR DS:[ECX+200]
 6 |68DECEAD|                16|                         PUSH SS
 7 |68DECEB0|                16|                         PUSH SS
 8 |68DECEB4|            83E008|                       ADD EAX,8
 9 |68DECEB7|              85C0|                   TEST EAX,EAX
10 |68DECEB9|              7405|                   JZ 1759432384
11 |68DECEC0|            80FB2D|                      CMP BL,45
12 |68DECEC3|              7506|                  JNZ 1759432395
13 |68DECECB|            80FB2B|                      CMP BL,43
14 |68DECECE|              7503|                  JNZ 1759432403
15 |68DECED3|            8B4514|  MOV EAX,DWORD PTR SS:[EBP+20]
16 |68DECED6|              85C0|                   TEST EAX,EAX
17 |68DECED8|                16|                         PUSH SS
18 |68DECEDE|            83F801|                       ADD EAX,1
19 |68DECEE1|                16|                         PUSH SS
20 |68DECEE7|            83F824|                      ADD EAX,36
```

FIGURE 5. The result of collecting().

D. COMPARISON

If we use the traditional way like IDA [23] to debug the game, just for the starting end (1.09 MB), 1863 functions can be resolved. To explore call relations among 1863 functions is like looking for a needle in a haystack. For the client end which is compressed into some unpublished format, IDA is unable to load the file into workspace.

Compared with the latest way which dumps [14] or records the instructions from OEP [15], [24], our gray box can record instruction more purposefully and avoid timeout detection [7] effectively. If our gray box runs the game from OEP, before we do currency related operations, the game can detect itself is being debugged and kill itself. As a result we can't record currency related instruction set at all.

III. COMPRESSING

This section aims at reading TEXT into memory and mapping these context into a 3D LIST. The reason why the result is called 3D list is that every node in the list has 3 direction pointers: PRIGHT, PDOWN and PNEXT. The PRIGHT represents the first-lined, right adjacent and different kind relationship. The PDOWN represents the same column, down adjacent and the same kind relationship. The PNEXT represents the order adjacent relationship. And we denote the basic node structure of 3D LIST as NOOD.

The innovation of this block is rather than the traditional way of using a single character as the basic analysis unit, we compress and map the sentence into structure node as basic unit. Actually we formulate 100 characters as a sentence and map it into a node. There will be a great execution speed benefit facing the "bad situation", even promote ten thousand times.

TABLE 2. The variables of compression.

Num	Name	Type	Definition
1	ROOT	NODE	the head node of the 3D list.
2	CT	int	the current time of reading-per-thousand.
3	SUM	int	the number of TEXT's total line.
4	FP	FILE *	the pointer of TEXT's context.
5	TC	BYTE *	the temporary storage memory saving TEXT context.
6	AL	int	the actual line number of TC.
7	IS	int	the current time of inner segmentation.
8	PTC	BYTE *	the pointer of TC's context.
9	NODE	NODE	the structure saving the mapped single line of TEXT.
10	PNODE	NODE *	the pointer of NODE.
11	PRIGHT	NODE *	the pointer connecting the right direction relationship between two NODEs.
12	PDOWN	NODE *	the pointer connecting the down direction relationship between two NODEs.
13	PNEXT	NODE *	the pointer connecting the next direction relationship between two NODEs.
14	CN	int	the column number of NODE.
15	LN	int	the line number of NODE.
16	AF	FLAG	the flag to represent if NODE has been added into the 3D-linked list.
17	RT	int	the repeat time of all NODEs.
18	UT	int	the unique time of all NODEs.

A. DEFINITION

The definitions of related variables in compressing part are like table 2 shown:

The pseudo code of compressing part is described in Algorithm 2:

Algorithm 2 Compressing()

```
1:RT=0;UT=0;CT=1
2:build ROOT
3:while CT≤SUM/1000+1
4:   local FP
5:   set TC by FP
6:   initial AL
7:   IS=1
8:   while IS≤AL
9:      initial NODE
10:     local PTC
11:     set NODE by PTC
12:     build3D()
13:     IS++
14:  CT++
```

The pseudo code of build3D() is described in Algorithm 2.1:

Algorithm 2.1 Build3D()

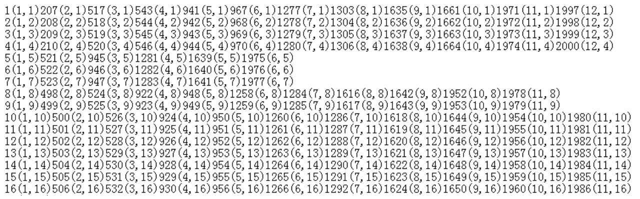
```
1:CN= −1;LN=1;PNEXT=&ROOT;PRIGHT
      =&ROOT
2:while PRIGHT!=NULL
3:   CN++
4:   if PRIGHT->eip==PNODE->eip&&
        PRIGHT->opcode==PNODE->opcode
5:      PDOWN=PRIGHT
6:      while PDOWN->next!=NULL
7:         LN++
8:         PDOWN=PDOWN->dowm
9:      PDOWN->dowm=PNODE
10:     PNODE->line=LN+1;PNODE->
          column=CN;AF=1
11:     PNEXT->next=PNODE;PNEXT=PNODE
12:  if AF==1
13:     RT++;break
14:  if PRIGHT->right==NULL&&AF==0
15:     PRIGHT->right=PNODE;UT++
16:     PNODE->line-1;PNODE->column=CN+1;
17:     PNEXT->next=PNODE;PNEXT=PNODE;
       break
18:  PRIGHT=PRIGHT->right
```

B. EXPERIMENT

By the Algorithm 2, we transform the TEXT into 3D LIST like figure 3 shown. And we test the result by traverse the 3D LIST and print the node contents into txt file like figure 6 shown:

C. COMPARISON

By compressing the TEXT into 3D LIST, we not only realize the compression of memory space, but also realize the conversion of basic unit from "char" to "sentence". Lai Huoyao's team improved the UKK suffix tree and proposed the SBA suffix tree [25]. In their experiment, the basic unit is "char", and the capacity of experiment is 600000 characters. The time consume of their construction algorithm is like figure 7 shown:



```
1(1,1)207(2,1)517(3,1)543(4,1)941(5,1)967(6,1)1277(7,1)1303(8,1)1635(9,1)1661(10,1)1971(11,1)1997(12,1)
2(1,2)208(2,2)518(3,2)544(4,2)942(5,2)968(6,2)1278(7,2)1304(8,2)1636(9,2)1662(10,2)1972(11,2)1998(12,2)
3(1,3)209(2,3)519(3,3)545(4,3)943(5,3)969(6,3)1279(7,3)1305(8,3)1637(9,3)1663(10,3)1973(11,3)1999(12,3)
4(1,4)210(2,4)520(3,4)546(4,4)944(5,4)970(6,4)1280(7,4)1306(8,4)1638(9,4)1664(10,4)1974(11,4)2000(12,4)
5(1,5)521(2,5)945(3,5)1281(4,5)1639(5,5)1975(6,5)
6(1,6)522(2,6)946(3,6)1282(4,6)1640(5,6)1976(6,6)
7(1,7)523(2,7)947(3,7)1283(4,7)1641(5,7)1977(6,7)
8(1,8)498(2,8)524(3,8)922(4,8)948(5,8)1258(6,8)1284(7,8)1616(8,8)1642(9,8)1952(10,8)1978(11,8)
9(1,9)499(2,9)525(3,9)923(4,9)949(5,9)1259(6,9)1285(7,9)1617(8,9)1643(9,9)1953(10,9)1979(11,9)
10(1,10)500(2,10)526(3,10)924(4,10)950(5,10)1260(6,10)1286(7,10)1618(8,10)1644(9,10)1954(10,10)1980(11,10)
11(1,11)501(2,11)527(3,11)925(4,11)951(5,11)1261(6,11)1287(7,11)1619(8,11)1645(9,11)1955(10,11)1981(11,11)
12(1,12)502(2,12)528(3,12)926(4,12)952(5,12)1262(6,12)1288(7,12)1620(8,12)1646(9,12)1956(10,12)1982(11,12)
13(1,13)503(2,13)529(3,13)927(4,13)953(5,13)1263(6,13)1289(7,13)1621(8,13)1647(9,13)1957(10,13)1983(11,13)
14(1,14)504(2,14)530(3,14)928(4,14)954(5,14)1264(6,14)1290(7,14)1622(8,14)1648(9,14)1958(10,14)1984(11,14)
15(1,15)505(2,15)531(3,15)929(4,15)955(5,15)1265(6,15)1291(7,15)1623(8,15)1649(9,15)1959(10,15)1985(11,15)
16(1,16)506(2,16)532(3,16)930(4,16)956(5,16)1266(6,16)1292(7,16)1624(8,16)1650(9,16)1960(10,16)1986(11,16)
```

FIGURE 6. The result of compressing().

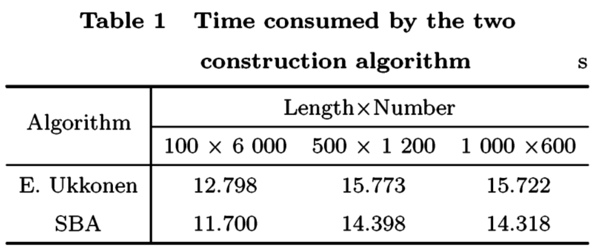


Table 1 Time consumed by the two construction algorithm s

Algorithm	Length×Number		
	100 × 6 000	500 × 1 200	1 000 ×600
E. Ukkonen	12.798	15.773	15.722
SBA	11.700	14.398	14.318

FIGURE 7. The time consume of Lai Huoyao's construction algorithm.

According to the figure 7, facing 600000 bytes, the time consume of SBA or UKK is between 11.700s-15.722s. When the basic unit is transformed into sentence through our gray box, the constructing time consume (building tick) of improved suffix tree is like table 3 shown:

TABLE 3. The constructing time consume of improved suffix tree.

num	length	T	building tick	building index
1	290809	7245116877	213048 ms	0.732604562
2	263553	6791256381	196558 ms	0.745800655
3	230001	3691873605	105390 ms	0.458215399
4	200001	1873888605	51757 ms	0.258783706
5	180001	1160113047	31964 ms	0.177576791
6	154164	826350680	21648 ms	0.140421888
7	130001	717590870	20438 ms	0.157214175
8	109151	602084536	19080 ms	0.174803712
9	70001	38295472	2832 ms	0.040456565
10	33133	4926941	1460 ms	0.04406483
11	1343	238099	5 ms	0.003723008
12	963	386071	5 ms	0.005192108
13	797	237608	4 ms	0.005018821
14	406	2647	1 ms	0.002463054
15	260	259	0 ms	0

By the 10th data in table 3, we can see that 33133 × 100 bytes, which is nearly five times as much as Lai Huoyao's sample, only takes about 1.4s. The sample size is increased by 5 times and the time consume is reduced to one-tenth. And the constructing processes of UKK, SBA and our improved suffix tree have no structural change in nature. In other words, by compressing part transforms the basic unit from "char" to "sentence", we get a nearly 50 times acceleration facing 3M TEXT. As the sample size becomes bigger, the acceleration will be more obvious.

IV. IMPROVED SUFFIX TREE

In this section, we first introduce the two main differences between our improved suffix tree and the traditional UKK suffix tree [10]. Then discuss the constructing time consume of suffix tree. Finally, give out a fast prediction formula for constructing time consume.

A. DIFFERENCE

The algorithm architectures of UKK [10] and our improved suffix tree are nearly the same. The differences can be summed up into two points: first, we change the UKK's *suffix link* (pointer) into SUFFIX LINK (structure list). Second, our improved suffix tree keeps the necessary reverse relationships among each kind of structure.

1) SUFFIX LINK

In Ukkonen's paper [10], the illustration of string "cacao" is like figure 8 shown:

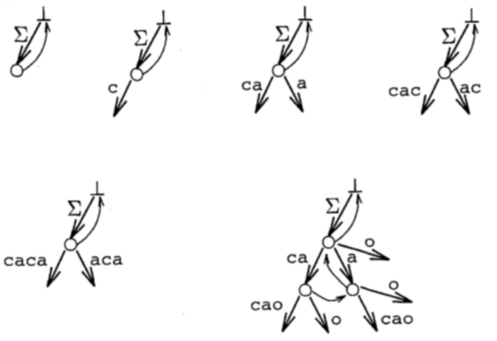


FIGURE 8. The suffix tree of string "cacao" in Ukkonen's paper.

Because of the *suffix link*, the constructing process UKK looks complicated. We replace *suffix link* (pointer) with SUFFIX LINK (structure list), and save SUFFIX LINK till the end of gray box testing. We denote the SUFFIX structure as S(p,n,f,o,l). The p represents the pointer which precisely locals the compared position on the improved suffix tree. The n represents the pointer which makes SUFFIXs into a link. The f represents the flag which represents the state of S. The o represents the order of corresponding NODE in the 3D LIST. The l represents the length of corresponding common prefix.

By replacing *suffix link* with SUFFIX LINK, the illustration of string "cacao" shown by the improved suffix tree is like figure 9 shown:

The replacement not only makes the algorithm clear, but also provides the foundation for the next section to reuse the improved suffix tree.

2) REVERSE RELATIONSHIP

Because the Ukkonen's suffix tree [10] is an one-way tree, it just can be traversed from root (top) to the leaves (bottom).

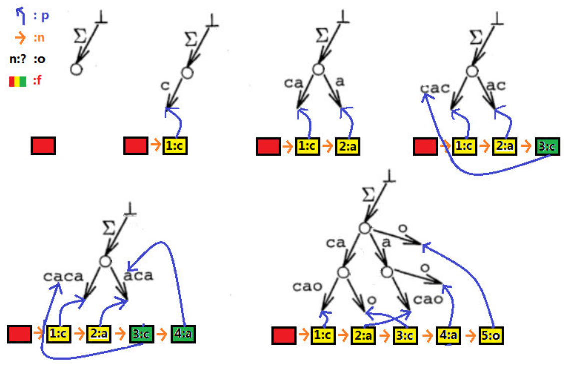


FIGURE 9. The improved suffix tree of string "cacao".

There is no doubt that the UKK suffix tree has achieved remarkable success in substring inclusion relationship search [26] and palindrome detection [27], which just need linear time to finish traversing. But if we want to use the UKK suffix tree to get the hierarchical inclusion relationship like figure 2 shown, we at least need to do DFS or BFS, which exactly need quadratic time.

For getting the hierarchical inclusion relationship in linear time, we add two reverse relationship pointer. We denote the node of improved suffix tree as N(n,e,o). The n represents the pointer which points to N's parent node. The e represents the pointer which points to N's incoming edge. The o represents the information which has been defined in UKK suffix tree.

B. CONSTRUCTING TIME CONSUME

In the abstract of paper *On-line construction of suffix trees* [10], Ukkonen said "Regardless of its quadratic worst case this latter algorithm can be a good practical method when the string is not too long". Besides Ukkonen's paper, nearly all the researches related with suffix tree emphasize that they can finish constructing process by linear time in specific background [25]. But what is the constructing time consume for general situation? The constructing process using *suffix link* or SUFFIX LINK is the same essentially. So in the following paragraphs, the improved suffix tree's constructing time consume can be considered the same as traditional UKK suffix tree's.

we have denoted the basic unit of improved suffix tree's source data as NODE in section III. In Ukkonen's paper, the NODE saves the information of "char". For our gray box, the NODE saves the information of "sentence". And we define the new-keyworded NODEs as debut NODE, such as the '1:c', '2:a' and '5:o' in the string of "cacao". We define the NODEs which make more SUFFIXs' p point to the leaf of improved suffix tree as acceleration NODE, such as the '1:c', '2:a' and '5:o' in the string of "cacao" shown by figure 9.

Three conclusions are emphasized as following:

Conclusion 1, debut NODE must be acceleration NODE, and debut NODE must make all the SUFFIX's p point to leaves at that time.

Conclusion 2, acceleration NODE can not be debut NODE, such as the '6:a'in the string of "abcabax". The a has been appear in '1:a' and '4:a' repeatedly, but when the '6:a' is added into improved suffix tree, the end position of the suffix which begin from '4:a' and '5:b' are linked to the leaf at that time. And the acceleration NODE doesn't ensure to make all the SUFFIXs' p point to leaves except the SUFFIX of itself, such as the '8:x' in the string of "abcbxabxy". When '8:x' is added, the suffix "abx" has never appeared before, but the suffix "bx" has appeared from '4:b'. So the suffix "abx" link to leaf, the suffix "bx" is still on the edge of improved suffix tree.

Conclusion 3, the SUFFIXs whose p have pointed to leaves must be added into the SUFFIX LINK earlier. Like figure 9 shown, the yellow SUFFIXs must be at the left side of the green ones.

Till now, we can divide the NODE link (source data) by the acceleration NODEs. Such as the string "cacao" can be divided into "c a cao", the string "abcbxabxy" can be divided into "a b c bx abx y". And we define some variables like table 4 shown:

TABLE 4. The definition of source data.

Num	Name	Type	Definition
1	i	int	the order number of the acceleration NODE from left to right,which can also be regarded as group's order number.
2	j	int	the intra-grouped order number of NODE.
3	$C_{i,j}$	NODE	the j-th NODE in the i-th group.
4	t_i	int	the total number of the i-th group.
5	t	int	the total number of the NODE link.

Now the NODE link can be represented like (1) showing:

$$C_{1,1} \ldots C_{1,t_1} \ldots\ldots C_{i,1} \ldots C_{i,t_i} \ldots\ldots C_{n,1} \ldots C_{n,t_n} \quad (1)$$

We suppose that all the acceleration NODEs can make all the SUFFIXs points to leaves. Now the i-th group's total comparing time can be represented by (2).

$$1 + 2 + 3 + \ldots + t_i = \frac{(1+t_i)t_i}{2} \quad (2)$$

So the total comparing time of constructing process can be calculated out by (3).

$$\sum_{i=1}^{n} \frac{(1+t_i)t_i}{2} \quad (3)$$

We use t to represent the total node number of NODE link and get (4) as the result.

$$\frac{1}{2}(\sum_{i=1}^{n} t_i^2 + t) \quad (4)$$

Lower boundary:

The n-dimensional Cauchy-inequality can be represented by (5).

$$(a_1^2 + a_2^2 + . + a_n^2) \times (b_1^2 + b_2^2 + . + b_n^2)$$
$$\geq (a_1 b_1 + a_2 b_2 + . + a_n b_n)^2 \quad (5)$$

When all a_i are set to t_i, and all b_i are set to 1, we can get (6).

$$t_1^2 + t_2^2 + \ldots + t_n^2 \geq \frac{(t_1 + t_2 + \ldots + t_n)^2}{n} \quad (6)$$

Because $t = t_1 + t_2 + \ldots + t_n$, (6) can be converted to (7).

$$\sum_{i=1}^{n} t_i^2 \geq \frac{t^2}{n} \quad (7)$$

Then we plus t and divide by 2 at both sides of (7) and get (8):

$$\frac{1}{2}(\sum_{i=1}^{n} t_i^2 + t) \geq \frac{1}{2}(\frac{t^2}{n} + t) \quad (8)$$

So the lower boundary of the constructing time consume is when n=t. In other words, all the NODEs' types are different from each other, such as the string of "ABCDEF12345HIGK". At that time, the lower boundary is t (or n).

Upper boundary:

Because t is a constant value, so the upper boundary of (4) is depended on the (9)'s upper boundary.

$$\sum_{i=1}^{n} t_i^2 \quad (9)$$

And (9) can be converted to (10).

$$\sum_{i=1}^{n} t_i^2 = (\sum_{i=1}^{n} t_i)^2 - 2 \sum_{i=1,j=1}^{n} t_i t_j \quad (10)$$

Because $t = t_1 + t_2 + \ldots + t_n$, (10) can be converted to (11).

$$\sum_{i=1}^{n} t_i^2 = t^2 - 2 \sum_{i=1,j=1}^{n} t_i t_j \quad (11)$$

Then we plus t and divide by 2 at both sides of (11) to get (12).

$$\frac{1}{2}(\sum_{i=1}^{n} t_i^2 + t) \leq \frac{1}{2}t^2 - \sum_{i=1,j=1}^{n} t_i t_j + \frac{t}{2} \quad (12)$$

In reality situation, the value of $t_i(t_j)$ can be 0. And the most extreme situation (worst case) is that the NODE link only has two acceleration NODEs at the begin and end, such as "AAAAAAAAAB". The '1:A' and '10:B' are acceleration NODEs. And the most extreme situation can be generally regarded as: The t-lengthed NODE link is divided into 2 (n=2) parts, the first part's length t_1 is 1, and the second part's length t_2 is t-1. Put t_1 and t_2 into (12), we can get the upper boundary as (13).

$$\frac{1}{2}(\sum_{i=1}^{n} t_i^2 + t) \leq \frac{1}{2}t^2 - \frac{1}{2}t + 1 \quad (13)$$

Sum up:

The constructing time consume can be represented by (4) approximately. The exact constructing time consume is more complex and bigger than (4). Because the acceleration NODE doesn't ensure to make all the SUFFIXs' p point to leaves except the SUFFIX of itself, such as the '8:x' in the string of "abcbxabxy". When '8:x' is added, the suffix "abx" link to leaf, the suffix "bx" is still on the edge of improved suffix tree,but the (4) regards "bx" links to leaf at the same time. So when the '9:y' is added, the (4) will miss to calculate "bxy", which makes the exact constructing time consume bigger than the (4).

The (4) is an n-dimensional problem, which is not suitable for human to understand. So we use the knowledge of inequality to reduce the n-dimensional problem into an one-dimensional problem. And finally the scale result is like (14) showing:

$$t \leq \frac{1}{2}(\sum_{i=1}^{n} t_i^2 + t) \leq \frac{1}{2}t^2 - \frac{1}{2}t + 1 \tag{14}$$

According to table 3, we can draw the line chart between length and building tick as figure 10:

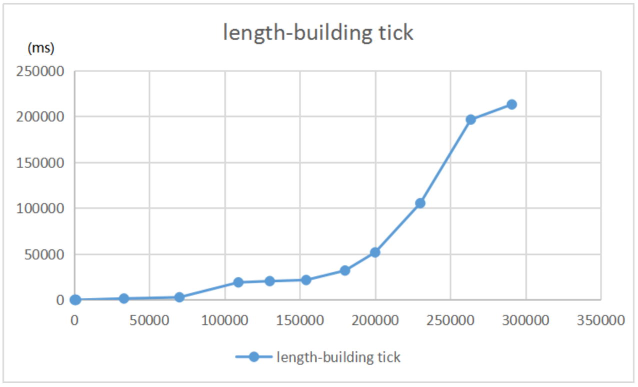


FIGURE 10. The line chart between length and building tick.

The figure 10 illustrates that the constructing time consume is in a growth trend between O(n) and O(n^2).

And we can draw the line chart between length and building index as figure 11:

The figure 11 illustrates that the constructing time consume of each basic unit is in a growth trend between O(n) and O(n^2) too. And in Prakash's paper [28], they got the similar line chart like figure 12 shown:

In conclusion, the time complexity of the improved suffix tree's constructing process can not strictly maintain at O(n) level. It depends on the distribution of the acceleration NODEs mainly. And the final range is between O(n) and O(n^2).

C. FAST PREDICTION FORMULA

We have proved that the constructing time consume of the improved suffix tree is between O(n) and O(n^2), actually,

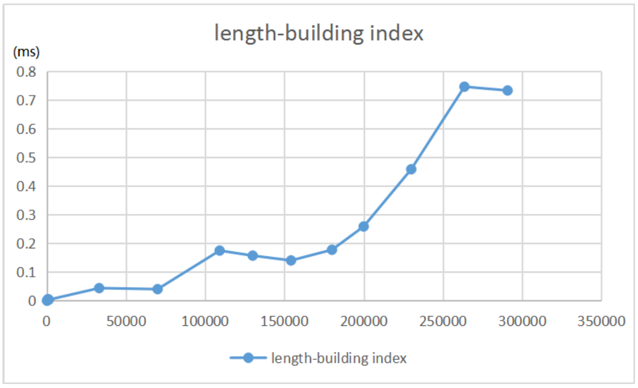


FIGURE 11. The line chart between length and building index.

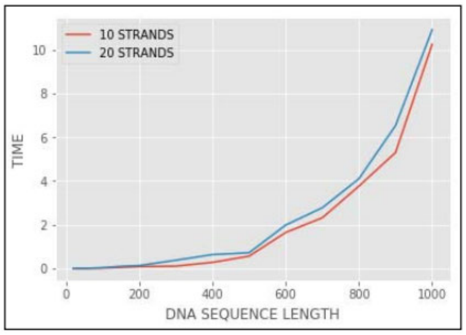


FIGURE 12. Prakash's line chart between length and constructing time consume.

it is more closer to the O(n^2). By the previous experiment data in table 3, we know that 290809 sentences (28.0 MB) need 213048ms (nearly 3.5 minutes). As the source data becomes bigger, it is necessary to predict the time consume before constructing. If the prediction time is far beyond the time limit, we can stop constructing in time.

The prediction formula is on the basis of the (4) and like (15) shown:

$$P = \frac{1}{2}(\sum_{i=1}^{n} T_i^2 + t) \times \frac{D}{M} \tag{15}$$

Let's explain the (15) in detail.

First, we use debut NODE (T_i) to replace acceleration NODE (t_i) in (4). Because we can get all the positions of debut NODEs before constructing in a very short time.

Second, for compensating the error caused by the above replacing, we divide (4) by M which represents the ratio of exact construction time consume to the prediction value calculated by debut NODE. We can explain M more vividly with the help of figure 13:

In figure 13, the red area represents the value calculated by debut NODE. The green area and blue area represent the value calculated by acceleration NODE. And the green, blue and yellow ares represent the exact constructing time consume. So the M can be regarded as the area ratio of red area to the sum of green, blue and yellow ares.

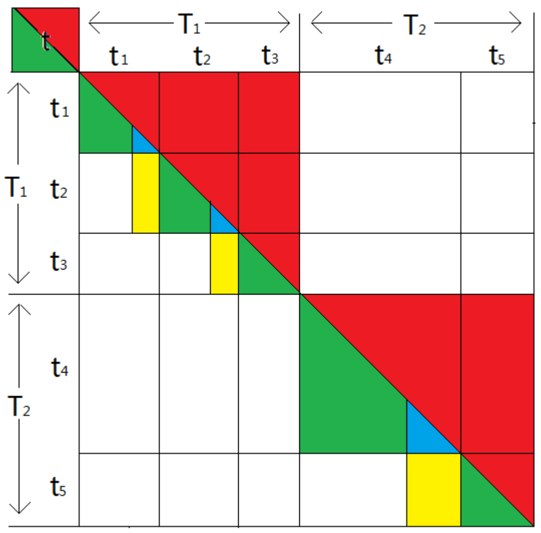


FIGURE 13. The relationship between the exact constructing time consume and calculated values.

Third, when calculate the (4), we regard each explicit situation and implicit situation as the same time unit as Ukkonen's paper [10] did. In fact, the average time use of once explicit situation is longer than that of implicit. And the average time use of once comparing is related with the hardware condition. So we use D to represent the two differences between exact constructing time consume and (4).

In experiment, the T represents the actual number of comparing. We can get D by dividing building tick with T. And we can get M by divide (4) with T. The actual measured values are like table 4 shown:

TABLE 5. The actual measured values.

num	length	D	M	formula(4)
1	290809	2.94057E-05 ms	1.76198638	12765797260
2	263553	2.89428E-05 ms	1.392410902	9456219426
3	230001	2.85465E-05 ms	1.719691816	6348884826
4	200001	2.76201E-05 ms	2.414118861	4523789826
5	180001	2.75525E-05 ms	3.281628317	3807059826
6	154164	2.61971E-05 ms	4.203176136	3473297459
7	130001	2.84814E-05 ms	2.471602725	1773599550
8	109151	3.16899E-05 ms	1.227064534	738796580.5
9	70001	7.39513E-05 ms	1.991315018	76258348.5
10	33133	0.00029633 ms	2.99670678	14764597.5
11	1343	2.09997E-05 ms	1.0000021	238099.5
12	963	1.29511E-05 ms	1.000001295	386071.5
13	797	1.68345E-05 ms	1.000885913	237818.5
14	406	0.000377786 ms	1.000188893	2647.5
15	260	0 ms	1.001930502	259.5

According to table 5, we can see when the length is over 100000, the value of D is basically fixed near 3×10^{-5}. And the maximum of M is just 4.203176136. Compared with 100000-lengthed source data, we can regard the value of $\frac{D}{M}$ as a constant represented by C. So the (15) can be transformed into (16).

$$P = \frac{1}{2}(\sum_{i=1}^{n} T_i^2 + t) \times C \tag{16}$$

According to our experiment situation, When we set C as 10^{-5}, the relationship between prediction constructing time consume and actually recorded time consume is like figure 14 shown:

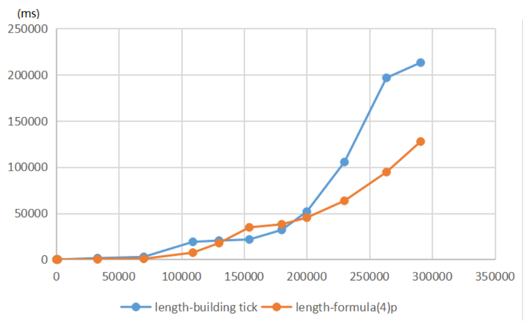


FIGURE 14. Length-building tick|length-P.

According to figure 14, we can see that the difference between the predicted value and the accurate value is not more than twice. We can finish the prediction of 290809-lengthed source data in 47959 ms just after compressing, which is far less than 213048 ms. Because the compressing process is linear time complexity and the constructing process is quadratic time complexity, when the source data becomes bigger, the effect of saving time will be more obvious.

V. REUSING AND VISUALIZING

A. REUSING (TRAVELING)

In the IV.A, we illustrate two differences between improved suffix tree and traditional UKK suffix tree. Those two differences don't reduce constructing time consume, but change the traveling (reusing) time consume form quadratic time to linear time essentially.

For getting the hierarchical inclusion relationship like figure 2 shown, we need to finish three steps.

Step 1, find all the end position of suffixes on the improved suffix tree. such as for the string of "cacao", if we want to get all 5 suffixes end position on UKK suffix tree, once DFS or BFS is necessary, whose time complexity is O(n^2). In contrast, by improved suffix tree, we just need to do once linear-timed traversing on the SUFFIX LINK.

Step 2, find common prefixes for all suffixes. For UKK suffix tree, the DPS or BFS in the Step 1 can achieve this aim at the same time, whose time complexity is O(n^2). For improved suffix tree, on the basis of SUFFIX LINK and reverse relationship, we can get all the end position and length of common prefixes in O(n) time complexity totally like figure 16 shown:

Step 3, combine the common prefixes to get COMBINE LINK. The common prefixes whose begin positions are continuous and end positions are the same can be combined into COMBINE structure. We denote the COMBINE structure as C(b,e,n,i,l). The b and e represent the begin and end positions of the substring respectively. The n represents the pointer to the neighbouring COMBINE structure. The i represents

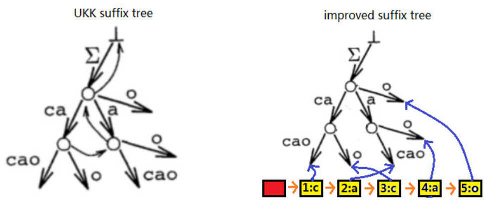


FIGURE 15. Find all the end position of suffixes.

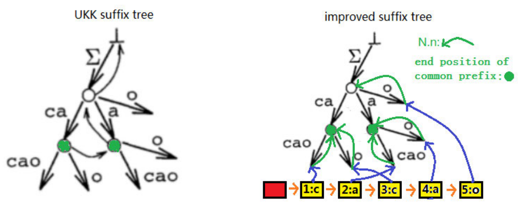


FIGURE 16. Find common prefixes for all suffixes.

identification of COMBINE which depends on the length and end position of the according substring. The l represents the length of the longest prefix.

To get COMBINE LINK, the UKK suffix need to do once DPS or BFS at least whose time complexity is O(n^2). For improved suffix tree, by traversing the SUFFIX LINK, we can ensure that the begin positions of common prefixes are continuous. After combine the SUFFIXs into COMBINEs, through a simple bubble sort, we can the COMBINEs whose l are the same into the same i. The process to get the COMBINE LINK (blue part) of "cacao" is like figure 17 shown:

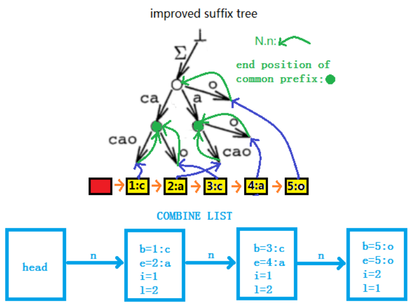


FIGURE 17. Combine the common prefixes to get COMBINE LINK.

We denote the variables of traversing as table 6:

The general traversing process can be abstracted into algorithm 3:

When we start to traverse an improved suffix tree, L is equal to 1, the PSB points to the next position of SUFFIX HEAD, PSE points to the end position of SUFFIX LINK. After traversing, we get sorted COMBINE LINK. At this moment, we can chose any COMBINE whose i appears in the COMBINE LINK more than once to expand its hierarchical relationship further. Now L is equal to 2, the PSB points to the chosen COMBINE's b, PSE points to the chosen COMBINE's e. We can do the above operations iteratively to

TABLE 6. The variables of traversing.

Num	Name	Type	Definition
1	t	int	the total length of substring.
2	E(b,e)	EDGE	the edge of improved suffix tree, and b represents the begin order of the edge, e represents the end order of the edge.
3	PSB	SUFFIX *	the pointer to the begin position of substring on the SUFFIX lINK.
4	PSE	SUFFIX *	the pointer to the end position of substring on the SUFFIX lINK.
5	PS	SUFFIX *	the ergodic pointer to a SUFFIX structure.
6	PES	SUFFIX *	the end position of corresponding suffix on the suffix tree.
7	PEP	SUFFIX *	the end position of corresponding common prefix on the suffix tree.
8	PC	COMBINE *	the ergodic pointer to a COMBINE structure.
9	PCH	COMBINE *	the pointer to head node of COMBINE LIST.
10	i	int	the number of cycle time for common condition.
11	L	int	total lawyer number of hierarchical inclusion relationship.
12	DL	int	the different part's length of corresponding suffix.
13	CPL	int	the common prefix's length of corresponding suffix.
13	CPL	int	the common prefix's length of corresponding suffix.
14	TL	double	the l of PS's former SUFFIX.

get the n-lawyered hierarchical inclusion relationship. (when L=3, when can get the hierarchical inclusion relationship like figure 2 shown)

Then we do experiments to record the traversing (reusing) time consume for a 290809-lengthed source data, the recorded data is like table 7 shown:

TABLE 7. The recorded and calculated values for logicalize.

num	length	b	e	L	traversing tick
1	290809	1	290807	1	48 ms
2	263553	1	263553	1	48 ms
3	230001	1	230001	1	33 ms
4	200001	1	200001	1	34 ms
5	180001	1	180001	1	30 ms
6	154164	1	154164	1	28 ms
7	130001	1	130001	1	22 ms
8	109151	45014	154164	2	19 ms
9	70001	1	70001	1	11 ms
10	33132	45014	78145	3	5 ms
11	1342	15216	16557	2	0 ms
12	962	14942	15903	2	0 ms
13	796	39403	40198	2	0 ms
14	405	14526	14930	2	0 ms
15	269	44774	45032	2	0 ms

The 11th-15th data's traversing tick is 0 ms, the reason is that the traversing process is so fast and highest recordable accuracy is millisecond. And we make a chart line about length and traversing tick like figure 18 shown:

According to figure 8, it is obvious that as the length becomes bigger, the traversing tick increases at the same time

Algorithm 3 Traversing()

```
1:for(PS=PSB,t=0;PS!=PSE->n;PS=PS->n)
2:   t++
3:for(PS=PSB,t=0;PS!=PSE->n;PS=PS->n)//step 1
4:   PES=PS->p
5:   for(i=0,DL=0,PEP=PS->p;i<L;i++)//step 2
6:      DL=DL+(PS->e->e-PS->e->b)
7:      PEP=PEP->n
8:   CPL=t-DL-(PS->o-PSB->o+1)
9:   PS->l=CPL+PEP/0xffffff
10:TL=PSB->l;//step 3
11:PC1=new COMBINE
12:PC1->b=PSB->o;PC1->l=PSB->l;PCH->n=PC1
13:for(PS=PSB,t=0;PS!=PSE->n;PS=PS->n)
14:   if PS->l!=TL
15:      PC1->e=PS->o-1;TL=PS->l
16:      PC2=new COMBINE
17:      PC2->b=PS->o;PC2->l=PS->l;PC1->n=PC2
18:      PC1=PC2
19:do bubble sort the SUFFIX LINK by l
```

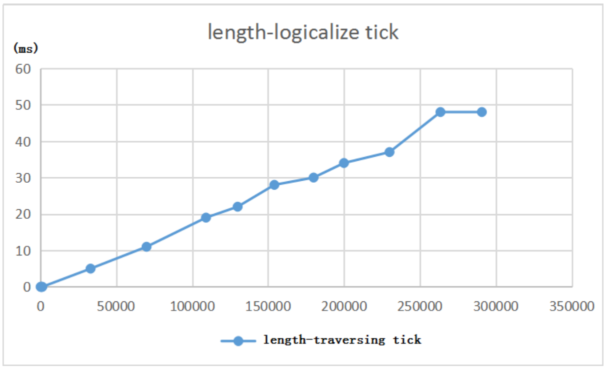


FIGURE 18. Combine the common prefixes to get COMBINE LINK.

by a linear type. And in Huisheng Zhu's paper [29], they traversed the suffix tree by DFS, the relationship between length average running time is like figure 19 shown:

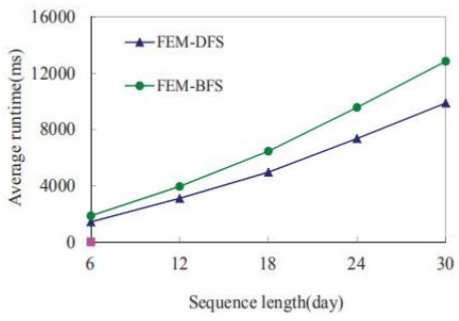


FIGURE 19. Combine the common prefixes to get COMBINE LINK.

It's obvious that the chart line in the figure 19 shows a trend of nonlinear growth. In contrast, on the basis of IV.A, our improved suffix tree can finish traversing by linear time.

B. VISUALIZING

In this subsection, we visualize the COMBINE LINK into 2D defrag picture like figure 1 shown, which then can be combined into 3D hierarchical inclusion relationship like figure 2 shown. The pseudo code of visualizing is like Algorithm 4 shown.

Algorithm 4 Visualizing()

```
  PEC represents pErgodicCombine;
 1:width=nodeLinkLength;
 2:if width%2==1
 3:   width=width+1
 4:height=width;memorySize=54+width*height*2
 5:pMemoryHead=malloc(sizeof(char)*
        (54+width*height*2))
 6:initializeHeadFile(width,height)
 7:pHeadFile=(unsigned char *)pHeadOverAll
 8:pInput=pMemoryHead
 9:i=0
10:while i<54
11:   pInput=pHeadFile;pInput++;pHeadFile
12:   i++
13:picWide=0;picHeight=0;debounce=0;before=0
14:gamut=65536;PEC=combineHead.pNext
15:while PEC!=NULL
16:   debounce=PEC->rankNumber/(double)colourType
        *gamut
17:   pInput=pMemoryHead+54
18:   i=1
19:   while i<=(width*height*2)
20:      picHeight=(i+1)/2/width+1;picWide=(i+1)/2%
        width
21:      if picWide>=PEC->blockBegin
         &&picWide<=PEC->blockEnd
         &&picHeight>=PEC->blockBegin
         &&picHeight<=PEC->blockEnd
           *pInput =(int)debounce
22:      i++,pInput++
23:   PEC=PEC->pNext
24:write pMemoryHead into file
```

The core idea of visualization is representing the i-equaled COMBINEs into same-coloured squares. And the diagonal direction represents the execution process of source code. The 3D hierarchical inclusion relationship is as the final result of our gray box to show the inner logic of source code.

We make an example for visualizing 290809 instruction into 3 lawyers.

When L=1, the PSB must points to the next node of COMBINE HEAD, and the PSE must points to the last node of COMBINE LINK. The 2D defrag picture shows the 1-29089 instructions inner logic. The same-coloured squares in the 2D defrag picture represent the identical sub-instructions.

The 2D defrag picture of 1-290809 instructions is like figure 20 shown:

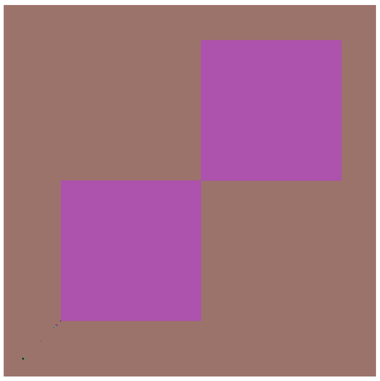


FIGURE 20. L=1 PSB=1th PSE=290809.

By figure 20, we can immediately realize that the most obvious code block of 1-29089 instructions is like the purple square shown, and this code block executes twice continuously. The lower purple square represents the 45014-154164 instructions, and the upper purple square represents the 154402-263553 instructions.

Then we expand the 45014-154164 instructions. Now L=2, PSB points to the 45014th node on the SUFFIX LINK, PSE points to the 154164th node on the SUFFIX LINK. After traversing, we can get new COMBINE LIST, and visualize the COMBINE LIST by the 2D defrag picture like figure 21 shown:

The 2D defrag picture of 45014-154164 instructions is like figure 21 shown:

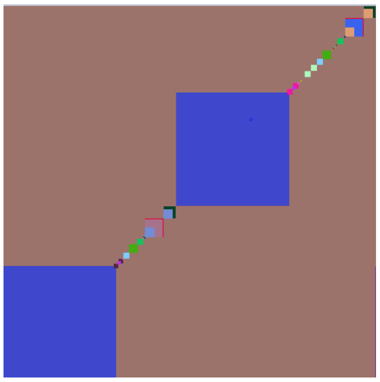


FIGURE 21. L=2 PSB=45014th PSE=154164.

We can iterate the results of figure 21 to expand the lower blue block. Now L=3, PSB points to the 45014th node on the SUFFIX LINK, PSE points to the 78145th node on the SUFFIX LINK. The 3th-lawyered 2D defrag picture like figure 22 shown:

Finally, we combine the figure 19, figure 20 and figure 21 into a 3D space. The final hierarchical inclusion relationship is like figure 23 shown:

By the hierarchical inclusion relationship, our gray box can show the inner logic of source instructions vividly.

There have been lots of similar visualizing researches. In Fontana's paper [29], they developed a debugger to show the inner logic of JAVA source code. Their visualizing result is like figure 24 shown:

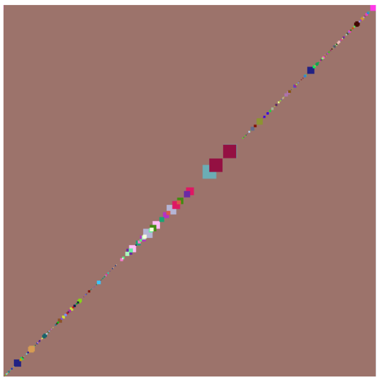


FIGURE 22. L=3 PSB=45014th PSE=78145.

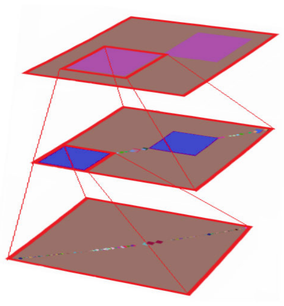


FIGURE 23. The final hierarchical inclusion relationship.

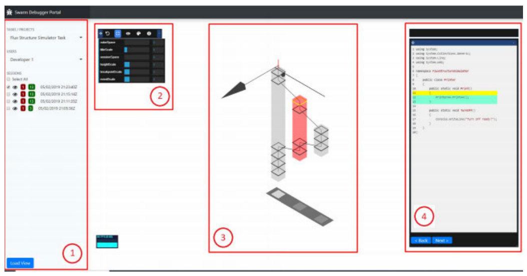


FIGURE 24. The visualizing result of Fontana's paper.

And the visualizing origin of Fontana's paper is like figure 25 shown:

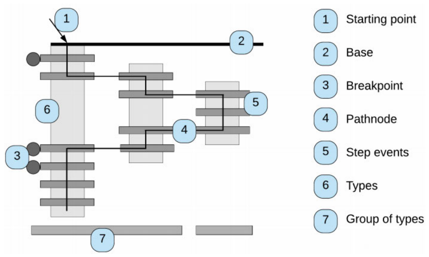


FIGURE 25. The visualizing origin of Fontana's paper.

Fontana's research did visualize the JAVA program excellently, but like figure 25 shown, its visualization origin is based on advanced language code and call stack, which is suitable for white-box not gray-box. In contrast, our gray-box can get the similar visualizing effect just by assembly instructions.

VI. CONCLUSION

Through collecting, compressing, constructing (improved suffix tree), traversing (reusing) and visualizing, our gray box finally realizes to show the inner logic of on line game's currency-related instructions. And the gray-box can be used to visualize the 32-bit PE structure program in a short time generally.

The detailed conclusions of each part can be summarized as the following four points:

1.By II, we prove that the collecting part can record aimed instructions without the jam of compression encryption.

2.By III, we prove that the compressing part can bring obvious acceleration to the later constructing part.

3.By IV, with inequality technique, we prove the time consume range of construction process, and propose a fast prediction formula.

4.By V, we prove that the improved suffix tree can finish traversing (reusing) in linear time and achieve similar effect with white-box testing.

VII. FUTURE WORK

In the future, we plan to simulate the human brain activity by improved suffix tree, it will be Epoch-making.

What's the source of human brain's logic? The answer is human memory. We can imagine as the memorable things are saved into a suffix tree and the trifles are abandoned. In other words, the memory is interrupted but can be reformed into linear type. And suffix tree can transform the data from left to right in real time. So we can use suffix tree to record human memory.

The thing that traditional suffix tree can't reach is that the traversing of any data needs $O(n^2)$ time complexity, as the amount of data becomes bigger and bigger with time going, The $O(n^2)$ time complexity is absolutely unacceptable. In other words, the simulated human brain will be an idiot as the time goes by.

But with the help of Algorithm 3, we can get all-layered memory logic in a linear time, which make the simulated human brain can react quickly and human likely. That is what we will do in the future.

REFERENCES

[1] L. Meng, M. Lu, B. Huang, and X. Xu, "Using relative complexity measurement which from complex network method to allocate resources in complex software System's gray-box testing," presented at the Int. Symp. Comput. Sci. Soc., Kota Kinabalu, Malaysia, Jul. 16–17, 2011.

[2] O. Loyola-Gonzalez, "Black-box vs. white-box: Understanding their advantages and weaknesses from a practical point of view," *IEEE Access*, vol. 7, pp. 154096–154113, Oct. 2019, doi: 10.1109/ACCESS.2019.2949286.

[3] X. Li, X. Wang, and W. Chang, "CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 2, pp. 101–114, Mar. 2014, doi: 10.1109/TDSC.2012.83.

[4] U. Sabir, F. Azam, S. U. Haq, M. W. Anwar, W. H. Butt, and A. Amjad, "A model driven reverse engineering framework for generating high level UML models from java source code," *IEEE Access*, vol. 7, pp. 158931–158950, Nov. 2019, doi: 10.1109/ACCESS.2019.2950884.

[5] Z. He, G. Ye, L. Yuan, Z. Tang, X. Wang, J. Ren, W. Wang, J. Yang, D. Fang, and Z. Wang, "Exploiting binary-level code virtualization to protect Android applications against app repackaging," *IEEE Access*, vol. 7, pp. 115062–115074, Jun. 2019, doi: 10.1109/ACCESS.2019.2921417.

[6] *UPX*. Accessed: Jan. 23, 2020. [Online]. Available: https://upx.github.io/

[7] O. Nguena Timo, D. Prestat, and F. Avellaneda, "Fault detection in timed FSM with timeouts by SAT-solving," presented at the 19th Int. Conf. Softw. Qual., Rel. Secur. (QRS), Sofia, Bulgaria, Jul. 22–26, 2019.

[8] A. Aggarwal and P. Jalote, "Monitoring the security health of software systems," presented at the 17th Int. Symp. Softw. Rel. Eng., Raleigh, NC, USA, Nov. 7–10, 2006.

[9] M. El Boujnouni, M. Jedra, and N. Zahid, "New malware detection framework based on N-grams and support vector domain description," presented at the 11th Int. Conf. Inf. Assurance Secur. (IAS), Marrakech, Morocco, Dec. 14–16, 2015

[10] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, Sep. 1995, doi: 10.1007/BF01206331.

[11] H. Zhu, L. Chen, J. Li, A. Zhou, P. Wang, and W. Wang, "A general depth-first-search based algorithm for frequent episode discovery," presented at the 14th Int. Conf. Natural Comput., Fuzzy Syst. Knowl. Discovery (ICNC-FSKD), Huangshan, China, Jul. 28–30, 2018.

[12] M. Burch, "Visual analysis of compound graphs," presented at the IEEE Symp. Vis. Lang. Hum.-Centric Comput. (VL/HCC), Cambridge, U.K., Sep. 4–8, 2016.

[13] T.-E. Wei, Z.-W. Chen, C.-W. Tien, J.-S. Wu, H.-M. Lee, and A. B. Jeng, "RePEF—A system for restoring packed executable file for malware analysis," presented at the Int. Conf. Mach. Learn. Cybern., Guilin, China, Jul. 10–13, 2011.

[14] B. Egger, Y. Cho, C. Jo, E. Park, and J. Lee, "Efficient checkpointing of live virtual machines," *IEEE Trans. Comput.*, vol. 65, no. 10, pp. 3041–3054, Oct. 2016, doi: 10.1109/TC.2016.2519890.

[15] H. Yang, D. Liu, Z. Zhao, and Y. Li, "Research and implementation of OEP search based on API-monitoring," presented at the Int. Conf. Comput. Sci. Service Syst., Nanjing, China, Aug. 11–13, 2012.

[16] *DrX, EFlag*. Accessed: Dec. 5, 2018. [Online]. Available: https://docs.microsoft.com/zh-cn/windows/win32/api/winnt/ns-winnt-context

[17] S. Sharmeen, Y. A. Ahmed, S. Huda, B. S. Kocer, and M. M. Hassan, "Avoiding future digital extortion through robust protection against ransomware threats using deep learning based adaptive approaches," *IEEE Access*, vol. 8, pp. 24522–24534, Jan. 2020, doi: 10.1109/ACCESS.2020.2970466.

[18] H. Jiang and S.-J. Lin, "A rolling hash algorithm and the implementation to LZ4 data compression," *IEEE Access*, vol. 8, pp. 35529–35534, 2020, doi: 10.1109/ACCESS.2020.2974489.

[19] T. Gong, X. Tan, and M. Zhu, "Malware detection via classifying with compression," presented at the 1st Int. Conf. Inf. Sci. Eng., Nanjing, China, Dec. 26–28, 2009.

[20] Y. Qiao, Y. Yang, L. Ji, C. Tang, and J. He, "A lightweight design of malware behavior representation," presented at the 12th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun., Melbourne, VIC, Australia, Jul. 16–18, 2013.

[21] R. Koschke, "Large-scale inter-system clone detection using suffix trees," presented at the 16th Eur. Conf. Softw. Maintenance Reeng., Szeged, Hungary, Mar. 27–30, 2012.

[22] K. Huang, Y. Ye, and Q. Jiang, "ISMCS: An intelligent instruction sequence based malware categorization system," presented at the 3rd Int. Conf. Anti-Counterfeiting, Secur., Identificat. Commun., Hong Kong, Aug. 20 22, 2009.

[23] *IDA*. Accessed: Jan. 1, 2020. [Online]. Available: https://www.hex-rays.com/products/ida/

[24] *Windbg*. Accessed: May 1, 2020. [Online]. Available: www.windbg.org/

[25] L. Huoyao and L. Gongshen, "On-line linear time construction of sequential binary suffix trees," *J. Syst. Eng. Electron.*, vol. 20, pp. 1104–1110, Oct. 2009.

[26] A. Zaky and R. Munir, "Full-text search on data with access control using generalized suffix tree," presented at the Int. Conf. Data Softw. Eng. (ICoDSE), Denpasar, Indonesia, Oct. 26–27, 2016.

[27] S. Gupta, R. Prasad, and S. Yadav, "Searching gapped palindromes using inverted suffix array," presented at the IEEE Int. Conf. Comput. Intell. Commun. Technol., Ghaziabad, India, Feb. 13–14, 2015.

[28] S. Prakash, H. Agarwal, U. Agarwal, P. Biswas, and S. D. Jaypee, "Discovering motifs in DNA sequences: A suffix tree based approach," presented at the IEEE 8th Int. Advance Comput. Conf. (IACC), Greater Noida, India, Dec. 14–15, 2018.
[29] E. A. Fontana and F. Petrillo, "Visualizing sequences of debugging sessions using swarm debugging," presented at the IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC), Montreal, QC, Canada, May 25–26, 2019.

DONGLIN WANG (Member, IEEE) was born in Zhangjiakou, China, in 1994. He received the B.S. degree in software engineering from the China University of Petroleum, in 2017. He is currently pursuing the M.S. degree in computer application technology with the Inner Mongolia University of Technology.

He is also an Assistant Lecturer with the Division for Digital Electronics Technology and Analog Electronics Technology. He holds two patents and one software copyright. His main research interests include software reverse engineering, computer vision, computer simulation, information processing, and intelligent control.

JIANDONG FANG was born in Dalian, China, in 1966. She received the B.S. degree from Inner Mongolia University, the M.S. degree from the Taiyuan University of Technology, and the Ph.D. degree in computer application technology from Tianjin University.

Since 2010, she has been a Professor with the Institute of Information Engineering, Inner Mongolia University of Technology. Since 2016, she has been the Director of the Inner Mongolia Key Laboratory of Perceptual Technology and Intelligent Systems. She has authored four books and more than 60 articles. She holds more than ten inventions. Her research interests include information processing and intelligent control, intelligent perception and analysis decision, and embedded intelligent systems.