基于后缀树的二进制可执行代码的克隆检测算法

张凌浩[1] 桂盛霖[2,3] 穆逢君[2] 王 胜[1]
(国网四川省电力公司电力科学研究院 成都 610000)[1]
(电子科技大学计算机科学与工程学院 成都 611731)[2]
(中国电子科技集团公司第三十研究所 成都 610041)[3]

摘 要 如何发现代码克隆,是软件维护和软件侵权纠纷案件中的一个关键问题。由于商业保密等原因,在商业软件的侵权纠纷案中往往无法使用基于源代码比对的克隆检测技术。因此,针对这类无法获得源代码进行代码克隆检测的场景,文中提出一种针对二进制可执行文件分析的代码克隆检测方法。首先,通过反编译与指令类型抽象得到二进制可执行目标文件的指令类型序列;然后,对指令类型序列构建后缀树,利用后缀树的性质获取函数级的指令序列间的克隆信息,并通过消除沙砾指令进一步提高检测性能;最后,基于 MIPS32 指令集,使用 Linux 内核和经过混淆处理的代码分别作为克隆级别 0—级别 2 与级别 1—级别 4 的二进制可执行文件代码克隆测试样本,并与源代码检测工具进行对比测试。结果表明,所提算法在缺少源代码的场景下同样能进行细粒度的克隆分析,且对各级代码克隆均具有较好的检测性能。

关键词 代码克隆,二进制可执行文件,后缀树,性能优化
中图法分类号 TP311.5 **文献标识码** A **DOI** 10.11896/jsjkx.180801573

Clone Detection Algorithm for Binary Executable Code with Suffix Tree

ZHANG Ling-hao[1] GUI Sheng-lin[2,3] MU Feng-jun[2] WANG Sheng[1]
(State Grid Sichuan Electric Power Research Institute, Chengdu 610000, China)[1]
(School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China)[2]
(The 30th Institute of China Electronics Technology Group Corporation, Chengdu 610041, China)[3]

Abstract How to detect code clones is an important issue in software maintenance and software infringements. Clone detection techniques based on source code tend to fail in the infringement disputes of commercial software due to trade secret. Therefore, in the scenario when the source code is unavailable for detection, this paper presented a clone detection algorithm based on binary executable file analysis. Firstly, instruction type sequences of binary executable files are obtained by decompilation instruction type abstraction, then a suffix tree is constructed based on these instruction type sequences. The clone pairs among functions can be figured out based on this suffix tree. In addition, this paper eliminated gravel instructions for enhancing performance. At last, based on MIPS32 instruction set, this paper used respectively Linux kernel and obfuscated test code as samples on clone level 0—level 2 and level 1—level 4 to compare with the source code detection tools. Test results show that even in the scenario where the source code is lacking, this algorithm can also perform fine-grained clone analysis and has high detection performance for code clones at all levels.

Keywords Code cone, Binary executable file, Suffix tree, Performance optimization

1 引言

代码克隆检测(Code Clone Detection),也称为代码同源性检测,用于分析待比对代码文件之间的相似之处,在软件知识产权侵权纠纷案件、软件的来源分析等多个场景中具有重要用途。近年来,随着软件行业的高速发展,关于软件侵权的维权案件时有发生。“Jacobsen V. Katzer 案”与“久合成对康能普视侵权案”是软件侵权类的典型案例[1],在它们的判决过程中,均对源码是否存在克隆行为进行了检测[2-3],且检测结果对案件的判决产生了决定性影响。此外,代码克隆检测的结果在软件维护、软件复用、软件演化分析、代码压缩等多个方面均具有重要的应用价值。因此,代码克隆问题的检测准确性是各类代码克隆检测技术研究的关键。

市场上所销售的商业软件出于商业保密要求,其源代码

到稿日期:2018-08-25 返修日期:2019-01-08 本文受国家自然科学基金(61401067),国网四川省电力公司科技项目(521997170001P,521997170017)资助。

张凌浩(1985—),男,博士,工程师,主要研究方向为电力信息安全技术;桂盛霖(1983—),男,博士,副教授,CCF 会员,主要研究方向为嵌入式软件技术、信息安全,E-mail:shenglin_gui@uestc.edu.cn;穆逢君(1997—),男,主要研究方向为软件克隆检测技术;王 胜(1987—),男,硕士,工程师,主要研究方向为网络安全技术。

往往无法获取,因此在涉及到商业软件的侵权纠纷案中通常无法使用基于源代码比对的克隆检测技术;并且随着代码混淆技术[4]的广泛应用,代码克隆方可能使用代码混淆技术干扰或绕开基于源代码的克隆检测技术,降低了分析结果的准确性。

因此,针对无法或难以获得源代码进行代码克隆检测的现实场景,本文提出了一种针对二进制可执行文件的代码克隆检测方法。为了验证所提算法的性能,将其与多个代码克隆检测工具进行了克隆级别 1—级别 4 的检测性能对比。结果表明,在缺少源代码的场景下,本文算法能进行细粒度的克隆分析,并且对各级代码克隆均具有较好的检测性能。

2 相关工作

根据被分析的代码对象,代码克隆检测技术可分为两类:基于源代码的检测技术[5-7]与基于二进制文件的检测技术[8]。

2.1 基于源代码的克隆检测

基于源代码的检测技术主要分为 4 类方法:基于文本的检测技术(Text-based Techniques)、基于标记的检测技术(Token-based Techniques)、基于度量的检测技术(Metrics-based Techniques,MT)和基于图的检测技术(Graph-based Techniques,GT)。

基于文本的检测技术在对源代码进行删除、缩进等简单的预处理后,直接对比源代码文件之间的相似性。文献[9]使用窗口滑块技术来计算窗口内的源代码字符串的散列值,并将其作为代码克隆段的检测方式。该技术对源代码的预处理比较简单,因此通常只支持克隆级别 0(整个源代码文件完全相同)和级别 1(除空白、注释及代码布局的改变外,其他部分完全相同)的代码检测。

基于标记的检测技术也称基于词法的检测技术,它基于词法分析器对源代码进行处理,根据预设的词法规则将源代码抽象为 Token 序列,通过比对 Token 序列之间的相似序列片段检测源码间的克隆行为。CCFinder[10]支持对 C,C++,Java 等语言的源码生成 Token 序列进行检测,并能可视化地展示相似序列片段;SourcererCC[11]使用索引查询的方法减少了代码克隆检测中的资源消耗,可在标准工作站上实现对大型项目代码库的克隆检测。基于标记的检测技术对源代码文件的预处理较为复杂,虽然性能略逊于基于文本的检测技术,但具有更高的检测准确率,可支持克隆级别 0、级别 1 和级别 2(在级别 1 的基础上,仅允许发生变量名与常量值的改变)的代码检测。

基于度量的检测技术与基于图的检测技术的共同特点是都需要先将源代码表示为抽象语法树(Abstract Syntax Tree,AST)、控制流图(Control Flow Graph,CFG)或程序依赖图(Program Dependency Graph,PDG)等图结构,然后再进行后续分析。其中,基于度量的检测技术对图结构中的函数出入度、基本块数量等度量值进行对比,并将其作为代码克隆检测的特征。由于该类技术须对代码文件做更细粒度的分析和特征提取,其性能显著低于前两类技术。文献[12]使用程序中的控制语句、运算符和操作数信息计算度量值,并通过比较度量值检测两个程序间的相似性。而基于图的检测技术是一种语义级的检测方法,通过比较 PDG 的图结构进行代码克隆检测,从而支持克隆级别 0—级别 3(在级别 2 的基础上,级别 3 允许进行语句的增加、修改或删除)的代码检测。CCSharp[13]通过简化 PDG 图的方式进一步解决了基于图的检测工具消耗时间较多的问题。

此外,Nicad[14]先使用 TXL 框架对源代码进行标准化等预处理,再基于文本与分析树(Parse Tree)对函数间的相似性进行分析。文献[15]提出了一个耦合编译器前端与深度神经网络相结合的框架,在词法与语法层面上对源代码进行模式挖掘和匹配。Vincent[16]则对代码的截图进行高斯模糊处理,然后根据图片间的差异与重合部分来判断代码是否具有相似的特征。

文献[5]按照克隆级别 0—级别 4 对 42 款源代码克隆检测工具进行了详细测试,测试结果显示其中 41 款工具支持级别 0—级别 2 的代码克隆检测,仅有 8 款工具可检测出级别 3 的代码克隆(但其假阳性结果接近半数),并且无一款工具支持级别 4(语义级克隆,即使用不同的语法实现了相同的语义)的测试。

2.2 基于二进制文件的检测技术

对二进制文件的代码克隆研究工作远少于基于源码的检测技术,现有的技术主要可分为基于汇编指令的检测技术与基于结构的检测技术。

基于汇编指令的检测技术将汇编指令序列作为分析对象,通过检索相似指令序列的方式进行二进制文件的克隆检测。该方法效率高,但编译器引入的指令变化会对检测结果产生较大的影响,降低了该技术的准确率。文献[17]使用窗口滑块算法对操作码序列的相似部分进行检测。

基于结构的检测技术与源代码级的 MT 和 GT 技术的原理相似,从二进制文件中提取出函数基本块个数、控制流图边数等结构特征并进行比对,将具有相同结构特征的函数视为相似函数。文献[18]对二进制可执行文件中的跳转指令进行分析,通过比较程序中每个执行分支的方式实现了对函数间相似度的计算。文献[19]通过对二进制文件中的参数和跳转地址进行分析,实现了对 IA-32,ARM 和 MIPS 平台上运行的二进制文件间的语义级克隆检测。由于处理过程较复杂,因此其检测性能偏低。

综上所述,相比于二进制文件的克隆检测,已有的研究工作更多地关注源代码级的克隆检测问题。相比于现有的二进制代码克隆检测技术,本文针对不可获得源代码进行克隆检测的实际应用场景进行研究,具有以下创新点。

(1)对代码的预处理方式与现有技术不同。本文从二进制代码序列中提取出代码的操作类型序列作为分析对象,提高了对源代码进行语义等价语句替换的鲁棒性。测试结果表明,本文算法对克隆级别 4 的代码克隆检测性能优于 Nicad。

(2)与文献[17]使用的窗口滑块方法不同,本文利用后缀树对后缀字符序列的表达能力进行序列间相似部分的检索,并给出了相应的定理证明。测试结果表明,对级别 0—级别 2 进行代码克隆检测时本文算法与 Nicad 性能一致。

(3)本文对相邻克隆对及其间的沙砾指令进行合并,以支持对增加、修改或删除语句的克隆代码片段的检测,并降低了编译器的优化策略对结果的干扰。测试结果表明,本文算法对级别 3 的代码克隆检测性能优于 Nicad。

3　基于二进制文件的代码克隆检测

3.1　预处理

二进制可执行文件[20]由 data 段、Text 段等部分组成，其中 Text 段为程序代码段，包含机器指令的操作码和操作数等信息。

图 1 给出了相同功能的不同源代码实现，其语句差异包括赋值位置的改变、while/for 语句的替换、注释标记的增加等。对图 1 给出的两段源代码分别编译后发的现，尽管两者的二进制可执行文件中每条指令的操作数存在较大差异，但指令操作码序列基本一致。

表 1 中的代码 a 和代码 b 分别给出了图 1(a)和图 1(b)编译后得到的 MIPS32 操作码序列，不难看出两者之间仅有一条指令的操作码存在差异(nopw 和 nopl)，其相同率大于 94%。因此，指令的操作码序列可以更好地体现源代码的相似性。但是，由于源代码中同一操作可由不同的同类机器指令完成，因此编译器的优化策略往往会进行同类指令之间的选择。为避免编译器的优化策略对克隆检测过程造成干扰，本文算法对指令序列进行指令类型的抽象，并将得到指令的类型序列作为下一步的分析对象。图 2 给出了本文对二进制可执行文件的预处理流程。

```
while (id[Tk]){
if (condition=(t==id[Hash]){
    tk = id[Tk];
  return;}
  id = id+Idsz;
}
```

(a)混淆前代码

```
for (;id[Tk];) {
  condition=(tk==id[Hash]);
  if (condition) {
tk = id[Tk]; return; }
//某些注释
id+= Idsz;
}
```

(b)混淆后代码

图 1　相同功能的两种代码实现

Fig. 1　Two implementations with the same function

表 1　图 1 中两段代码对应的二进制指令

Table 1　Binary instructions corresponding to two pieces of codes in Fig. 1

代码	操作码序列															
代码 a	mov	mov	test	je	cmp	lea	jne	jmpq	nopw	add	cmp	je	mov	mov	test	jne	mov
代码 b	mov	mov	test	je	cmp	lea	jne	jmpq	nopl	add	cmp	je	mov	mov	test	jne	mov

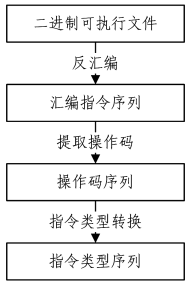

图 2　对二进制可执行文件进行预处理的流程

Fig. 2　Process for preprocessing binary executables

表 2 给出了本文对 MIPS32 处理器指令进行抽象的 9 种指令类型[21]。这些类型覆盖了该指令集的所有指令。

表 2　MIPS32 指令集中的 9 类指令

Table 2　Nine types of instructions in MIPS32 instruction set

指令类型	类型简称	指令操作码
算术加减类	ARI	ADD，SUB 等
算术乘除类	MAD	DIV，MUL，MADD 等
移位运算类	SAR	SLL，SRL，ROTR 等
逻辑运算类	LBF	AND，OR，XOR 等
条件测试和条件传送类	CTCM	MOVN，SLT 等
累加器存取类	ACC	MFHI，MTLO 等
分支跳转类	JAB	BEQ，JAL，JR 等
访存操作类	LAS	LW，SW，LB 等
原子操作类	ARMW	LL，SC

3.2　克隆检测算法

3.2.1　后缀树的构建

首先给出本文所使用的相关概念。

定义 1　字符序列 s 对应的后缀树(Suffix tree)[22]具有如下特征：1)后缀树的每条边都标记为 s 的一个子序列；2)后缀树从根节点到叶节点的每条路径所对应的序列都是 s 的后缀；3)s 的每个后缀在其后缀树上都对应一条从根节点到叶节点的路径。

根据后缀树的定义，不难看出后缀树具有如下性质。

性质 1　对于字符序列 s 的后缀树，若其中存在一非根节点 P 具有两个或以上的子节点，则由根节点至节点 P 的路径对应的序列为从根节点到节点 P 再到叶节点所对应路径集合的公共前缀。

图 3 给出序列 BANANA♯NAN♯对应的后缀树的一部分，其中节点 8 有子节点 11 与子节点 12，则根节点至节点 8 对应的序列 NAN 为根节点至节点 11 与根节点至节点 12 的路径对应序列的公共前缀。

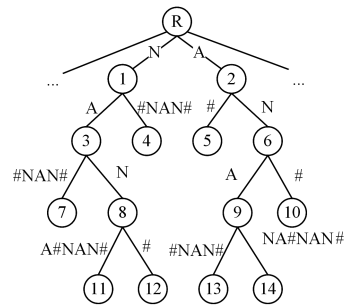

图 3　序列 BANANA♯NAN♯对应的后缀树的部分路径

Fig. 3　Part of suffix tree of "BANANA♯NAN♯"

定义以下对后缀树进行操作的函数。

函数 *getLeaves*(*sTree*)：返回后缀树 *sTree* 中所有叶节点集合。

函数 *getNodes*(*sTree*)：返回后缀树 *sTree* 中的全部节点。

函数 $str(P_i, P_j)$：返回当前后缀树中节点 P_i 至节点 P_j 的路径对应的序列。

函数 *initiatehaveSign*(*sTree*)：将后缀树 *sTree* 中每个节点 P 的 $P.haveSign1$ 值和 $P.haveSign2$ 值均初始化为 False。

除非特别说明，下文中的符号"＋"表示对不同序列进行连接操作。为了表述方便，后缀树 *sTree* 的"*root*"属性表示其根节点。另外，后缀树中每个节点都有"*parent*"属性，用于指向其父节点。同时，后缀树中每个节点均有两个布尔型属性

“*haveSign*1”和“*haveSign*2”，其中 *haveSign*1 记录是否存在由该节点至叶节点的一条路径，其对应的序列中仅含一个分隔符“#”，*haveSign*2 记录是否存在由该节点至叶节点的一条路径，其对应的序列中含有两个分隔符“#”。

定理 1 在序列 s_1+“#”+s_2+“#”(非空序列 s_1 和非空序列 s_2 中均不含“#”)对应的后缀树中，根节点为 R，若存在一节点 P 并且以 P 为根节点的子树中存在两个叶节点 L_i 和 L_j，满足如下条件：

1)序列“#”+s_2+“#”是 $str(P,L_i)$的后缀。

2)$str(P,L_j)$是序列 s_2+“#”的后缀；

则 $str(R,P)$为序列 s_1 与序列 s_2 的公共子序列。

证明：由条件 1)可得出，由于 $str(P,L_i)$已包含两个“#”，所以序列 $str(R,P)$中不含“#”，并且序列 $str(R,P)$是 s_1 的子序列；同时，由于 $str(R,P)$中不含“#”，由条件 2)可推出，$str(R,L_j)$也是序列 s_2+“#”的后缀。因此，$str(R,P)$也是 s_2 的子序列。综上所述，$str(R,P)$为序列 s_1 与 s_2 的公共子序列。证毕。

算法 1 给出了对经过 2.1 节预处理之后的两段序列 IS_1 与 IS_2 进行后缀树构建与处理的过程。

算法 1 后缀树构建算法

```
输入:指令类型序列 IS1,指令类型序列 IS2;
输出:序列 IS1+"#"+ IS2+"#"对应的后缀树 sTree;
1. function buildSuffix()
2. sTree=Ukkonen(IS1+"#"+IS2+"#"); //构建后缀树
3. //对后缀树每个节点的属性值进行修改
4. initiatehaveSign(sTree);
5. updatehaveSign(sTree);
6. return sTree;
7. end function
8.
9. function updatehaveSign (sTree)
10. for node Px∈getLeaves(sTree) do
11. while (Px!=sTree.root)
12. if (str(Px.parent,Px). count("#")==0)
13.    Px. parent. haveSign1=Px. parent. haveSign1|Px. haveSign1;
14.    Px. parent. haveSign2=Px. parent. haveSign2|Px. haveSign2;
15. else if (str(Px. parent,Px). count("#")==1)
16. if (Px. haveSign1=True)
17.    Px. parent. haveSign2=True;
18. else
19.    Px. parent. haveSign1=True;
20. else if (str(Px. parent,Px). count("#")==2)
21.    Px. parent. haveSign2=True;
22.    Px=Px. parent;
23. end function
```

算法 1 中第 2 行调用 *Ukkonen*(IS)[23] 函数对序列 IS 进行后缀树的构建与返回。第 4 行调用 *initiatehaveSign* 函数将每个节点的 *haveSign*1 值和 *haveSign*2 值初始化为 False。第 5 行调用 *updatehaveSign* 函数对每个节点的这两个属性进行更新。第 9－23 行利用函数 *updatehaveSign* ($sTree$)对后缀树 $sTree$ 的每个叶节点 L 至根节点路径上的每个节点进行处理：计算当前节点 P_x 至其父节点 $P_x.parent$ 的边上所具有的“#”数量 $str(P_x.parent,P_x).count$(“#”)，若不含分隔符“#”，则将节点 $P_x.parent$ 与节点 P_x 的 *haveSign*1 值与 *haveSign*2 值分别取或后更新 $P_x.parent$ 的对应值；若含一个分隔符“#”，而节点 P_x 的 *haveSign*1 为 True，则表明存在一条由节点 $P_x.parent$ 经过节点 P_x 至叶节点的路径，其对应的序列中含有两个分隔符“#”，因此需将节点 $P_x.parent$ 的 *haveSign*2 置为 *True*，否则将节点 $P_x.parent$ 的 *haveSign*1 置为 True；若含两个分隔符，同样表明存在一条由节点 $P_x.parent$ 经过节点 P_x 至叶节点的路径，其对应的序列中含有两个分隔符“#”，因此将 $P_x.parent$ 的 *haveSign*2 置为 True。对所有叶节点均进行上述处理后，该后缀树中所有节点的 *haveSign*1 与 *haveSign*2 的值更新完毕。算法 1 的复杂度为 $O(n^2)$，其中 n 为用于构建后缀树的指令类型序列的长度。

3.2.2 克隆对检测过程

定义 2 若在指令类型序列 IS_1 中的一段子序列 CF_1 与 IS_2 的子序列 CF_2 为相似的序列，则称 $CP_i=\{CF_1,CF_2\}$为克隆对(Clone Pair，CP)，CF_1 与 CF_2 也被称为克隆块(Clone Fragment，CF)。克隆对 CP_i 的长度定义为其最长克隆块的长度：$length(CP_i)=\max(length(CF_1),length(CF_2))$。

本文使用 $CF_x.sp$ 和 $CF_x.ep$ 分别表示克隆块 CF_x 的起始位置和结束位置。

定义 3 克隆对 $CP_1=\{CF_1,CF_2\}$与 $CP_2=\{CF_3,CF_4\}$之间如果满足如下关系：CF_1 是 CF_3 的子序列且 CF_2 也是 CF_4 的子序列，则称克隆对 CP_2 包含克隆对 CP_1，也将 CP_1 称为 CP_2 的子克隆对。

定义 4 边界匹配长度(Boundary Match Length，BML)为代码克隆检测结果中的最小匹配序列长度。

算法 2 给出了从算法 1 得到的后缀树 $sTree$ 中提取长度不小于 BML 的克隆对的过程。其中，第 3 行对后缀树 $sTree$ 中的每个节点 P_x 判断其 *haveSign*1 与 *haveSign*2 值是否均为 1(即满足定理 1)，若是，则计算出 $str(sTree.root,P_x)$，该值为 IS_1 和 IS_2 的公共子序列。第 4 行的函数 *makeCP*(s)根据输入序列 s 计算出 s 所对应的克隆对 CP_i，若该克隆对 CP_i 的长度不小于给定的 BML 值，则将该克隆对 CP_i 的信息加入克隆对集合 $cpSet$。最后，通过函数 *mergeCP* ($cpSet$)计算克隆对集合 $cpSet$ 中各克隆对之间的包含关系，并删除所有的子克隆对。

算法 2 克隆检测算法

```
输入:后缀树 sTree,整数 BML;
输出:克隆对集合 result;
1. function cloneDetecting()
2. for node Px∈getNodes(sTree) do
3. if(Px. haveSign1==1&& Px. haveSign2==1)
4.    CPi=makeCP(str(sTree. root,Px))
5.       if (length(CPi)≥BML)
6.          cpSet. append(CPi);
7.          mergeCP(cpSet);
8. end function
```

算法 2 使用后缀树性质与定理 1，实现了两个操作类型序列间的克隆对检测。对于待检测的两个操作类型序列 IS_1 和 IS_2，算法 1 和算法 2 的总时间复杂度为 $O(n^2)$，其中 n 为序列 IS_1+“#”+IS_2+“#”的长度。

3.3 沙砾指令的优化

源代码中的语句增加、修改或删除，都会导致编译后的指

令序列发生变化；另外，编译器也可能对相邻的基本块调整位置顺序，以提升软件性能。以上这些情况都可能导致位置相邻克隆对的出现。因此，为了增强本文算法对克隆级别 3 的检测性能，需要对满足条件的相邻克隆对进行合并。

定义 5　同一指令类型序列中的两个克隆块 CF_x 与 CF_y 间的最近距离 $disMin(CF_x,CF_y)$ 为 CF_x 和 CF_y 间最相近两个元素间的距离，其最远距离 $disMax(CF_x,CF_y)$ 为 CF_x 和 CF_y 间相距最远的两个元素的距离（假定 CF_x 比 CF_y 更接近序列的起始处），则有：

$$disMin(CF_x,CF_y)=CF_y.sp-CF_x.ep$$

$$disMax(CF_x,CF_y)=CF_y.ep-CF_x.sp$$

定义 6　给定参数 D 和 R，若两个相邻克隆对 $CP_x=\{CF_1,CF_2\}$ 与 $CP_y=\{CF_3,CF_4\}$ 满足以下条件：

1）CP_x 的克隆块 CF_1 与 CP_y 的克隆块 CF_3 位于同一指令类型序列，CP_x 的克隆块 CF_2 与 CP_y 的克隆块 CF_4 位于另一个指令类型序列；

2）$\max(disMin(CF_1,CF_3),disMin(CF_2,CF_4))\leqslant D$。

则称 CF_1 与 CF_3 间的指令和 CF_2 与 CF_4 间的指令均为沙砾指令。

定义 6 中的参数 D 和 R 用于控制沙砾指令的粒度大小，其中参数 D 用于限制两克隆块间最近距离的上限值，参数 R 用于限制沙砾指令占合并后克隆对的长度比例。为了消除沙砾指令对克隆检测能力的影响，可以将满足定义 6 中条件的 CP_x 和 CP_y 合并为一个更大的克隆对 CP_{x+y}。因此需对算法 2 进行如下扩展：

1）在原算法 2 输入参数的基础上增加两个参数：D 和 R。

2）将原算法 2 中第 7 行函数 $mergeCP(cpSet)$ 改为 $mergeCP(cpSet,D,R)$，该函数在原功能基础上增加“满足定义 6 中条件时合并相邻克隆对”的功能。

图 4 给出了两个相邻克隆对的例子，其中给定参数 D 的值为 10；R 的值为 0.25。因此，IS_1 的第 762－771 条指令及 IS_2 的第 364－371 条指令为沙砾指令。根据扩展后的算法 2，CP_1 与 CP_2 被合并为一个克隆对 $CP'=\{CF_1',CF_2'\}$，其中 $CF_1'.sp=135$，$CF_1'.ep=965$，$CF_2'.sp=170$，$CF_2'.ep=998$。

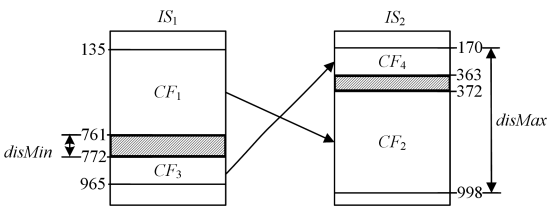

图 4　相邻克隆块的合并

Fig. 4　Combination of two near CPs

4　测试

为了验证本文所提算法的性能，将其与其他代码克隆对检测技术进行对比测试。本文用于实验的计算设备的硬件配置为：内存 8GB，CPU 为 i5-3320M@2.6GHz。本节所使用的二进制可执行文件均基于 MIPS32 指令集生成，反汇编过程使用工具 Objdump 完成。

文献[5]的测试结果显示，源代码级克隆检测工具 Nicad[14] 对级别 1 的代码克隆测试结果为“非常好”，级别 2 与级别 3 的代码克隆的测试结果均为“好”，并可实现对级别 4 的代码克隆的部分检测，其综合性能优于剩余 41 款检测工具。因此，本文选择 Nicad 与所提算法进行对比测试。

本文先使用 Linux 内核两个版本 kernel4.0.1 和 kernel 4.9.10 的二进制可执行文件来测试本文算法对克隆级别 0－级别 2 的检测性能。两个内核均使用 make allnoconfig 进行编译配置，编译后的内核大小分别为 1 383 228 字节（包含 4 860 个函数）与 1 749 508 字节（包含 5 824 个函数）。经过对 MIPS32 二进制可执行文件的人工抽样检查后发现，由编译器引入的沙砾指令的长度通常小于 10。同时，级别 3 的代码克隆所影响的行数范围通常为 1 到 5 行，对应的二进制指令约为 10 个。因此，本节实验将 D 值设为 10。

图 5 给出了当 D 值为 10 时，本文算法计算出的克隆数量随 BML 值和 R 值的变化情况。从图 5 中可以看出，BML 值为 15 时，检测出了 35 820 个克隆对；当 BML 值增加到 20 时，检测出的克隆对的数量减少到 6 931。另外，当 R 值从 0 增加到 0.25 时，克隆对数量有少量减少，说明当 R 值设置为 0.25 时，有部分克隆对进行了合并；当 R 值继续增大时，R 值几乎不再影响克隆对的数量。

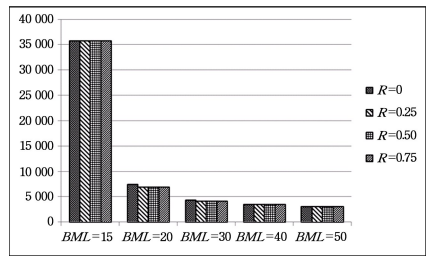

图 5　克隆对数量随 BML 值和 R 值的变化情况

Fig. 5　Number of CPs varing with BML value and R value

表 3 给出了当 D 值为 10 且 R 值为 0.25 时，本文算法在 BML 值分别为 15，20，30，40 和 50 时检测出的存在克隆对的函数对数量。使用 Nicad 对两个版本内核源代码进行分析（去除了两个版本中与 MIPS32 架构无关的底层代码文件），表 4 给出了当分析粒度分别设置为 10 行、15 行、20 行、30 行和 40 行（即源代码克隆对的长度不小于 10 行、15 行、20 行、30 行和 40 行）时，Nicad 所分析出的源代码中克隆对所在函数对的数量。

表 3　不同 BML 值对应的函数对数量

Table 3　Number of function pairs with different BML

BML	存在克隆对的函数对数量
15	14 609
20	2 626
30	1 240
40	802
50	483

表 4　使用 Nicad 对源码分析出的函数对数量

Table 4　Number of function pairs through detecting source code by Nicad

源代码行数	存在克隆对的函数对数量
10	1 395
15	796
20	514
30	255
40	138

图 6 进一步给出了表 3 与表 4 中本文算法与 Nicad 计算出的函数对的对应关系，其中两条曲线分别表示本文算法与 Nicad 所计算出的函数对数量；柱状图表示在对应 Nicad 行数设置和本文算法 *BML* 设置的条件下，两者所计算出的相同函数对数量占 Nicad 计算结果的比例。从图中可以看出，在 Nicad 行数设定为 10 行，*BML* 值设置为 15 时，两者结果的重合比例为 67.7%；当 Nicad 行数提升为 15 行，*BML* 值提升为 20 时，两者结果的重合比例为 68.1%；当 Nicad 行数值和 *BML* 值继续增加时，重合比例开始下降。经过人工分析，在 Nicad 行数为 15 行且 *BML* 值为 20 时，两者的差异部分中，对于 Nicad 检测出而本文算法未检测出的函数对，其主要原因在于 kernel 源代码文件中大量使用的条件编译语句在不满足编译配置选项时未被编译进二进制可执行文件；对于本文算法检测出而 Nicad 未检测出的函数对，其主要原因在于 Nicad 的检测对象是基于源代码中的大括号对将源代码分割为对应的代码片段，如果该代码片段中 70%以上的代码与其他代码片段相似，则将该代码片段对检测为克隆对，否则不进行记录。因此，相比于 Nicad，本文算法发现了更多更细粒度的克隆对。

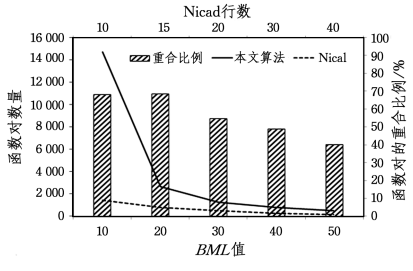

图 6 函数对的重合比例随 *BML* 值与 Nicad 行数的变化情况

Fig. 6 Matchingratio of function pair varing with *BML* value and Nicad line number

为了进一步测试本文算法对克隆级别 1—级别 4 的检测性能，对 C4 代码[24]（共 527 行，包含 4 个函数）使用 Stunnix C/C++ Obfuscator[25]工具修改，并且使用同样的编译器配置对修改前后的代码进行编译得到各自的二进制可执行文件。表 5 给出了本文对 C4 代码的修改明细以及对应的克隆级别，其中“其他”表示不属于 4 个函数的其他代码。

表 5 C4 代码的修改明细

Table 5 Modification details of C4 code

函数名称	克隆级别 1/处		克隆级别 2/处		影响行数/行	
	注释修改	换行符修改	变量名替换	常量替换	修改前	修改后
main()	28	12	484	17	197	185
expr()	1	3	560	14	150	147
next()	0	1	210	44	86	85
stmt()	0	2	68	10	49	47
其他	21	18	76	—	48	22

表 6 给出了本文算法、Nicad（检测级别设置为“Type3-2”）和 SimCad[26]（检测级别设置为“Type-3”）对修改前后的 C4 代码的检测结果，其中本文算法与 Nicad 对 C4 修改前后代码间的代码克隆全部检测成功，而 SimCad 只对函数 main() 的代码克隆进行了成功检测，其余 3 个函数的克隆检测全部失效。

表 6 对克隆级别 1 和级别 2 的测试结果

Table 6 Test results for clone level 1 and 2

函数名称	工具名称		
	本文算法	Nicad	SimCad
main()	全部	全部	全部
expr()	全部	全部	失效
next()	全部	全部	失效
stmt()	全部	全部	失效

为测试本文算法对克隆级别 3 的检测性能，向上述 4 个函数中插入 10 处代码（合计插入 30 行）和删除 10 处代码（合计删除 30 行），表 7 给出了本文算法、Nicad（检测级别设置为“Type3-2”）和 SimCad（检测级别设置为“Type-3”）的检测结果。测试结果表明，本文算法能将克隆级别 3 的克隆对全部检出，而 Nicad 与 SimCad 均只能检测出部分克隆对。

表 7 对克隆级别 3 的测试结果

Table 7 Test results for clone level 3

函数名称	工具名称		
	本文算法	Nicad	SimCad
main()	全部	全部	全部
expr()	全部	全部	全部
next()	全部	全部	失效
stmt()	全部	失效	失效

最后，本文对 C4 代码进行了满足克隆级别 4 的语义等价代码替换，包括使用 while 循环替换 for 循环，使用 switch 语句替换多重 if 语句，将多个变量定义合并在结构体中，使用 #define 替换函数内部实现。表 8 给出了本文算法、Nicad（检测级别设置为“Type3-2”）和 SimCad（检测级别设置为“Type-3”）的检测结果。测试结果表明，本文算法能将克隆级别 4 的克隆对全部检出，而 Nicad 与 SimCad 均不能检出任何克隆对。

表 8 对克隆级别 4 的测试结果

Table 8 Test results for clone level 4

函数名称	工具名称		
	本文算法	Nicad	SimCad
If/switch 替换	是	否	否
While/for 替换	是	否	否
#define 替换	是	否	否
变量合并	是	否	否

结束语 本文针对无法获取源代码的应用场景，设计并实现了一种新的二进制可执行代码克隆检测方法，提取二进制代码的操作类型序列作为分析对象，利用后缀树对后缀字符序列的表达能力进行序列间相似部分的检索，并给出了相应的定理证明。此外，使用了合并沙砾指令的方法对相邻克隆对进行处理，以提升对语句增加、修改或删除的克隆对的检测能力，并降低了编译器的优化策略对结果的干扰。测试结果表明，本文算法在缺少源代码的场景下同样能进行细粒度的克隆分析，并且在级别 0—级别 2 的代码克隆对比测试中获得了与 Nicad 一致的结果；级别 3 与级别 4 的代码克隆对比测试结果显示，本文算法的性能优于 Nicad。

未来将结合代码的结构特征进一步优化检测算法，以降低检测结果的假阳性；另外，还需进一步测试本文算法对不同架构和指令集的二进制代码分析的普适性。

参 考 文 献

[1] United States Court of Appeals for the Federal Circuit: JACOB-

SENV. KATZER [A/OL]. (2018-03-27)[2018-07-07]. http://www.cafc.uscourts.gov/sites/default/files/opinions-orders/08-1001.pdf.

[2] LV H. Computer software copyright infringement judge[D]. Lanzhou: Lanzhou University, 2015. (in Chinese)
吕浩. 计算机软件著作权侵权的判定研究[D]. 兰州：兰州大学，2015.

[3] DONG H, TANG Q. Review of Jacobsen V. Katzer Case: The Influence of the Differences Between Chinese and American Copyright Systems on the Nature of Open Source Agreements and Its Enlightenment[J]. China Copyright, 2009, 2009(3): 48-51. (in Chinese)
董皓，唐磬. Jacobsen V. Katzer 案评述：中美版权制度差异对开源协议性质的影响及启示[J]. 中国版权，2009，2009(3)：48-51.

[4] SU Q, WU W M, LI Z L, et al. Research and Application of Chaos Opaque Predicate in Code Obfuscation[J]. Computer Science, 2013, 40(6): 155-159. (in Chinese)
苏庆，吴伟民，李忠良，李景樑，陈为德. 混沌不透明谓词在代码混淆中的研究与应用[J]. 计算机科学，2013，40(6)：155-159.

[5] ROY, CHANCHAL K, CORDY, et al. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach[J]. Science of Computer Programming, 2009, 74(7): 470-495.

[6] RATTAN D, BHATIA R, SINGH M. Software clone detection: A systematic review[J]. Information & Software Technology, 2013, 55(7): 1165-1199.

[7] SU X H, ZHANG F L. A Survey for Management Oriented Code Clone Research[J]. Chinese Journal of Computers, 2018, 41(3): 628-651. (in Chinese)
苏小红，张凡龙. 面向管理的克隆代码研究综述[J]. 计算机学报，2018，41(3)：628-651.

[8] CHEN H, GUO T, CUI B J, et al. Comparison and analysis on binary file similarity detection technique[C] // Proceedings of Conference on Vulnerability Analysis and Risk. Beijing: China Information Technology Security Evalution Center, 2011.

[9] JOHNSON J H. Substring matching for clone detection and change tracking[C] // Proceedings of International Conference on Software Maintenance, 1994. New York: IEEE, 1994: 120-126.

[10] KAMIYA T, KUSUMOTO S, INOUE K. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code[J]. IEEE Transactions on Software Engineering, 2002, 28(7): 654-670.

[11] SAINI V, SAJNANI H, KIM J, et al. SourcererCC and SourcererCC-I: tools to detect clones in batch mode and during software development[C] // Proceedings of the 38th Proceedings of International Conference on Software Engineering Companion, 2016. New York: IEEE, 2016: 597-600.

[12] SUDHAMANI M, RANGARAJAN L. Code clone detection based on order and content of control statements[C] // Proceedings of International Conference on Contemporary Computing & Informatics. New York: IEEE, 2017: 59-64.

[13] WANG M, WANG P, XU Y. CCSharp: An Efficient Three-Phase Code Clone Detector Using Modified PDGs[C] // Proceedings of Asia-pacific Software Engineering Conference. New York: IEEE, 2018: 100-109.

[14] ROY C K, CORDY J R. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization[C] // Proceedings of IEEE International Conference on Program Comprehension. Washington DC: IEEE Computer Society, 2008: 172-181.

[15] WHITE M, TUFANO M, VENDOME C, et al. Deep learning code fragments for code clone detection[C] // Proceedings of IEEE/ACM International Conference on Automated Software Engineering. NewYork: IEEE, 2016: 87-98.

[16] RAGKHITWETSAGUL C, KRINKE J, MARNETTE B. A picture is worth a thousand words: Code clone detection based on image similarity[C] // Proceedings of International Workshop on Software Clones. NewYork: IEEE, 2018: 44-50.

[17] SÆBJØRNSEN A, WILLCOCK J, PANAS T, et al. Detecting code clones in binary executables[C] // Proceedings of Eighteenth International Symposium on Software Testing and Analysis. New York: ACM, 2009: 117-128.

[18] DONG M, ZHUANG H, ZHANG R, et al. A New Method of Software Clone Detection Based on Binary Instruction Structure Analysis[C] // Proceedings of International Conference on Wireless Communications. NewYork: IEEE, 2013: 1-4.

[19] HU Y K, ZHANG Y Y, LI J R, et al. Binary code clone detection across architectures and compiling configurations[C] // Proceedings of International Conference on Program Comprehension. New York: IEEE, 2017: 88-98.

[20] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification[S/OL]. http://refspecs.linuxbase.org/elf/elf.pdf.

[21] MIPS32® Instruction Set Quick Reference [S/OL]. https://www.mips.com/?do-download=mips32-instruction-set-quick-reference-v1-01.

[22] MCCREIGHT E M. A Space-Economical Suffix Tree Construction Algorithm[J]. Journal of the Acm, 1976, 23(2): 262-272.

[23] UKKONEN E. On-line construction of suffix trees[J]. Algorithmica, 1995, 14(3): 249-260.

[24] rswier c4: C in four functions[CP/OL]. (2017-08-22) [2018-08-22]. https://github.com/rswier/c4.git.

[25] C/C++ Obfuscator[EB/OL]. (2018-08-09)[2018-08-09] http://stunnix.com/prod/cxxo/.

[26] UDDIN M S, ROY C K, SCHNEIDER K A. SimCad: An extensible and faster clone detection tool for large scale software systems[C] // Proceedings of International Conference on Program Comprehension. NewYork: IEEE, 2013: 236-238.