

于 Dalvik 指令 `add-int/2addr v0,v1`,在污点传播逻辑下表达为 $Taint_{v_0} = Taint_{v_0} | Taint_{v_1}$,其传递函数 f_s 将污点变量 v_0 从输入状态 $[100\dots 0]$ 转化为 $[110\dots 0]$.对该虚拟机指令进行污点跟踪,需要额外完成以下 4 步:首先,通过 `SET_TAIN_TFP(r10)` 保存变量污点的内存地址存入寄存器;然后,通过 `GET_VREG_TAIN(r3,r3,r10)` 与 `GET_VREG_TAIN(r2,r2,r10)` 获取变量污点值;然后,通过 `orr r2,r3,r2` 计算更新后污点值;最后,通过 `SET_VREG_TAIN(r2,r9,r10)` 将该污点值存入内存.对于 Dalvik 指令 `add-int/lit8 v2,v2,0x1`,在污点传播逻辑下表达为 $Taint_{v_2} = Taint_{v_2} | clean$.其传递函数 f_s 将污点变量 v_2 从输入状态 $[001\dots 0]$ 转化为 $[001\dots 0]$.对于由该 3 条语句组成的循环体,其传递函数 f_s 将污点变量 v_0 从输入状态 $[100\dots 0]$ 转化为 $[110\dots 0]$,将污点变量 v_1 从输入状态 $[010\dots 0]$ 转化为 $[110\dots 0]$,将污点变量 v_2 从输入状态 $[001\dots 0]$ 转化为 $[001\dots 0]$.

3 污点传播优化方法

通过上述污点传播分析,污点跟踪代码如下特征.

- 其一,在程序逻辑向污点传播逻辑转化后,污点传播逻辑对应的本地指令集变小.最主要包含以下 4 种: `setTaintClear(*cUnit,rlDest)` 方法和 `setTaintClearWide(*cUnit,rlDest)` 方法编译为 `mov` 指令; `loadTaintDirect(*cUnit,rlSrc,reg1)` 方法和 `loadTaintDirectWide(*cUnit,rlSrc,reg1)` 方法编译为 `ldr` 指令; `storeTaintDirect(*cUnit,rlDest,reg1)` 方法和 `storeTaintDirectWide(*cUnit,rlDest,reg1)` 方法编译为 `str` 指令; `opRegRegReg(cUnit,kOpOr,taint1,taint1,taint2)` 方法编译为 `orr` 指令.因此,污点传播本地指令集主要包括 `ldr`,`str`,`mov` 和 `orr` 指令.相对 `arm` 指令集是非常小的一个子集.
- 其二,程序逻辑向污点传播逻辑转化后,污点传播代码具备了一些新的特征.例如, $a=a+b$ 和 $a=a-b$ 虽然在程序逻辑下计算出的 a 数值不一样,但在污点传播逻辑下, a 的污点值是一样的.又如 `for (i=0; i<100;i++) a=a+b`,在循环执行时, a 的数值是一直变化的,但在污点传播逻辑下, a 的污点值是不变的.
- 其三,在插入的污点跟踪代码中,存在许多内存和寄存器之间存取数据的指令.比如,要完成 $a=a+b$,就需要在内存中取 a 的污点值,取 b 的污点值;同时,完成运算后,还要将 a 的污点值保存到内存.

针对污点传播代码的特征(上述 3 种特征),对经典流数据流算法进行改进,提出了 3 种适应污点传播代码特性、针对性更强、效率更高的优化算法.主要改进体现在如下两个方面.

- 其一,指令数据结构的改进.在 dalvik 即时编译过程中,程序指令一般被组织为一条双向链表.而本文将程序指令和污点传播指令组织为一条双向链表,污点传播指令又组织为另一条双向链表,其开销只是在指令数据结构中加入 `*TAIN_TPRE` 和 `*TAIN_TNEXT`.
- 其二,数据流算法的改进.由于污点传播指令单独组织,优化算法只需遍历插入的污点传播代码,无需遍历所有的程序指令.从而算法效率更高.

对于每一种优化算法,都是依据上述污点传播代码的特征进行优化的.同时,考虑在即时编译器下以最快速度进行污点跟踪优化,优化算法在污点传播分析的同时进行污点跟踪优化,而不像传统编译器将数据流分析和代码优化算法分开.

3.1 冗余污点存取消除

冗余污点存取消除是指消除冗余的污点在寄存器和内存之间移动.以虚拟机指令 `add-int/2addr v0,v1` 和 `add-int/2addr v1,v0` 为例.Dalvik 即时编译器将其转化为表 1 第 1 列所示 LIR(已将内存读指令集中前置),其中, r_5 表示栈针位置,内存数据偏移量以字节为单位.对于第 1 条虚拟机指令 `add-int/2addr v0,v1`,首先,由于污点值交叉保存在内存,将变量 v_0 和 v_1 从偏移位置 0 和 8 处读入寄存器 r_1 和 r_2 ,并将其污点值从偏移位置 4 和 12 处读入寄存器 r_0 和 r_3 .然后,将操作数及其污点进行数值运算 `adds r1,r1,r2` 和污点传播运算 `orr r0,r0,r3`.最后,将数值和污点值写回内存.同理,对于第 2 条指令完成相同动作.由于即时编译器仅简单地将虚拟指令翻译为 LIR 指令,因此产生的代码较为庞大和冗余.通过对虚拟机指令的污点传播分析,可得出第 1 条指令和第 2 条指令变量值及其污点值在内存中位置一致,此时,第 2 条指令不需要再次读取操作数及其污点,且第 1 条指令不需要立即将

数值运算结果和污点传播运算结果保存到内存,可在第 2 条指令执行完成后再保存相应数据值及其污点值.删除冗余的污点存取代码后优化结果见表 1 第 2 列.

Table 1 Example of redundant taint load-store elimination
表 1 冗余污点存取消除示例

Taint without optimization	Taint with optimization
(0x0010): <i>ldr</i> r_1 , [r_5 ,#0]	(0x0010): <i>ldr</i> r_1 , [r_5 ,#0]
(0x0012): <i>ldr</i> r_0 , [r_5 ,#4]	(0x0012): <i>ldr</i> r_0 , [r_5 ,#4]
(0x0014): <i>ldr</i> r_3 , [r_5 ,#12]	(0x0014): <i>ldr</i> r_3 , [r_5 ,#12]
(0x0016): <i>ldr</i> r_2 , [r_5 ,#8]	(0x0016): <i>ldr</i> r_2 , [r_5 ,#8]
(0x0018): <i>ldr</i> r_4 , [r_5 ,#12]	-
(0x001a): <i>adds</i> r_1 , r_1 , r_2	(0x0018): <i>adds</i> r_1 , r_1 , r_2
(0x001c): <i>ldr</i> r_0 , [r_5 ,#4]	-
(0x001e): <i>ldr</i> r_3 , [r_5 ,#12]	-
(0x0020): <i>orr</i> r_0 , r_0 , r_3	(0x001a): <i>orr</i> r_0 , r_0 , r_3
(0x0022): <i>movs</i> r_1 , r_1	-
(0x0024): <i>str</i> r_1 , [r_5 ,#0]	-
(0x0026): <i>str</i> r_0 , [r_5 ,#4]	-
(0x0028): <i>adds</i> r_2 , r_2 , r_1	(0x001c): <i>adds</i> r_2 , r_2 , r_1
(0x002a): <i>ldr</i> r_4 , [r_5 ,#12]	-
(0x002c): <i>ldr</i> r_7 , [r_5 ,#4]	-
(0x002e): <i>movs</i> r_7 , r_0	-
(0x0030): <i>orr</i> r_4 , r_4 , r_7	(0x001e): <i>orr</i> r_3 , r_3 , r_0
(0x0032): <i>movs</i> r_2 , r_2	-
(0x0034): <i>str</i> r_2 , [r_5 ,#8]	-
(0x0036): <i>str</i> r_4 , [r_5 ,#12]	-
(0x0038): <i>str</i> r_4 , [r_5 ,#12]	(0x0020): <i>str</i> r_3 , [r_5 ,#12]
(0x0040): <i>str</i> r_2 , [r_5 ,#8]	(0x0022): <i>str</i> r_2 , [r_5 ,#8]
(0x004a): <i>str</i> r_0 , [r_5 ,#4]	(0x0024): <i>str</i> r_0 , [r_5 ,#4]
(0x004c): <i>str</i> r_1 , [r_5 ,#0]	(0x0026): <i>str</i> r_1 , [r_5 ,#0]

污点存取冗余是由于在污点传播代码中有大量无用的内存存取指令(数量几乎占插入代码的 70%),通过分析污点在内存与寄存器之间移动的特性,可实现冗余污点存取消除.在区分必然别名和寄存器值不被破坏的情况下,对于污点存取指令,分以下 4 种情况讨论.

- (1) 读后读(read after read,简称 RAR):数据污点值读取后又再次读取.对于此种情况,如果污点数值在第 1 次读取内存与第 2 次读取内存之间不存在对该污点值的写入指令,则可以删除后一条读取内存指令.
- (2) 写后写(write after write,简称 WRW):数据污点值写入内存后又再次写入内存.对于此种情况,如果污点数值在第 1 次写入内存与第 2 次写入内存之间对该污点值的读入指令,则可以删除前一条污点值的写入指令.
- (3) 读后写(read after write,简称 RAW):数据污点值在读取后写入内存.对于此种情况,如果在读取和写入内存之间不存在该数据污点值的传播点和污染点,则可将读取和写入内存指令删除;如果在读取和写入内存之间存在对该数据污点值的传播点但不存在污染点,则可将写入内存指令删除.
- (4) 写后读(write after read,简称 WRA):数据污点值写入内存后读取.对于此情况,则可将读取指令删除.

其具体算法如算法 1 所示.

算法 1. 冗余污点存取代码消除.

输入:编译单元**cUnit*,路径污点跟踪首指令**headLIR*,路径污点跟踪尾指令**tailLIR*.

输出:编译单元**cUnit*.

1. **for** (*currentLIR*=*TAINT_PREV_LIR*(*t_tailLIR*);
 currentLIR!=*t_headLIR*; *currentLIR*=*TAINT_PREV_LIR*(*currentLIR*))
2. { **if** (!(*EncodingMap*[*currentLIR*→*opcode*].*flags* & (*IS_LOAD*|*IS_STORE*))) *continue*;
3. //判断当前污点传播指令类型是否为内存存取类型,若否,则继续循环
4. **for** (*moveLIR*=*TAINT_PREV_LIR*(*currentLIR*);

```

    moveLIR!=t_headLIR;moveLIR=TAINT_PREV_LIR(moveLIR)
5.  { if (!(EncodingMap[moveLIR→opcode].flags & (IS_LOAD|IS_STORE))) continue;
6.    //判断当前污点传播指令类型为内存存取类型,若否,则继续循环
7.    if (moveLIR→operands[0]==currentLIR→operands[0] &&
        moveLIR→aliasInfo==currentLIR→aliasInfo)
8.    //互为别名,操作数本地寄存器一致且未被破坏
9.    { if ((isCurrentLIRLoad && isMoveLIRLoad)||(!isCurrentLIRLoad && isMoveLIRLoad))
10.     { moveLIR→flags.isNop=true;}
11.     //将后一读取指令删除
12.     else if (!isThisLIRLoad && !isCheckLIRLoad)
13.     { CurrentLIR→flags.isNop=true;}
14.     //将前一污点写入指令删除
15.     else if (isThisLIRLoad && !isCheckLIRLoad)
16.     { if (currntTaintMask==moveTaintMask)
17.      moveLIR→flags.isNop=true;}
18.     //将后一污点写入指令删除}
19.     } end if
20.   } //end for
21. } //end for

```

3.2 污点重复计算消除

污点重复计算消除是指消除污点值的重复计算.同样以虚拟机指令 `add-int/2addr v0,v1` 与 `add-int/2addr v1,v0` 为例,执行第 1 条 Dalvik 指令后, v_0 污点值更新为 $Taint_{v_1} | Taint_{v_0}$;执行第 2 条 Dalvik 指令后, v_1 污点更新为 $Taint_{v_1} | Taint_{v_0}$.由污点传播分析可知,两条虚拟机指令的污点运算值是相等的.可将表 1 第 2 列编号为 0x001e 的指令 `orr r0,r0,r3` 替换成 `movs r3,r0`,以提高指令执行速度.

对于污点重复计算消除,由于其程序逻辑中复杂的数值计算都将转化为污点传播逻辑中的或运算,因而会产生大量重复的计算表达式.针对这种污点传播代码的特性,通过分析此种污点计算代码的特性,给出污点重复计算消除算法.其实现的基本思想是:对于给定形如 `orr r0,r0,r1` 和 `orr r1,r1,r0` 表达式,若其间不存在其他对 r_0 与 r_1 的污染点,则可将后一表达式替换为执行速度更快的指令 `mov r1,r0`.

其具体算法如算法 2 所示.

算法 2. 污点重复计算消除.

输入:编译单元*cUnit,路径污染跟踪首指令*headLIR,路径污点跟踪尾指令*tailLIR.

输出:编译单元*cUnit.

```

1. for (currentLIR=TAINT_PREV_LIR(tailLIR);
    currentLIR!=t_headLIR;currentLIR=TAINT_PREV_LIR(currentLIR))
2.  { if (!(EncodingMap[currentLIR→opcode].flags & (IS_ORR))) continue;
3.    //判断当前污点传播指令类型是否为 ORR 类型
4.    for (moveLIR=TAINT_PREV_LIR(currentLIR);moveLIR!=t_headLIR;moveLIR=PREV_LIR(moveLIR))
5.    { if (!(EncodingMap[moveLIR→opcode].flags & (IS_ORR))) continue;
6.      //判断当前污点传播指令类型是否为 ORR 类型
7.      if (moveLIR→operands[0]==currentLIR→operands[0] &&
          moveLIR→operands[2]==currentLIR→operands[2])
8.      //操作数本地寄存器一致

```



```

9.      {newLIR=dvmCompilerRegCopyNoInsert(cUnit,moveLIR→operands[0],
                                         moveLIR→operands[2]);
10.     dvmCompilerInsertLIRAfter(moveLIR,newLIR);
11.     moveLIR→flags.isNop=true;}
12.     //产生一条 mov 复制指令并插入,同时,将后一 orr 运算指令删除
13.     else if (moveLIR→operands[0]==currentLIR→operands[0]||
              moveLIR→operands[0]==currentLIR→operands[2])
14.         break;
15.         //操作数在该处污染,结束当前循环,在外循环中检查下一条 currentLIR
16.     } //end for
17. } //end for

```

3.3 循环不变污点传播代码外提

循环是程序中不可缺少的一种控制结构,因为循环中的代码要重复执行,所以对循环代码的污点优化效果十分明显.循环不变污点传播代码外提,是指将循环体中反复进行污点值运算但污点值不变的代码提取到循环体必经后置基本块.仍然以第 2 节的 dalvik 代码为例,即时编译器探测到此循环并选择 `add-int/2addr v0,v1` 和 `add-int/lit8 v2,v2,0x1` 和 `if-lt v2,v3,:goto_0` 为热点路径,并将其编译为 7 个基本块,其中,关键基本块 Code Block 与 Normal Chaining Cell 代码见表 2 第 2 列.在程序逻辑下,变量 a 的数值是循环变化的,但污点传播逻辑下,变量 a 的污点值是循环不变的.同理,对于循环归纳变量,其污点值也是不变的.因此,可将 `ldr r0,[r5,#4]`,`ldr r3,[r5,#12]`,`ldr r7,[r5,#28]`,`orr r0,r0,r3` 等污点操作指令提取到循环必经后置节点,见表 2 第 3 列.

Table 2 Example of loop invariant taint propagation code motion

表 2 循环不变污点传播代码外提示例

	Taint without optimization	Taint with optimization
Code block	(0x0010): <code>ldr r1,[r5,#0]</code>	(0x0010): <code>ldr r1,[r5,#0]</code>
	(0x0012): <code>ldr r0,[r5,#4]</code>	-
	(0x0014): <code>ldr r3,[r5,#12]</code>	-
	(0x0016): <code>ldr r2,[r5,#8]</code>	(0x0012): <code>ldr r2,[r5,#8]</code>
	(0x0018): <code>ldr r4,[r5,#24]</code>	(0x0014): <code>ldr r4,[r5,#24]</code>
	(0x001a): <code>adds r1,r1,r2</code>	(0x0016): <code>adds r1,r1,r2</code>
	(0x001c): <code>ldr r7,[r5,#28]</code>	-
	(0x001e): <code>ldrb r8,[r6,#50]</code>	(0x0018): <code>ldrb r8,[r6,#50]</code>
	(0x0022): <code>orr r0,r0,r3</code>	-
	(0x0026): <code>adds r4,r4,#1</code>	(0x001a): <code>adds r4,r4,#1</code>
	(0x0028): <code>cmp r8,#0[0]</code>	(0x001c): <code>cmp r8,#0[0]</code>
	(0x002c): <code>str r7,[r5,#28]</code>	-
	(0x002e): <code>str r4,[r5,#24]</code>	(0x001e): <code>str r4,[r5,#24]</code>
	(0x0030): <code>str r0,[r5,#4]</code>	-
	(0x0032): <code>ldr r9,[r5,#32]</code>	(0x0020): <code>ldr r9,[r5,#32]</code>
	(0x0036): <code>str r1,[r5,#0]</code>	(0x0024): <code>str r1,[r5,#0]</code>
(0x0038): <code>bne PC Reconstruction</code>	(0x0026): <code>bne PC Reconstruction</code>	
(0x003a): <code>cmp r4,r9</code>	(0x0028): <code>cmp r4,r9</code>	
(0x003c): <code>blt 0x0010</code>	(0x002a): <code>blt 0x0010</code>	
(0x0040): <code>b 0x004c</code>	(0x002e): <code>b 0x003a</code>	
Normal chaining cell	(0x004c): <code>b 0x0050</code>	(0x003a): <code>b 0x003e</code>
	(0x004e): <code>orrs r0,r0</code>	(0x003c): <code>orrs r0,r0</code>
	-	(0x003e): <code>ldr r0,[r5,#4]</code>
	-	(0x0040): <code>ldr r3,[r5,#12]</code>
	-	(0x0042): <code>orr r0,r0,r3</code>
	-	(0x0046): <code>str r0,[r5,#4]</code>
	(0x0050): <code>ldr r0,[r6,#100]</code>	(0x0048): <code>ldr r0,[r6,#100]</code>
	(0x0052): <code>blx r0</code>	(0x004a): <code>blx r0</code>
	(0x0054): <code>data 0xe7d6(59350)</code>	(0x004e): <code>data 0xe7d6(59350)</code>
	(0x0056): <code>data 0x4a2c(18988)</code>	(0x0050): <code>data 0x4a2c(18988)</code>

实现循环不变污点传播代码外提,关键在于找到循环污点不变量,分以下两种情况讨论.

- 第 1 是循环归纳变量,循环归纳变量较容易分析,其污染点和传播点都在 PHI 节点处,且其数值加减某一常数.如示例代码 `add-int/lit8 v2,v2,0x1`,当中 v_2 是归纳变量,其污点值在循环过程中不变.
- 第 2 是非归纳不变量,非归纳不变量分析较为复杂.第 1 种情况是某变量污染点和传播点都在一处且传播点位于该处的其他变量在循环体内不存在污染点,如示例代码 `add-int/2addr v0,v1` 满足此类情况, v_0 寄存器的污染点和传播点都在该指令处,且 v_1 寄存器在循环其他处不存在污染点;第 2 种情况是某变量污染点和传播点都在一处且传播点位于该处的其他变量是循环污点不变量,如示例代码 `add-int/2addr v0,v1` 与 `add-int/2addr v1,v0` 中, v_0 和 v_1 的污点值均是循环不变量.第 2 种情况需要使用迭代算法,由于污点传播框架中污点值域格具有有穷的高度,且是单调的,可证明迭代算法是收敛的.其证明过程可参见编译原理中数据流分析算法敛散性证明^[29].

因而,循环不变污点传播代码外提算法如算法 3 所示.

算法 3. 循环不变污点传播代码外提算法.

输入:编译单元**cUnit*,循环体污点传播首指令**headLIR*,循环体污点传播尾指令**tailLIR*.

输出:编译单元**cUnit*.

```

1. for (currentLIR=t_headLIR;currentLIR!=t_tailLIR;currentLIR=TAINT_NEXT_LIR(currentLIR))
2.   { if (!(EncodingMap[currentLIR→opcode].flags & (IS_ORR)) continue;
3.     //判断当前污点传播指令类型是否 ORR 类型
4.     findLIV=1;
5.     for (moveLIR=TAINT_NEXT_LIR(currentLIR);moveLIR!=t_tailLIR;
           moveLIR=TAINT_NEXT_LIR(moveLIR))
6.       { if (moveLIR→operands[0]==currentLIR→operands[0] &&
            !((moveLIR→operands[2]==const||isLIV(moveLIR→operands[2])))
7.         findLIV=0;
8.         break;}
9.       if (findLIV=1)
10.        {newLIR=dvmCompilerCopyLIR(cUnit,moveLIR);
11.         dvmCompilerInsertLIRAfter(chainingcellLIR,newLIR);
12.         moveLIR→flags.isNop=true;}
13.        //将循环不变污点传播代码外提
14.        } //end for
15.    } //end for

```

4 系统实现框架与测试

4.1 系统实现框架

在 Dalvik 即时编译器基础上实现了污点传播编译优化技术,其整体框架如图 3 所示.

- (1) 在解释器进行热点路径探测,若该路径入口指令执行次数达到阈值 *threshold*(40),则编译该路径并进行污点传播优化;否则,不进行污点传播优化(冷路径优化代价过大).
- (2) 创建编译线程,完成 MIR 到 SSA 转换和基本的方法内联优化,并填充 MIR 相应污点数据结构.
- (3) 将 MIR 转换成 LIR.首先,依据程序逻辑向污点传播逻辑转化特点,完成 MIR 指令污染点和传播点分析,并将插入的污点跟踪指令组织为双向链表;填充 LIR 相应污点数据结构(*Taint_map*),并完成路径流程图分析.若该路径是循环,则需完成循环信息分析.
- (4) 通过以上 3 种污点传播优化算法,完成对以双向链表组织的污点跟踪指令的 LIR 进行污点传播优化.

最后,将 LIR 转化成机器码,并返回路径将入口内存地址.

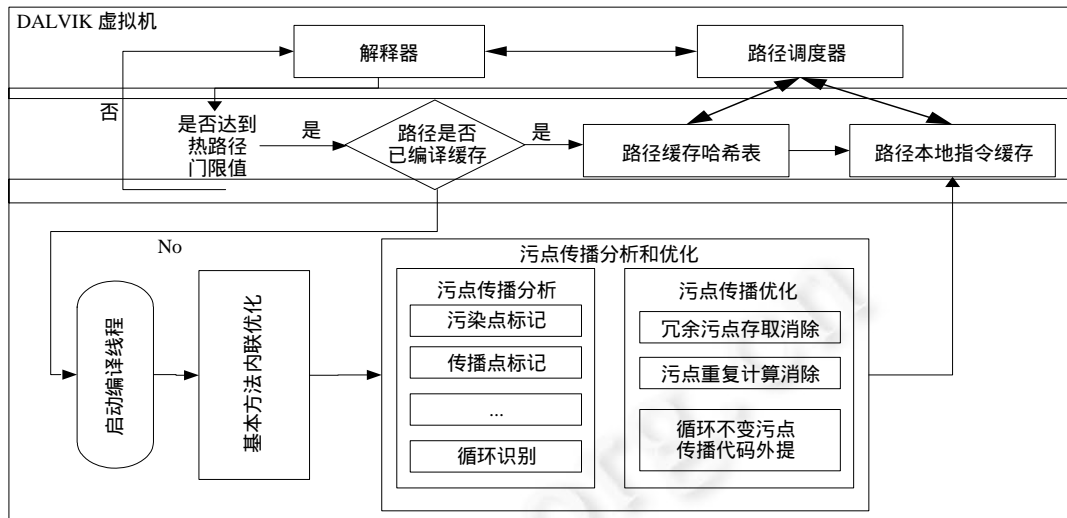


Fig.3 Framework of taint propagation optimization

图 3 污点传播优化整体框架

4.2 系统功能与性能测试

本文所有测试均在模拟器上进行,操作系统环境为 os/Ubuntu-12.04,cpu/2.27GHZ,RAM/4GB,version/Android-4.1.1.启动模拟器关键参数如下:emulator -kernel qumu-armv7 -ramdisk ramdisk.img -data userdata.img -sdcard sdcard.img -partition-size 500 -memory 512(本文原型系统 OFCDroid 和对比系统 FCDroid 可从 <http://pan.baidu.com/s/1pJC63uN> 处下载).

4.2.1 系统功能测试

功能测试的目的在于测试污点优化正确性,即在应用污点传播优化算法后,与未优化前系统隐私泄露报告是否一致.测试用例使用系统应用和自制应用 wzztest.apk,测试用例包括了不同类型 Dalvik 指令 `OP_MOVE`, `OP_RETURN`, `OP_CONST`, `OP_AGET`, `OP_APUT`, `OP_IGET`, `OP_IPUT`, `OP_SGET`, `OP_SPUT`, `OP_INVOKE`, `OP_INT_TO_LONG`, `OP_ADD_INT` 和 `OP_OR_INT`.测试的隐私数据类型包括 `string`, `char`, `byte`, `int`, `long`, `float`, `double` 和 `array`.以自带短信程序在通过系统调用接口读取联系人信息时,短信程序与安卓系统间通信方法 `writeString()` 所含的路径将被编译为热点路径,其测试结果如图 4 所示.其中,路径中包含的 Dalvik 指令如下.

```
const-string v1, "*"
new-instance v3, Ljava/lang/StringBuilder
invoke-virtual {v3,v1}, Ljava/lang/StringBuilder;
→append(Ljava/lang/String;) Ljava/lang/StringBuilder;
move-result-object v4
```

其次,使用污点跟踪测试床 DroidBench^[30]测试系统污点跟踪的正确性.DroidBench 是专门针对安卓平台开发的污点跟踪测试工具,可测试系统在数组污点跟踪、方法污点跟踪、指令污点跟踪、应用内部和外部通信污点跟踪和隐式污点跟踪等方面污点跟踪的正确性,测试结果见表 3.在所选取的 51 个小测试中,OFCDroid 检测到其中 36 个隐私泄露并发出警告,15 个未发出警告,7 个误报.未发出警告是由于未跟踪隐式信息流,误报是由于数组和字符串污点跟踪过于粗糙.OFCDroid 与 FCDroid 各项测试结果均一致,表明经过污点传播优化后,污点跟踪仍然是正确的.



Fig.4 Notification of privacy leak
图 4 隐私泄露通知

Table 3 Test results of DroidBench
表 3 DroidBench 测试结果

	正确警告数量 a	错误警告数量 b	未警告数量 c	准确率 $a/(a+b)$ (%)	漏报率 $c/(a+c)$ (%)
FCDroid	36	7	15	83.7	29.4
OFCDroid	36	7	15	83.7	29.4

4.2.2 系统性能测试

首先测试污点传播优化算法对单条热路径的优化效果和优化代价,优化效果主要通过编译后代码的大小来衡量,代码越小,其执行速度越快,内存占用越少,故优化效果更佳.优化代价主要通过编译每条热路径耗费的编译时间长短来衡量,编译时间越短,优化代价就越低.选择安卓操作系统最常被编译为热点路径所属的方法作为测试对象,其测试结果见表 4.

Table 4 Effects and costs of the hot trace optimization
表 4 热路径优化效果与代价

热路径所在方法名	Android		FCDroid		OFCDroid	
	路径大小 (byte)	编译时间 (μ s)	路径大小 (byte)	编译时间 (μ s)	路径大小 (byte)	编译时间 (μ s)
Ljava/io/File;fixSlashes	116	26 458	140	27 973	128	31 196
Ljava/util/ArrayList\$ArrayListIterator;next	76	17 325	76	18 443	76	20 912
Llibcore/util/collectionUtils\$1\$1;computenext	96	20 091	112	22 054	104	25 503
Llibcore/util/collectionUtils\$1\$1;hasnext	68	15 629	80	18 699	72	20 875
Ljava/lang/caseMapper;toUpperCase	188	31 740	244	33 872	228	35 280
Ljava/lang/character;isHighSurrogate	64	16 854	72	17 947	72	18 465
Ljava/lang/String;hashCode	76	20 448	112	22 263	96	26 283
LAndroid/os/systemproperties;getint	136	36 215	180	39 201	164	43 314

- 对于方法 Ljava/io/File;fixSlashes,Android 系统通过即时编译器编译后代码占用 116 字节内存空间,编译耗时 26 458 μ s;加入污点传播机制后,FCDroid 系统通过即时编译器编译后代码占用 140 字节内存空间,编译耗时 27 973 μ s;引入污点传播优化机制后,OFCDroid 系统通过即时编译器编译后代码占用 128 字节内存空间,编译耗时 31 196 μ s.编译器以 11.5%的编译时间代价获取了 8.5%的运行速度和内存占用的优化效果.
- 对于方法 Ljava/lang/String;hashCode,优化效果最佳达到 14.2%,但优化代价也最大.
- 对于方法 Ljava/lang/character;isHighSurrogate,其优化代价最小为 2.8%,但优化效果最差.

Android 系统平均每条热路径占用 102 字节,平均编译时间为 23 095 μ s;FCDroid 系统平均每条热路径占用 127 字节,平均编译时间为 25 056 μ s;OFCDroid 系统平均每条热路径占用 117 字节,平均编译时间为 27 728 μ s.实验结果表明,经过优化后,每条路径平均少占用 10 字节,相对于整条热路径指令,污点跟踪优化算法的平均优化率为 7.8%.相对于插入的污点跟踪指令,污点跟踪优化算法平均优化率为 38%.

然后,测试各污点传播优化算法对于单条热路径的优化性能.优化性能主要通过各污点传播优化算法执行时间的长短来衡量,执行时间越长,算法相对性能就越差.同样选择上述表 4 热点路径作为测试对象,其测试结果见表 5.

Table 5 Performance of taint propagation optimization algorithms

表 5 污点传播优化算法性能

热路径所在方法名	冗余污点存取 优化时间(μ s)	污点重复计算 优化时间(μ s)	循环不变污点代码 外提优化时间(μ s)	总优化时间 (μ s)
Ljava/io/File;fixSlashes	827	222	743	1 859
Ljava/util/ArryList\$ArraylistIterator;next	763	145	0	933
Llibcore/util/collectionUtils\$1\$1;computenext	821	125	0	971
Llibcore/util/collectionUtils\$1\$1;hasnext	132	21	0	197
Ljava/lang/caseMapper;toUpperCase	758	215	0	994
Ljava/lang/character;isHighSurrogate	139	186	0	344
Ljava/lang/String;hashCode	635	137	1 335	2 126
LAndroid/os/systemproperties;getint	1 169	31	0	1 217

- 对于优化算法冗余污点存取消除,平均优化时间 1 180 μ s,方法 LAndroid/os/systemproperties;getint 耗时最长为 2 169 μ s.
- 对于优化算法污点重复计算消除,平均优化时间 135 μ s,方法 Ljava/io/File;fixSlashes 耗时最长为 222 μ s.
- 对于优化算法循环不变污点传播代码外提,方法 Ljava/io/File;fixSlashes 和 Ljava/lang/String;hashCode 中路径包含循环,平均优化时间 1 936 μ s.由于其他路径不属于循环,故其用时为 0us.

其次,选择常见系统应用并测试其使用性能.选择见表 6 第 1 列所示的安卓应用程序作为测试对象,每个应用分别进行 20 次使用测试,并记录其平均执行时间和方差,经过优化后的执行时间越短,表明优化效果越好.系统优化效率的计算方法为

$$(Time_{FCDroid} - Time_{OFCDroid}) / Time_{FCDroid} \times 100\%$$

Table 6 Optimization effect of various application under prototype system

表 6 不同应用在原型系统下的优化效果

应用程序名	Android		FCDroid		OFCDroid		优化效果(%)
	平均耗时(ms)	方差	平均耗时(ms)	方差	平均耗时(ms)	方差	
短信	72	0.5	86	0.5	78	1.2	9.3
美图秀秀	745	34.2	944	37.5	880	35.5	6.7
联系人	89	1.3	111	1.7	101	3.3	9.0
电话	122	2.2	138	3.2	129	4.6	6.5
图片浏览	344	10.5	381	12.7	357	14.3	6.2
微信	255	15.4	298	17.3	275	17.2	7.7
uc 浏览器	1 723	66.0	2 001	68.3	1 874	77.3	6.3
ZArchiver 压缩	1 534	57.4	1 756	46.5	1 700	53.4	3.1

- 对于发送短信,测试点击信息发送直至发送成功所需时间,Android 系统耗时 72ms;加入污点传播机制后,FCDroid 系统耗时 86ms;经过优化后,OFCDroid 系统耗时 78ms.污点跟踪优化后,系统效率提高了 9.3%.
- 对于拨打电话,测试按下拨打键至系统硬件产生回应所需时间,其优化效率为 6.5%.
- 对于拍摄照片,测试按下拍摄键至保存照片完成(照片大小约在 546KB 左右)所需时间,其优化效率为 6.8%.
- 对于微信聊天,测试按下发送键至信息发送完成所需时间,其优化效率为 7.7%.

- 对于浏览网页,测试按下前进键至网页完成显示所需时间,其优化效率为 6.3%.
 - 对于读取联系人信息和浏览照片,其优化效率分别为 9.0%和 6.2%.
 - 对于压缩文件,选择 1M 大小的单个 txt 文件,测试开始压缩至压缩完成所需时间,其优化效率为 3.1%.
- 相对来说优化效率偏低,可能是由于压缩算法主要实现在 .so 文件中,而本文工作是基于即时编译的污点跟踪优化,是在将 dalvik 字节码编译转化为本地指令码过程中进行的污点传播优化.

整体上,通过优化后,系统效率平均提高了 6.8%.

最后,采用 caffeineMark 3.0 测试 OFCDroid 系统整体性能,其测试结果如图 5 所示.对于 caffeineMark 测试结果,在 float 得分与 logic 得分上,OFCDroid 与 FCDroid 差别并不明显,由于其得分高低主要与硬件性能相关;而在 loop 得分,OFCDroid 比 FCDroid 多出 26%.这是由于循环代码执行次数多,容易被探测为热点路径且被编译为本地代码.以方法 Ljava/lang/String;hashCode 为例,设其循环执行次数为 n ,每条指令执行时间为 $m\mu\text{s}$,其优化后节省指令数为 8,则对于该方法执行一次可节省 $8nm\mu\text{s}$.循环次数越多,优化效果就越明显.

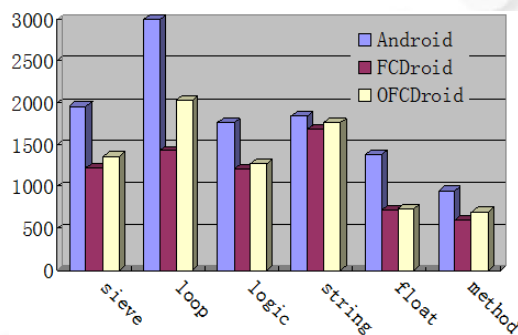


Fig.5 Test results of caffeineMark

图 5 CaffeineMark 测试结果

5 结束语

本文针对污点跟踪技术在移动隐私保护方面存在效率较低的问题,提出了一种基于即时编译的动态污点传播优化方法.首先,将程序逻辑精确抽象为污点传播逻辑,简化污点传播分析复杂性,然后,提出了一个污点传播框架,证明该框架下污点传播分析算法的正确性和有效性,给出了污点传播代码优化算法的实现及算法的性能测试结果.实验结果表明,该优化方法有效提高了动态污点跟踪系统的性能.由于污点传播优化精确性问题依赖于全局的污点传播分析和控制流分析,还有待后续进一步的深入研究.

References:

- [1] Wan ZY, Jiang X. Dissecting Android malware: Characterization and evolution. In: Proc. of the IEEE Symp. on Security and Privacy. Oakland: IEEE, 2012. 95–109. [doi: 10.1109/SP.2012.16]
- [2] Schwartz EJ, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proc. of the IEEE Symp. on Security and Privacy. Oakland: IEEE, 2010. 317–331. [doi: 10.1109/SP.2010.26]
- [3] Sun H, Li HP, Zeng QK. Statically detect and Run-time check integer-based vulnerabilities with information flow. Ruan Jian Xue Bao/Journal of Software, 2013,24(12):2767–2781 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/4385.htm> [doi: 10.3724/SP.J.1001.2013.04385]
- [4] Chow J, Pfaff B, Garfinkel T, Christopher K, Rosenblum M. Understanding data lifetime via whole system simulation. In: Proc. of the USENIX Security Symp. Berkeley: USENIX, 2004. 321–336.
- [5] Attariyan M, Flinn J. Automating configuration troubleshooting with dynamic information flow analysis. In: Proc. of the 9th OSDI. Berkeley: USENIX, 2010. 237–250.

- [6] Nair SK, Simpson PND, Crispo B, Tanenbaum AS. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 2008,197(1):3–16. [doi: 10.1016/j.entcs.2007.10.010]
- [7] Lam LC, Chiueh T. A general dynamic information flow tracking framework for security applications. In: *Proc. of the 22nd Annual Computer Security Applications Conf. (ACSAC 2006)*. IEEE, 2006. 463–472. [doi: 10.1109/ACSAC.2006.6]
- [8] Myers AC, Liskov B. Protecting privacy using the decentralized label model. *ACM Trans. on Software Engineering and Methodology*, 2000,9(4):410–442. [doi: 10.1145/363516.363526]
- [9] Hedin D, Sabelfeld A. Information-Flow security for a core of JavaScript. In: *Proc. of the IEEE 25th Computer Security Foundations Symp. (CSF)*. Cambridge: IEEE, 2012. 3–18. [doi: 10.1109/CSF.2012.19]
- [10] Efstathopoulos P, Krohn M, VanDeBogart S, Frey C, Ziegler D. Labels and event processes in the Asbestos operating system. In: *Proc. of the SOSOP*. Brighton: ACM Press, 2005. 17–30. [doi: 10.1145/1095810.1095813]
- [11] Krohn M, Yip A, Brodsky M, Cliffer N, Kaashoek MF, Kolher E. Information flow control for standard OS abstractions. In: *Proc. of the ACM SIGOPS Operating Systems Review*. New York: ACM Press, 2007. 321–334. [doi: 10.1145/1294261.1294293]
- [12] Yang Z, Yin LH, Duan MY, Wu JY, Jin SY, Guo L. Generalized taint propagation model for access control in operation systems. *Ruan Jian Xue Bao/Journal of Software*, 2012,23(6):1602–1619 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/4083.htm> [doi: 10.3724/SP.J.1001.2012.04083]
- [13] Portokalidis G, Homburg P, Anagnostakis K, Bos H. Paranoid Android: Versatile protection for smartphones. In: *Proc. of the 26th Annual Computer Security Applications Conf.* ACM Press, 2010. 347–356. [doi: 10.1145/1920261.1920313]
- [14] Chen S, Kozuch M, Strigkos T, Ryan M, Gibbons PB. Flexible hardware acceleration for instruction-grain program monitoring. *ACM SIGARCH Computer Architecture News*, 2008,36(3):377–388. [doi: 10.1145/1394608.1382153]
- [15] Ruwase O, Gibbons PB, Mowry TC, Ramachandran V, Chen S, Kozuch M. Parallelizing dynamic information flow tracking. In: *Proc. of the 20th Annual Symp. on Parallelism in Algorithms and Architectures*. ACM Press, 2008. 35–45. [doi: 10.1145/1378533.1378538]
- [16] Chow J, Garfinkel T, Chen PM. Decoupling dynamic program analysis from execution in virtual environments. In: *Proc. of the USENIX 2008 Annual Technical Conf.* Berkeley: USENIX, 2008. 1–14.
- [17] Jee K, Kemerlis VP, Keromytis AD, Portokalidis G. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In: *Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security*. ACM Press, 2013. 235–246. [doi: 10.1145/2508859.2516704]
- [18] Jee K, Portokalidis G, Kemerlis VP, Ghosh S, August DI. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In: *Proc. of the 19th NDSS*. San Diego: Internet Society, 2012. 324–335.
- [19] Chang W, Streiff B, Lin C. Efficient and extensible security enforcement using dynamic data flow analysis. In: *Proc. of the 15th ACM Conf. on Computer and Communications Security*. Alexandria: ACM Press, 2008. 39–50. [doi: 10.1145/1455770.1455778]
- [20] Ho A, Fetterman M, Clark C, Warfield A, Hand S. Practical taint-based protection using demand emulation. *ACM SIGOPS Operating Systems Review*, 2006,40(4):29–41. [doi: 10.1145/1218063.1217939]
- [21] Portokalidis G, Bos H. Eudaemon: Involuntary and on-demand emulation against zero-day exploits. In: *Proc. of the 2008 EuroSys*. ACM Press, 2008. 287–299. [doi: 10.1145/1352592.1352622]
- [22] Saxena P, Sekar R, Puranik V. Efficient fine-grained binary instrumentation with applications to taint-tracking. In: *Proc. of the 6th CGO*. ACM Press, 2008. 74–83.
- [23] Kim HC, Keromytis AD. On the deployment of dynamic taint analysis for application communities. *IEICE Trans. on Information & Systems*, 2009,92(3):548–551.
- [24] Qin F, Wang C, Li Z, Kim H, Zhou Y. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In: *Proc. of the 39th Annual IEEE/ACM Int'l Symp. on Microarchitecture*. IEEE, 2006. 135–148. [doi: 10.1109/MICRO.2006.29]
- [25] Kemerlis VP, Portokalidis G, Jee K, Keromytis AD. libdft: Practical dynamic data flow tracking for commodity systems. *ACM SIGPLAN Notices*, 2012,47(7):121–132. [doi: 10.1145/2365864.2151042]
- [26] Enck W, Gilbert P, Han S, Tendulkar, Chun BG. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: *Proc. of the OSDI*. Berkeley: USENIX, 2010. 255–270. [doi: 10.1145/2494522]

- [27] Huang Y, Chen Y, Yang W, Shann JJ. File-Based sharing for dynamically compiled code on Dalvik virtual machine. In: Proc. of the Int'l Computer Symp. IEEE, 2010. 489–494. [doi: 10.1109/COMPSYM.2010.5685462]
- [28] Ling M, Wu JP, Feng KH. An adaptive compilation system based on the dalvik virtual machine. Acta Electronica Sinica, 2013, 41(8):1622–1627 (in Chinese with English abstract). [doi: 10.3969/j.issn.0372-2112.2013.08.027]
- [29] Aho AV, Sethi R, Ullman JD. Compilers, Principles, Techniques. 2nd ed., Addison Wesley Publishing Company, 1986. 688–703.
- [30] Fritz C, Arzt S, Rasthofer S, Bodden E, Bartel A. Highly precise taint analysis for android application. Technical Report, TUD-CS-2013-0113, 2013. <http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf>

附中文参考文献:

- [3] 孙浩,李会朋,曾庆凯.基于信息流的整数漏洞插装和验证.软件学报,2013,24(12):2767–2781. <http://www.jos.org.cn/1000-9825/4385.htm> [doi: 10.3724/SP.J.1001.2013.04385]
- [12] 杨智,殷丽华,段冰毅,吴金宇,金舒原,郭莉.基于广义污点传播模型的操作系统的访问控制.软件学报,2012,23(6):1602–1619. <http://www.jos.org.cn/1000-9825/4083.htm> [doi: 10.3724/SP.J.1001.2012.04083]
- [28] 凌明,武建平,冯克环.一种 Dalvik 虚拟机的自适应编译系统.电子学报,2013,41(8):1622–1627. [doi: 10.3969/j.issn.0372-2112.2013.08.027]



吴泽智(1990 -),男,湖南长沙人,博士生,主要研究领域为网络与信息安全,信息流控制.



杨智(1975 -),男,博士,副教授,主要研究领域为操作系统安全,信息流控制,云计算安全.



陈性元(1963 -),男,博士,教授,博士生导师,主要研究领域为网络与信息安全.



杜学绘(1968 -),女,博士,教授,博士生导师,主要研究领域为信息系统多级安全,云计算安全.