doi:10.11959/j.issn.1000-436x.2016225

基于 simhash 与倒排索引的复用代码快速溯源方法

乔延臣[1,2,3]，云晓春[1,2,3]，庹宇鹏[2,3]，张永铮[2,3]

(1. 中国科学院计算技术研究所，北京 100080；
2. 中国科学院研究生院，北京 100039；3. 中国科学院信息工程研究所，北京 100093)

摘 要：提出了一种新颖的复用代码精确快速溯源方法。该方法以函数为单位，基于 simhash 与倒排索引技术，能在海量代码中快速溯源相似函数。首先基于 simhash 利用海量样本构建具有三级倒排索引结构的代码库。对于待溯源函数，依据函数中代码块的 simhash 值快速发现相似代码块，继而倒排索引潜在相似函数，依据代码块跳转关系精确判定是否相似，并溯源至所在样本。实验结果表明，该方法在保证高准确率与召回率的前提下，基于代码库能快速识别样本中的编译器插入函数与复用函数。

关键词：网络安全；复用代码；快速溯源；同源判定；恶意代码

中图分类号：TP393.08 **文献标识码**：A

Fast reused code tracing method based on simhash and inverted index

QIAO Yan-chen[1,2,3], YUN Xiao-chun[1,2,3],TUO Yu-peng[2,3], ZHANG Yong-zheng[2,3]

(1. Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, China;
2. Graduate School, Chinese Academy of Sciences, Beijing 100039, China;
3. Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China)

Abstract: A novel method for fast and accurately tracing reused code was proposed. Based on simhash and inverted index, the method can fast trace similar functions in massive code. First of all, a code database with three-level inverted index structures was constructed. For the function to be traced, similar code blocks could be found quickly according to simhash value of the code block in the function code. Then the potential similar functions could be fast traced using inverted index. Finally, really similar functions could be identified by comparing jump relationships of similar code blocks. Further, malware samples containing similar functions could be traced. The experimental results show that the method can quickly identify the functions inserted by compilers and the reused functions based on the code database under the premise of high accuracy and recall rate.

Key words: network security, reused code, retrieval method, homology identification, malware

1 引言

代码复用通常以函数为基本单位，即使被编译器高度优化仍然保留大量函数整体，所以，本文以函数为单位进行溯源更加符合复用场景。恶意代码同源判定的主要依据是恶意代码作者在不同恶意代码中对个人编写代码的复用，如 Sasser 与 Netsky[1]、Flame 与 Gauss 等[2]的同源判定均依据它们共享的特殊函数。但是，为提高开发速度，恶意代码作者经常复用他人编写的公开或半公开代码，

收稿日期：2016-05-12；**修回日期**：2016-10-11
通信作者：庹宇鹏，tuoyupeng@iie.ac.cn
基金项目：国家自然科学基金资助项目（No.61303261）；国家高技术研究发展计划（“863”计划）基金资助项目（No.2013AA014703，No.2012AA012803）；国家 242 信息安全计划基金资助项目（No.2014A094）；中国科学院战略性科技先导专项基金资助项目（No.XDA06030200）

Foundation Items: The National Natural Science Foundation of China (No. 61303261), The National High Technology Research and Development Program of China (863 Program) (No. 2013AA014703, No. 2012AA012803), The National 242 Information Security Research Program of China (No. 2014A094), The Strategic Priority Research Program of the Chinese Academy of Sciences (No.XDA06030200)

如 Chthonic[3]是一款在 Zeus[4]源码基础上修改开发的恶意代码。据报告，Equation[5] 提出的高级持续性威胁(APT，advanced persistent threat)攻击中使用的一个样本，被判定属于 Zlob 家族，说明 APT 攻击组织也会复用开源代码。为执行需要，编译器在编译时通常插入大量函数代码。经测试，当编译仅有一个函数的 C 语言代码时，Windows 下的 VC6.0 编译器插入了 103 个函数，Linux 下的 GCC4.7.2 编译器插入了 18 个函数。不同的编译器插入的函数与函数的插入位置均不同，需要大量经验与技巧才能识别这些函数。复用函数对恶意代码分析与同源判定工作造成了很大干扰，目前，主要依靠恶意代码分析人员的经验识别，导致同源判定效率不高。快速识别复用函数将大大提高效率，并提升同源判定结论的可信度。

复用函数溯源的基础是相似函数判定，若在某样本中存在一个函数的相似函数，则说明该函数为复用函数。目前，大多相似函数判定技术具有很高的准确率与召回率，但是判定效率较低，不适应海量代码的复用函数溯源，Qiao 等[6]提出了一种快速溯源方法，但未考虑代码块间的跳转关系。一个函数源码的少量修改，编译选项、所在位置的不同都会造成逆向后汇编代码中指令顺序、寄存器、跳转位置等的差异，因此，若使用散列等方法进行溯源将导致非常低的召回率。在函数中，代码块的跳转结构是相似判定的重要特征，而跳转关系提取、结构图的比对要耗费大量时间，是导致目前相似判定准确率、召回率与速度难以兼得的一个重要原因。

本文提出了一种基于 simhash 与倒排索引的复用函数快速溯源方法。核心思想是先基于相似代码块缩小相似函数判定范围，再应用基于跳转关系的精确比对方法过滤出相似函数。主要包括预处理、索引构建与代码溯源 3 个阶段。

本文主要贡献有：1) 提出一种基于 simhash 的相似代码块快速检索方法；2) 提出一种复用函数快速溯源方法，根据该方法将有助于开发代码搜索引擎，大大提高同源判定效率。

2　相关研究工作

目前，同源判定主要依靠人工分析、发现并比对其中的特殊相似函数，这些函数在最终同源证据链中占有很大比重。董志强等[1]发现 Sasser 与 Netsky 二者拷贝自身和修改注册表的功能、实现思路和代码相似。CrySyS 实验室[7]发现 Duqu 与 Stuxnet 的 dll 文件中具有多个相似的导出函数，driver 文件中也存在大量相似函数。Kaspersky 实验室[2]发现 Flame 与 Gauss 的 C&C 服务功能函数、字符串的初始化函数、字符串解密函数等相似。Cloud Atlas 与 RedOctober[8]的压缩算法实现函数相似。Pitou 与 Srizbi[9]使用了相同的域名生成函数。本文对其中一些样本进行了人工分析，发现除报告中公布的作为同源判定证据的相似代码外，还有很多相似代码，但是这些相似代码并没有用作同源判定证据，据此推测，业界分析人员认为这些代码可能属于编译器插入代码，或者复用的其他代码，不具备同源证明能力。

上述相似代码判定工作在特征选择、精确判定、效率提升等方面做了很多贡献。Myles 等[10]使用反汇编后的 *k*-gram 指令序列表示模块，其中，*k*=4 或 5，之后根据相同序列的数量判定是否相似，经实验证明，该方法的漏报率与误报率均非常低，可用于软件盗版检测等。Sæbjørnsen 等[11]提出了一种汇编代码的中间表示形式，并使用 *n*-gram 序列表示一个函数，之后利用精确检测与 LSH 算法检测，发现二进制可执行文件中的克隆代码，该方法准确快速地发现了 WinXP 系统文件中的克隆代码。Lakhotia 等[12]开发了一个将代码片段中的指令、寄存器、立即数替换并转换成与原代码块语义相似的 BinJuice 方法，用于解决混淆代码的相似判定问题，经实验验证该方法能有效应对编译之后的混淆技术。随后，Ruttenberg 等[13]利用该方法实现了恶意代码取证中的共享模块识别方法，通过 2 次聚类，能快速识别恶意代码中的相似模块。Ouellette 等[14]基于 BinJuice 方法开发了应对恶意代码进化的半监督算法，该算法使用深度学习方法学习恶意代码的本质属性，基于本质属性发现同类别的新型恶意代码。David 等[15]提出了基于指令切片的相似代码搜索方法，经实验验证具有较高的准确率与召回率。Alrabaee 等[16]提出了 SIGMA 方法，该方法将多种特征如控制流图(CFG，control flow graph)、寄存器流图(RFG，register flow graph)、函数调用图(FCG，function call graph)等融合组成新的表示方法，用于代码的相似判定，经案例验证该方法具有非常好的判定效果。Qiao 等[6]提出的复用代码溯源方法，将 simhash 应用于相似判定，速度较快，但忽略了代

码块间的跳转关系，易于造成判定失误。

simhash 是局部敏感散列(LSH，locality sensitive hashing)算法的一种，最早由 Charika 等[17]在 2002 年提出，谷歌的 Manku 等[18]在 2007 年将该算法用于海量相似网页去重，根据该算法为每个网页计算一个 64 bit 的 simhash 值，simhash 值的汉明距离在 3 以内的网页都认为是相似的，Manku 还提出了基于抽屉原理的时间、空间均较优的特定汉明距离 simhash 值的快速检索方法。目前，simhash 算法被应用在多个方面，尤其源代码克隆领域。Uddin 等[19]提出将 simhash 算法用于大规模软件系统的源代码克隆检测，并提出基于该方法开发代码搜索引擎与克隆管理工具。郭等[20]同样将 simhash 算法用于大规模代码的克隆检测，达到了较高的准确率。Qiao 等[6]验证了 simhash 应用于代码判定的可行性。

基于以上工作，本文提出一种先利用 simhash 算法与倒排索引进行搜索以缩小判定范围，再利用精确比对完成溯源的方法。

3 方法描述

复用函数快速溯源方法总体架构如图 1 为所示，共包括 3 个阶段：预处理阶段、索引构建阶段与函数溯源阶段。预处理阶段：输入样本，逆向获取样本的汇编代码，提取其中的函数，依据跳转结构将函数划分为多个代码块，并计算代码块的 simhash 值。索引构建阶段：依据倒排索引，构建 simhash 值与代码块、代码块与函数、函数与所在样本的索引关系。函数溯源阶段：对某函数利用 simhash 算法检索与其代码块相似的代码块，根据倒排索引定位潜在相似函数，最终依据精确相似判定方法确定是否相似，并定位相似函数所在的样本完成函数溯源。

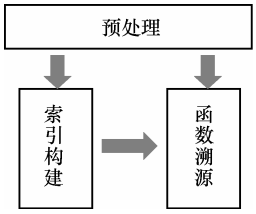


图 1 函数溯源方法阶段架构

3.1 预处理

预处理阶段负责汇编代码获取、函数代码提取、代码块切分以及 simhash 值的计算，具体如图 2 所示。首先，使用 IDA Pro 逆向未加壳或脱壳样本获取汇编代码，表示为 S；其次，依据汇编文件中的特殊标识，将其拆分成多个函数，函数用 P 表示，一个样本表示为多个函数的集合 $S=\{P_1, P_2,\cdots, P_n\}$；之后，依据函数中的 jnz、jmp 等跳转指令将函数切分为多个代码块，用 L 表示，每个函数表示为多个代码块的集合 $P=\{L_1, L_2,\cdots, L_m\}$；然后，将代码块中的指令进行标准化处理，以忽略由寄存器、内存地址等的不同造成的差异；最后，为每个标准化代码块计算 simhash 值，simhash 值在本文表示为 sh。经预处理阶段，最终每个函数表示为 simhash 值的集合 $P\rightarrow\{L_1, L_2,\cdots, L_m\}\rightarrow\{sh_1, sh_2,\cdots, sh_m\}$。

下面对函数提取与拆分、代码标准化处理以及 simhash 算法进行详细介绍。

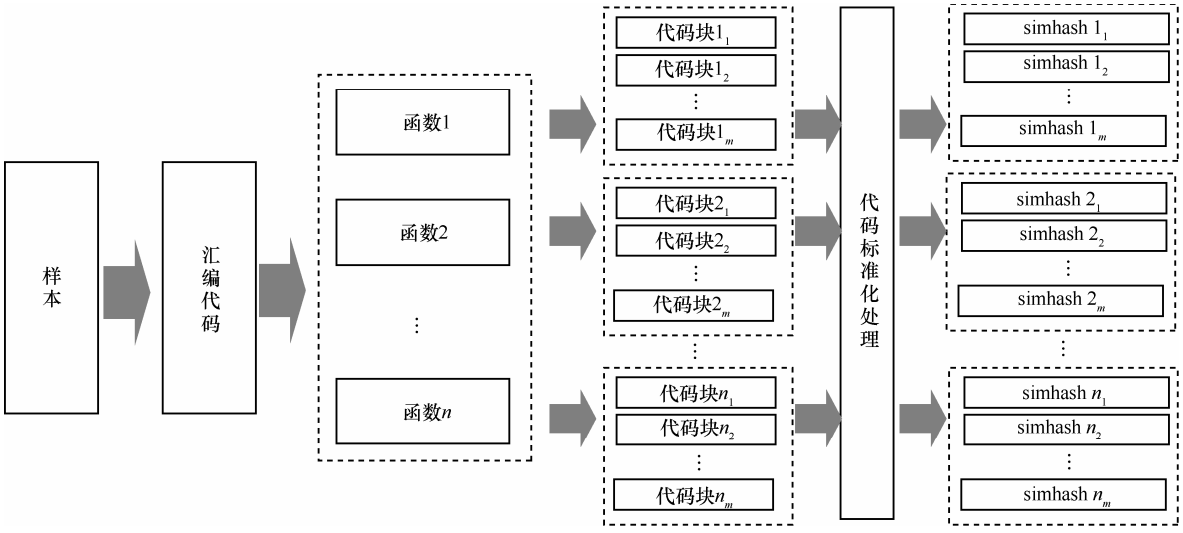


图 2 预处理流程

3.1.1　函数提取与拆分

逆向获取的汇编代码中，“proc near”标识函数的开始，而“endp”标识函数结束，依据这 2 个标识将汇编代码拆分为多个函数。对内部有 jnz、jmp 等跳转指令的函数，依据跳转指令及跳转位置，将函数划分为多个代码块，而没有跳转指令的则将函数主体看作一个代码块。函数 sub_4023BC 的代码为如下。

```
    proc near
    push    esi
    mov     esi, [esp+4+arg_0]
    push    0
    and     dword ptr [esi], 0
    call    ds:GetModuleHandleA
    cmp     word ptr [eax], 5A4Dh
    jnz     short loc_4023E7
loc_4023E7
    pop     esi
    retn
sub_4023BC
    endp
```

函数 sub_4023BC 具有跳转指令 jnz，跳转到代码块的开始位置 loc_4023E7，根据跳转指令“jnz”与跳转地址“loc_4023E7”将其划分为 2 个代码块，对第一个代码块，依据函数位置命名，如表 1 所示。

表 1　　代码块拆分示例

代码块名称	内容
loc_4023BC	push esi mov esi, [esp+4+arg_0] push 0 and dword ptr [esi], 0 call ds:GetModuleHandleA cmp word ptr [eax], 5A4Dh jnz short loc_4023E7
loc_4023E7	pop esi retn

3.1.2　代码标准化处理

对源代码的轻微改动将造成汇编代码中寄存器、立即数、内存地址的大幅变动，为了忽略这种差异对代码造成的影响，依据以下规则对汇编代码进行标准化处理。

1) 寄存器如 eax、ax、al 等依据所占位数分别标准化为 REG32、REG16、REG18。

2) 内存如[eax]、[edi+4]等均表示为 MEM。

3) 立即数如 0、5A4Dh 表示为 VAL。

4) call 指令调用外部的系统库函数时指令不做处理，调用内部函数如“call sub_105A8”时规范化为“call sub_xxx”。

5) 跳转指令如“jz short loc_4023E7”规范化为“jz　loc_xxx”。

以代码块 loc_4023BC 为例，标准化处理具体如表 2 所示。

表 2　　代码标准化处理示例

原代码块	标准化处理代码块
push esi	push REG32
mov esi, [esp+4+arg_0]	mov REG32, MEM
push 0	push VAL
and dword ptr [esi], 0	and MEM, VAL
call ds:GetModuleHandleA	call GetModuleHandleA
cmp word ptr [eax], 5A4Dh	cmp　MEM, VAL
jnz short loc_4023E7	jnz loc_xxx

3.1.3　simhash 计算

一个文本对象需要经过分词、散列、加权、合并、降维等 5 个步骤，最终得出 N bit 的 simhash 值。Manku[18]将长度值 N 设定为 64，算法具体过程如下。

输入　文本 txt

输出　64 bit simhash 值

1) 创建一个 64 bit 的向量，并初始化为 0。

2) 对文本 txt 进行分词处理，一般有 2 种方式：n-gram 字符串或 n-gram 单词。

3) 为每个分词赋予权值：通常基于频率，即分词出现的次数。

4) 对每个分词做散列处理，得出 64 bit 散列值：通常使用 MD5 或 SHA1 散列算法，然后取其中 64 bit。

5) 加权合并：对分词散列的每一位，如果该位为 1，则向量相应位的值加上该分词的权值，否则减去该分词的权值。

6) 降维：对向量的每一位，若该位大于 0，则设为 1，否则设为 0，形成一个 64 bit 的 simhash 值。

由于汇编代码与文本、网页等具有不同的特性，

为此，对 simhash 的传统算法做了如下 2 点改动。

分词：汇编代码以指令为基础，一条指令包括助记符、寄存器等，由于这种格式的特殊性，*n*-gram 字符串或 *n*-gram 单词都难以反映代码的语义特征，因此本文使用 2-gram 标准化的指令序列分词，如表 2 中的指令将被序列化为为“push REG32; mov REG32, MEM”、“mov REG32, MEM; push VAL”、“push VAL; and MEM, VAL”等。

权值：在文本分类中一般依据分词的频率、属性计算权值，在汇编代码块的 simhash 计算中，将分词的频率作为基础权值，由于调用的 API 在很大程度上决定了函数的功能，所以调用指令在代码块中具有重要作用，因此对包含调用指令分词的权值加倍。

根据以上算法，本文将标准化后的代码块视为一个文本，应用 simhash 算法将每个代码块映射为一个 64 bit 的 simhash 值。

3.2 索引构建

倒排索引(inverted index)记录的是关键词与具有该关键词的各记录项。这种索引方式在搜索引擎中普遍使用，由关键词索引该词所在文档，加快了检索速度。

根据预处理阶段获取的样本与函数、函数与代码块、代码块与 simhash 值，构建它们的倒排索引。主体倒排索引分 3 层，如图 3 所示。

1) simhash 值索引具有该 simhash 值的代码块，simhash 碰撞概率较大，所以存在几个代码块具有同一个 simhash 值的情况。

2) 代码块索引代码块所在的函数，代码块标准化后，增大了相似的概率，不相似的函数也有可能存在相同的标准化代码块。

3) 函数索引函数所在的样本。

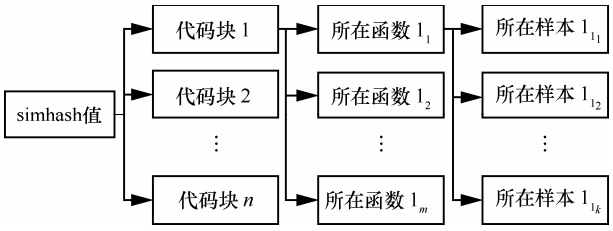


图 3 倒排索引结构

按照 Manku[18]的论述，simhash 值的汉明距离(对应位上数值不同的位数)在 3 以内的都可以认为相似，但是在海量的 simhash 值列表中搜索汉明距离为 3 以内的 simhash 值复杂度非常高，为提高效率，Manku[18]提出了兼顾时间与空间的多表索引方法。该方法基于抽屉原理，若 2 个 simhash 值的汉明距离在 3 以内，则它们切分成的 $l(l\leqslant 64)$块中，必有 $l-3$ 块相等。为提高检索效率，同时兼顾空间开销，本文设 $l=8$，将每个 simhash 值均分为 8 块，为所有的 simhash 值创建 8 个表，不同的表存储不同位置的块，如第一个表存储 0~7 位，第二个表存储 8~15 位，第三个表存储 16~23 位等，表中同样使用倒排索引，由 8 位比特串索引 simhash 值。

当依据某 simhash 检索汉明距离在 3 以内的其他 simhash 值时，将该 simhash 平分为 8 块，每块在相应的表寻找相似块，取相似块对应的 simhash 集合，筛选出至少在 5 个块对应的集合中出现的 simhash 值，然后逐一计算汉明距离。假如 simhash 平均分布，则该方法使汉明距离计算次数减至总数的 0.375 次方，在本文实验中，WinXP 文件构建的代码库中共有 3 242 151 个 simhash 值，通过该方法，计算次数平均减至 276 次，大大提高了溯源速度。

本阶段共构建了 2 类索引，一类 simhash 索引用于快速检索汉明距离在 3 以内的 simhash 值；一类代码索引用于快速定位相似函数及其所在的样本。

3.3 函数溯源

函数溯源包括 2 个过程：函数检索与相似判定，首先依据函数中代码块的 simhash 值检索出潜在的相似函数，之后依据精确比对方法进行相似判定，并溯源至所在样本。

3.3.1 函数检索

函数检索流程如图 4 所示。

1) 溯源函数代码 P，依据跳转指令拆分为多个代码块，假设共有 n 个，之后计算每个代码块的 simhash 值为

$$P\rightarrow\{sh_1, sh_2, \cdots, sh_n\} \tag{1}$$

2) 用 simhash 多表索引方法，为 $sh_i|i\in[1,n]$快速检索汉明距离在 3 以内的相似 simhash，构成相似 simhash 集合为

$$shSet_i=\{sh:d(sh_i,sh)\leqslant 3\}|i\in[1,n] \tag{2}$$

其中，$d(sh_i, sh)$表示 sh_i 与 sh 的汉明距离，若所有的相似 simhash 集合均为空集，说明不存在与该函数具有相似代码块的函数，否则，继续下一步。

3) 据已构建的 simhash 与代码块的倒排索引，

检索 simhash 值属于 $shSet_i$ 的代码块 L，构成相似代码块集合

$$LSet_i = \{L: simhash(L) \in shSet_i\} \mid i \in [1,n] \quad (3)$$

其中，$simhash(L)$表示代码块 L 的 simhash 值。

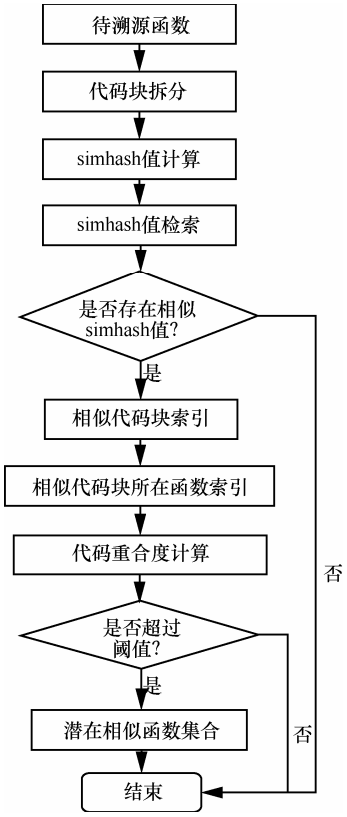


图 4　代码检索流程

4) 依据已构建的代码块与所在函数的倒排索引，检索相似代码块所在的函数 P'，构成函数集合为

$$PSet_i = \{P': \exists L \in P' \wedge L \in LSet_i\} \mid i \in [1,n] \quad (4)$$

5) 对每个检索到的函数，计算该函数 P'在待溯源函数 P 中的代码重合度，即 P'与 P 的相似代码在 P 中的占比，表示为 $iSim(P, P')$，如式(5)所示。

$$iSim(P, P') = \sum_{\exists L \in P' \wedge L \approx Li} \frac{len(L_i)}{len(P)} \quad (5)$$

其中，$len(P)$表示函数 P 的总指令数，$L \approx L_i$ 表示代码块 L 与 L_i 相似，$len(L_i)$表示代码块 L_i 的指令数，通过实验发现当函数的代码重合度不小于 0.5 时最有可能与待溯源函数代码相似，因此最终得出潜在相似函数集合为

$$PSet = \{P': iSim(P, P') \geqslant 0.5\} \quad (6)$$

若潜在相似函数集合 $PSet$ 为空集，说明不存在与待溯源函数具有足够相似代码的函数，难以认定它们相似，待溯源函数在已有代码集上可认为是原创函数。

3.3.2　精确相似判定

使用 simhash、倒排索引等方法检索出潜在的相似函数集合 $PSet$。由于函数的代码块具有跳转结构，不同的跳转将导致很大的差异，存在相似代码块并不一定意味着相似，因此，需要进一步精确判定。上一阶段检索出的相似函数，与待溯源函数可能只有一个相似代码块，也可能有多个，对于只有一个相似代码块的，直接判定为相似，对于有多个相似代码块的，将依据代码块间的跳转关系等进行相似判定。恶意代码中的复用情况普遍且多样，虽然，以函数为复用基本单位，但也存在只复用部分代码或者对原函数增加功能代码的情况，所以，部分相似也是相似的一种情况。对多个代码块相似的，根据代码块间的跳转关系，形成跳转关系矩阵，进而生成跳转关系序列，如图 5 所示。

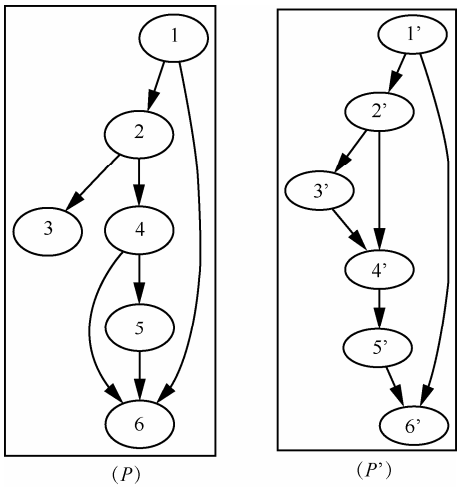


图 5　函数代码块跳转关系

图 5 是 2 个函数 P 与 P'的代码块间的跳转关系图，假设函数 P 的 2、3、4、6 代码块与函数 P'的 2'、3'、4'、6'代码块分别相似，则提取这几个代码块在 2 个函数中的的跳转关系矩阵，如图 6 所示。

	2	3	4	6
2	0	1	1	0
3	0	0	0	0
4	0	0	0	1
6	0	0	0	0

	2'	3'	4'	6'
2'	0	1	1	0
3'	0	0	1	0
4'	0	0	0	0
6'	0	0	0	0

图 6　跳转关系矩阵

其次，将 2 个矩阵分别表示为 16 bit 的比特串：0110000000100000 与 0110001000000000，当 2 个比特串相等时说明跳转关系相同，判定 2 个函数相似，否则不相似。该方法对函数的部分代码相似能进行有效判定。

4 实验评估

4.1 准确率与召回率

首先手工提取发现相似函数，其次利用这些函数评估本文方法的准确率与召回率。

4.1.1 相似函数数据

Linux 系统下的 gcc 编译器有不同的优化级别 O、O_1、O_2、O_3、O_s 等，不同的优化级别对代码的优化程度不同。分别使用 gcc-O_2 与 gcc-O_3 这 2 个级别的优化选项编译 ssh-3.2.9.1[21]，获取其中的二进制可执行文件 scp_2，分别命名为 scp_2_O_2、scp_2_O_3。

scp_2_O_2 包含 1 939 个函数，scp_2_O_3 包含 1 938 个函数，对 2 个文件中的函数进行手工相似判定，发现其中二者函数名相同且相似的有 1 033 对，函数名不同但相似的有 515 对。名字不同但相似的函数包括 2 种情况：1) 内嵌其他函数的代码，导致代码主体相似，如 ssh_file_copy_file_get_attributes、ssh_buffer_clear、ssh_file_copy_file_get_dir_entries、ssh_adt_clear、ssh_file_client_get_version 等多个函数直接或间接调用了同一个函数 ssh_generic_assert，而 gcc 编译时将 ssh_generic_assert 的函数代码嵌入了这些函数中，导致它们虽然功能不同但是非常相似；2) 存在相似子图，如 ssh_file_client_remove 与 ssh_file_client_mkdir 等。

4.1.2 评估结果

提取 scp_2_O_2 中的函数作为基础代码库，并构建索引结构，然后对 scp_2_O_3 中的每个函数进行溯源，结果如表 3 所示。

表 3 溯源结果

已知	溯源	未溯源
相似	1 486	62
不相似	131	—

根据表 3 中的溯源结果，该方法溯源准确率为 $\frac{1\,486}{1\,617}$=91.90%，召回率为 $\frac{1\,486}{1\,548}$=95.99%。该实验结果表明，本文的溯源方法具有很高的准确率与召回率。

4.2 编译器函数溯源

编译器常常插入大量函数，这些函数与恶意代码的功能意图几乎无关联关系，快速识别它们将大大提高分析人员的逆向分析效率。

4.2.1 WinXP 系统文件

利用 32 位 WinXP 系统中“Program Files”与“Windows”文件夹下的所有 PE 文件构建基础代码库，该代码库共包含 2 689 272 个函数，30 151 841 个代码块。

4.2.2 溯源实验与结果

使用 VC6.0 编译器将示例代码编译为 32 位的 Release 版可执行文件，逆向获取该文件的汇编代码，IDA Pro 能自动识别并剔除库函数，所以最终会被代码中除 main 函数外还有 19 个编译器插入函数。

示例代码：

```
#include <stdio.h>
void main() {
    printf("Hello World\n");
}
```

由于 WinXP 系统中存在大量 VC6.0 编译的文件，据此推测 19 个编译器插入函数有一定概率在 WinXP 文件构建的代码库中溯源到相似函数，因此对 19 个编译器插入函数进行溯源，发现其中 16 个存在相似函数，另外 3 个（sub_401010、sub_4057BC 与 sub_402AD1）未溯源到相似函数。在配有 16 核 Intel(R) Xeon(R) CPU E562 与 16 GB 内存的 DellPower Edge R410 服务器上进行该实验，每个函数的平均溯源时间约为 0.149 s。表 4 列出了部分溯源到的相似函数。之后对溯源到的相似函数进行了手工相似判定，发现 sub_4 045 DC 溯源的一个函数不相似，即错误溯源结果，其他均正确。该实验说明本文方法能快速精确的溯源编译器插入代码。

4.3 复用函数溯源

恶意代码作者常常复用开源代码或者前期工作代码，快速识别这些代码中的函数，将有助于恶意代码的同源判定。

4.3.1 恶意代码样本集

vxheaven[22]网站上有个公开的恶意代码样本集，包含多个家族的共 271 092 个样本，利用这些样本构建基础代码库，该代码库共包含 6 391 053 个函数，13 990 583 个代码块。

4.3.2 样本 Agobot.hw 溯源实验与结果

Agobot[23]是一款开源恶意代码，从 Agobot 恶意代码家族中挑选出 VC6.0 编译的一个样本，该样

本被 Kaspersky 的反病毒引擎样本命名为 Backdoor. Win32. Agobot.hw。根据 Agobot 已经被开源推测，vxheaven 样本库中可能存在复用 Agobot 源码的样本，通过对 Backdoor. Win32. Agobot.hw 样本中函数的溯源，建立样本间的复用关系。首先，对 Backdoor.Win32.Agobot.hw 中的函数在基于 WinXP 文件构建的代码库中进行溯源，其中，372 个函数溯源到了相似函数，将这些函数从样本 Backdoor. Win32.Agobot.hw 中剔除；其次，对 Backdoor. Win32. Agobot.hw 中的其他函数在基于 vxheaven 样本库构建的代码库中进行溯源，通过溯源到的相似函数发现与非 Agobot 家族样本间的复用关系，表 5 列出了具有复用关系的前 10 个样本。

表 4　　编译器函数溯源详情

编译器函数	WinXP 系统文件中的相似函数
sub_40451A	imjputyc.dll 中的___old_sbh_decommit_pages 等
sub_402A5B	TPPS.DLL 中的 sub_10005660 等
sub_4023E9	TPVMW32.dll 中的 sub_10006046 等
sub_4044C4	MSCOMCTL.OCX 中的 sub_27608F15 等
sub_4023BC	tprdpw32.dll 中的 sub_10009720 等
sub_404380	imjpuex.exe 中的___old_sbh_new_region 等
sub_4045DC	tprdpw32.dll 中的 sub_1000A73D 等
sub_404880	msvcr70.dll 中的___old_sbh_alloc_block_from_page 等
sub_402531	tprdpw32.dll 中的 sub_10009895 等
sub_404678	imjpdct.dll 中的___old_sbh_alloc_block 等
sub_402E2A	imjprw.exe 中的_calloc 等
sub_403BA2	imjpmig.exe 中的___sbh_free_block 等
sub_40292A	imjpcus.dll 中的__heap_alloc 等
sub_402799	imskdic.dll 中的__NMSG_WRITE 等
sub_404633	TPVMW32.dll 中的 sub_1000708D 等
sub_401233	tprdpw32.dll 中的 sub_1000F109 等

表 5　　Agobot 样本溯源实验结果

非 Agobot 家族样本	相似函数数量	总指令数
Net-Worm.Win32.Kolabc.eph	44	2 411
Trojan-Spy.Win32.SCKeyLog.fp	44	1 048
Trojan-Spy.Win32.SCKeyLog.ij	43	1 141
Backdoor.Win32.IRCBot.ol	40	4 036
Trojan-Dropper.Win32.Agent.tif	31	712
Trojan-Spy.Win32.SCKeyLog.fb	29	669
Trojan-Downloader.Win32.Delf.gij	28	711
Backdoor.Win32.SuperSpy.b	28	665
Trojan-Downloader.Win32.Realtens.h	28	619
Trojan-PSW.Win32.Zombie.10	26	646

之后对这 10 个样本进行手工分析，发现除样本 Backdoor.Win32.IRCBot.ol 是被 VC7.0 编译外，其他 9 个样本均是被 VC6.0 编译。这 10 个样本与 Backdoor.Win32.Agobot.hw 确实存在大量相似函数，但暂时无法确定复用的主从关系。该实验说明本文方法能快速精确地溯源复用函数代码。

4.3.3　Zlob 家族溯源实验与结果

Zlob 是一款经典的木马病毒，危害巨大。通过溯源其样本中的函数代码，发现 Zlob 与其他恶意代码家族的关联关系，对恶意代码检测分类等具有重要作用。从 vxheaven 网站收集的 271 092 个样本中选出 35 个 Zlob 家族恶意代码样本，从中提取了 2 965 个函数，然后花费了 1 042.59 s 在基于 vxheaven 样本库构建的代码库中进行溯源，进而检索到包含相似代码的样本及其所属家族。表 6 列出了与 Zlob 家族关联样本数量最多的 10 个恶意代码家族，从表中看到 Rbot、Sdbot 家族分别存在 45、25 个与 Zlob 家族具有代码复用关系的样本，而 Rbot、Sdbot 的源码已经被公开。通过对样本的深入细致分析，发现 Zlog 家族与 Rbot、Sdbot 存在代码复用，但是仅从样本难以推断复用主从关系。该实验说明本文方法能快速发现不同恶意代码家族间的关联关系。

表 6　　Zlob 家族溯源实验结果

恶意代码家族	关联样本数量
Trojan.Win32.Vapsup	2 566
Trojan.Win32.Agent	124
Trojan-Downloader.Win32.Agent	114
Trojan-Clicker.Win32.Agent	69
Trojan-Ransom.Win32.Hexzone	58
Trojan-GameThief.Win32.OnLineGames	47
Backdoor.Win32.Rbot	45
Backdoor.Win32.Agent	43
Trojan.Win32.BHO	40
Backdoor.Win32.SdBot	25

5　讨论与未来工作

前期工作主要集中在代码相似判定技术的研究，仅有 Khoo[24]与 David[15]等少数几个二进制代码搜索的研究工作。与这些工作相比，本文方法初始

阶段忽略函数内部代码块的跳转关系，直接应用 simhash 检索相似代码块，进而利用倒排索引发现潜在的相似函数，最后通过跳转关系精确判定是否相似。本文方法提高了代码的溯源速度与相似判定速度，且具有较高的准确率与召回率。

然而，该方法局限有如下方面。

1) 为避免查杀，恶意代码作者会使用加壳、混淆技术处理样本，获取这类样本的正常函数代码需要使用脱壳、去混淆等技术，降低了效率。

2) 溯源结果依赖代码库，需要收集大量恶意代码样本构建尽可能全面的代码库。

实验结果表明，该方法的准确率尚有提升空间，可以进一步优化 simhash 的计算过程以保证召回率的同时进一步提高准确率。另外，将基于该方法与收集的样本，识别不同编译器、不同壳插入的函数，识别样本中的复用函数，并识别不同恶意代码家族的特殊函数，为恶意代码同源判定技术的研究奠定基础。

6 结束语

针对编译器插入代码与复用代码在恶意代码同源判定工作中造成巨大干扰，导致同源判定效率不高的问题，本文提出了基于 simhash 与倒排索引，以函数为单位的二进制代码快速溯源方法。该方法的核心思想是基于 simhash 算法搜索相似代码块，继而基于倒排索引获取潜在相似函数，最后使用精确代码相似判定方法判定 2 个函数是否相似，完成代码溯源。实验表明，该方法能在大量样本中快速溯源与某函数相似的函数代码及其所在样本，且具有较高的准确率与召回率。基于该方法可以开发代码搜索引擎等工具，帮助逆向分析人员提高效率，提升同源判定工作的自动化程度。

参考文献：

[1] 董志强, 肖新光, 张栗伟. 编码心理学分析病毒同源性[J]. 信息安全与通信保密, 2005(8):55-59.
DONG Z Q, XIAO X G, ZHANG S W. Malware homology identification based on programming psychology[J]. China Information Security, 2005(8):55-59.

[2] GReAT. Gauss: abnormal distribution 2012[R/OL]. https://securelist.com/analysis/publications/36620/gauss-abnormal- distribution/

[3] YURY Y, NAMESTNIKOV V K, OLEG K. Chthonic: a new modification of ZeuS 2014[R/OL]. https://securelist. com/blog/ virus-watch/68176/chthonic-a-new-modification-of-zeus/.

[4] SKELORU V. Visgean/Zeus[EB/OL]. Github 2011. https://github.com/Visgean/Zeus.

[5] GREAT. A fanny equation: “i am your father, stuxnet”2015[EB/OL]. https://securelist.com/blog/research/68787/a-fanny-equation-i-am-your-father-stuxnet/.

[6] QIAO Y C, YUN X, ZHANG Y. Fast reused function retrieval method based on simhash and inverted index[C]// 2016 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. 2016.

[7] BENCSATH B, PEK G, BUTTYAN L, et al Duqu: a stuxnet-like malware found in the wild[R]. CrySyS Lab Technical Report. 2011.

[8] GREAT. Cloud Atlas: RedOctober APT is back in style 2014[R/OL]. http://securelist.com/blog/research/68083/cloud-atlas-redoctober-apt-is-back-in-style/.

[9] LABS F S. PITOU: The “silent” resurrection of the notorious Srizbi kernel spambot[R]. 2014.

[10] MYLES G, COLLBERG C, *K*-gram based software birthmarks[C]// Proceedings of the 2005 ACM Symposium on Applied Computing. 2005: 314-318.

[11] SÆBJØRNSEN A, WILLCOCK J, PANAS T, et al. Detecting code clones in binary executables[C]//18th International Symposium on Software Testing and Analysis. 2009:117-128.

[12] LAKHOTIA A, PREDA M D, GIACOBAZZI R. Fast location of similar code fragments using semantic'juice'[C]//2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop. 2013: 1-6.

[13] RUTTENBERG B, MILES C, KELLOGG L, et al. Identifying shared software components to support malware forensics[J]. Detection of Intrusions and Malware, and Vulnerability Assessment: Springer, 2014, 21-40.

[14] OUELLETTE J, PFEFFER A, LAKHOTIA A. Countering malware evolution using cloud-based learning[C]// 2013 8th International Conference on Malicious and Unwanted Software, 2013.

[15] DAVID Y, YAHAV E, Tracelet-based code search in executables[C]// ACM SIGPLAN Notices. 2014.

[16] ALRABAEE S, SHIRANI P, WANG L. SIGMA: a semantic integrated graph matching approach for identifying reused functions in binary code[J]. Digital Investigation, 2015, 12: S61-S71.

[17] CHARIKAR M S. Similarity estimation techniques from rounding algorithms[C]//34th Annual ACM Symposium on Theory of Computing. 2002.

[18] MANKU G S, JAIN A, SARMA A D. Detecting near-duplicates for web crawling[C]//16th International Conference on World Wide Web. Banff, Alberta, Canada, 2007:141-50.

[19] UDDIN M S, ROY C K, SCHNEIDER K A, et al. On the effectiveness of simhash for detecting near-miss clones in large scale software systems[C]// 2011 18th Working Conference on Reverse Engineering

(WCRE), 2011.

[20] 郭颖, 陈峰宏, 周明辉. 大规模代码克隆的检测方法[J]. 计算机科学与探索, 2014(4):417-426.
GUO Y, CHEN F H, ZHOU M H. Code clone detection method for large scale source code[J]. Journal of Frontiers of Computer Science & Technology, 2014(4):417-426.

[21] TIMO J. RINNE S L. ssh-3.2.9.1 2003[EB/OL]. http://down1.chinaunix.net/distfiles/ssh-3.2.9.1.tar.gz.

[22] VX Heaven[EB/OL]. http://vxheaven.org/.

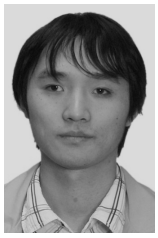
[23] Wikipedia. Agobot 2016[EB/OL]. https://en. Wikipedia. org/wiki/Agobot.

[24] KHOO W M, MYCROFT A, ANDERSON R. Rendezvous: a search engine for binary code[C]// Proceedings of the 10th Working Conference on Mining Software Repositories. 2013.

作者简介：

乔延臣（1988-），男，山东聊城人，中国科学院博士生，主要研究方向为网络信息安全、恶意代码等。

云晓春（1971-），男，黑龙江哈尔滨人，博士，中国科学院研究员、博士生导师，主要研究方向为信息安全、计算机网络等。



庹宇鹏（1984-），男，河北廊坊人，中国科学院信息工程研究所助理研究员，主要研究方向为网络异常检测、移动互联网大数据挖掘。

张永铮（1978-），男，黑龙江哈尔滨人，博士，中国科学院研究员、博士生导师，主要研究方向为网络安全态势感知。