

二进制代码相似度分析及在嵌入式设备固件漏洞搜索中的应用^{*}

于颖超¹, 甘水滔¹, 邱俊洋¹, 秦晓军¹, 陈左宁²

¹(数学工程与先进计算国家重点实验室, 江苏 无锡 214125)

²(中国工程院, 北京 100088)

通信作者: 甘水滔, E-mail: ganshuitao@gmail.com



摘 要: 在当今“万物互联”的时代, 嵌入式系统逐渐成为接入云端的重要组成部分, 常用于安全和隐私敏感的应用或设备中. 然而, 其底层软件(即固件)也在频繁遭受着安全漏洞的影响. 由于嵌入式设备底层硬件平台的复杂异构, 软硬件实现差异较大, 且其专用性强、源码/文档等往往不会公开, 加之其运行环境受限等原因, 使得一些在桌面系统上运行良好的动态测试工具很难(或根本不可能)直接适配到嵌入式设备/固件环境中. 近年来, 研究人员逐渐开始探索基于二进制相似度分析技术来检测嵌入式设备固件中存在的已知漏洞, 并且取得了较大的进展. 围绕二进制代码相似度分析面临的关键技术挑战, 系统研究了现有的二进制代码相似度分析技术, 对其通用流程、技术特征、评估标准进行了综合分析和比较; 然后分析并总结了现有二进制代码相似度分析技术在嵌入式设备固件漏洞搜索领域的应用; 最后, 提出了该领域应用仍然存在的一些技术挑战及未来的一些开放性的研究方向.

关键词: 二进制代码相似度分析; 嵌入式固件; 漏洞搜索; 深度学习

中图法分类号: TP311

中文引用格式: 于颖超, 甘水滔, 邱俊洋, 秦晓军, 陈左宁. 二进制代码相似度分析及在嵌入式设备固件漏洞搜索中的应用. 软件学报, 2022, 33(11): 4137–4172. <http://www.jos.org.cn/1000-9825/6540.htm>

英文引用格式: Yu YC, Gan ST, Qiu JY, Qin XJ, Chen ZN. Binary Code Similarity Analysis and Its Applications on Embedded Device Firmware Vulnerability Search. Ruan Jian Xue Bao/Journal of Software, 2022, 33(11): 4137–4172 (in Chinese). <http://www.jos.org.cn/1000-9825/6540.htm>

Binary Code Similarity Analysis and Its Applications on Embedded Device Firmware Vulnerability Search

YU Ying-Chao¹, GAN Shui-Tao¹, QIU Jun-Yang¹, QIN Xiao-Jun¹, CHEN Zuo-Ning²

¹(State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214125, China)

²(Chinese Academy of Engineering, Beijing 100088, China)

Abstract: In the era of today's Internet of Things, embedded systems are becoming important components for accessing the cloud, which are used in both secure and privacy-sensitive applications or devices frequently. However, the underlying software (a.k.a. firmware) often suffered from a wide range of security vulnerabilities. The complexity and heterogeneous of the underlying hardware platform, the difference of the hardware and software implementation, the specificity and limited document, together with limited running environment made some of very good dynamic testing tools for desktop systems hard to (even impossible) be adapted to embedded devices/firmware environment directly. In recent years, researchers have made great progress in detecting well-known vulnerabilities in embedded device firmware based on binary code similarity analysis. Focusing on the key technical challenges of binary code similarity analysis, the existing binary code similarity analysis technologies are studied systematically; the general process, technical characteristics, and evaluation criteria of these technologies are analyzed and compared comprehensively. Then, the application of these technologies is analyzed and summarized in the field of embedded device firmware vulnerability search. At last, some technical challenges in this field are presented and some open future research directions are proposed for the related researchers.

Key words: binary code similarity analysis; embedded firmware; vulnerability search; deep learning

* 收稿时间: 2021-03-29; 修改时间: 2021-06-09, 2021-08-21; 采用时间: 2021-11-16; jos 在线出版时间: 2021-12-24

近年来,开源社区的规模迅速扩大,提升了软件开发行业的能力,缩短了软件开发周期,与此同时,也加速了漏洞的传播能力.尤其是在嵌入式设备领域,出于成本考虑,设备制造商通常会复用大量的第三方组件(开源社区、历史代码库,第三方公司等),将成熟的 x86 终端设备上的软件程序平滑移植到 ARM 或 MIPS 等平台上,这些都将导致同一厂商不同型号的产品甚至是不同厂商不同型号的产品都有可能受到同一个复用组件漏洞/已经固件漏洞的影响,而作为产品使用人员甚至是安全分析人员(非厂商)很难对设备固件的内部关联有一个很清晰的认识,无疑也加剧了潜在漏洞的传播.

Cui 等人^[1]在报告中指出很多固件更新在第三方库内包含了已知漏洞若干年,而且他们还进一步表明,80.4% 的厂商发布了带有已知漏洞的固件.当考虑嵌入式系统时,可能会产生非常严重的破坏性的影响,因为它们可以控制安全关键系统,因此,攻击这样的设备可能会导致大规模的公共系统故障,在全国乃至全世界范围内造成严重的安全后果.臭名昭著的 Mirai botnet 就是利用某款智能摄像头的一个已知漏洞,成功入侵大量的连接在互联网上的嵌入式设备,实现对目标网络 DNS (domain name system) 服务器的 DDoS (distributed deny of service) 攻击,进而导致全球数十万个网络被拒绝服务^[2].

由于嵌入式设备底层硬件平台的种类繁多、厂商众多、软硬件实现差异巨大,且其专用性强、源码/文档等往往不会公开,加之其运行环境受限等原因,使得一些在桌面系统上运行良好的动态测试工具很难(或根本不可能)直接适配到嵌入式设备/固件环境中.在这种形势下,研究人员逐渐开始探索如何基于二进制相似度分析技术来检测嵌入式设备固件中存在的已知漏洞,并且取得了较大的进展.然而,相比于普通桌面应用程序的二进制相似度分析,将二进制相似度分析技术应用于嵌入式设备固件的漏洞搜索,面临着巨大的挑战:(1) 固件源代码通常是无法获取的;(2) 固件的指令集架构多种多样,所使用的编译器和编译优化级别也不统一,相同的源代码,使用不同的编译参数,甚至是不同的编译器,针对不同的目标平台可以编译生成不同的二进制文件,虽然其实现的功能是一样的,但其语法和结构特征各异.针对这些挑战,近年来,跨平台/跨体系结构的二进制固件漏洞搜索受到了越来越多的关注,先后提出了若干表现良好的工作^[3-6].

本文关注到,虽然文献[7]对二进制相似度分析进行了详细的综述,但其立足点在于二进制分析技术的分类,并没有将对二进制分析应用于漏洞检测,特别是嵌入式设备固件中的漏洞检测的效果进行调查,而且其关注的论文大多在 2019 年之前.因此,本文收集了 2015–2020 年间提出的具有代表性的二进制相似度分析方法作为研究基础,对二进制相似度分析的技术特征进行了详细的分析和评估,详细对比了其在嵌入式设备固件漏洞搜索中的表现,最终提出该领域的一些重要开放研究问题.这些工作主要发表在网络安全和软件工程领域的主要会议和期刊,共 34 篇,包括 CCS, NDSS, RAID, ASIACCS, DIMVA, FSE, ASE, ICSE, TSE 以及一些公开可访问的期刊(如 Access 等).

本文首先分析了基于二进制相似度分析的固件漏洞搜索的必要性(第 1.1 节)及基本思路(第 1.2 节),列出了其关键技术挑战(第 2 节)及现有工作分类(第 3 节).然后,围绕统一的“四阶段”分析框架(第 4.1 节),详细阐述了各阶段的技术特征(第 4.2–4.5 节)及各工具所使用的评估标准(第 4.6 节),并对其进行了一个横向的综合比较(第 4.7 节).第 5 节对二进制代码相似度分析技术在嵌入式设备固件漏洞搜索中的应用进行了综合分析和评估,比较了相关工作所使用的数据集、用于搜索的 CVE 以及搜索精度和搜索开销.最后,对现有技术进行了总结(第 6.1 节)、提出了将二进制代码相似度分析技术应用在嵌入式设备固件漏洞搜索中面临的进一步挑战(第 6.2 节),并提出了一些未来的研究方向(第 6.3 节).

1 背景

1.1 基于二进制相似度分析的嵌入式设备固件漏洞搜索的必要性

与桌面二进制分析相比,嵌入式设备固件安全分析更具挑战性^[8].首先,嵌入式设备固件逆向本身就是要克服的一项特殊挑战,涉及固件获取、固件解包和提取以及固件和二进制识别等^[9,10];其次,嵌入式设备固件是为特定的硬件量身定制的,因此,在没有标准和存在各种 CPU 架构的情况下,固件分析变得更具挑战性;第三,嵌入式设备固件通常是特定任务定制的且是资源受限的.因此,嵌入式系统在运行时存在特定的效率和可伸缩性问题,与桌面环境相比,使用多处理机制或虚拟化技术来加速进行并行测试在嵌入式系统中是不切实际的;最后,在嵌入式

系统中执行故障检测是有局限性的, 由于输入输出能力、计算能力和成本的限制, 内存破坏攻击在嵌入式系统中并不明显. 往往只能捕获取会造成设备崩溃的一些特定的攻击 (比如, 格式化字符串或基于堆栈的溢出等).

针对嵌入式设备固件的漏洞挖掘问题, 学术界和工业界不断尝试通过基于仿真^[8,11-13]/半仿真^[14-20]/固件托管^[21-26]等方法来突破传统漏洞挖掘方法无法平滑移植到嵌入式设备固件, 以突破高效且有效地进行嵌入式设备固件动态测试的技术瓶颈, 并且取得了很好的进展. 然而, 由于嵌入式设备底层硬件平台的种类较多, 软硬件实现差异较大, 且其专用性强, 源码/文档等往往不公开等原因, 使得嵌入式设备固件的动态测试大多仅针对特定的固件种类或体系结构进行, 很难有效地扩展到更多的设备种类和体系架构. 此外, 研究指出, 对设备固件的漏洞攻击中除了少部分是以前没有发现过的漏洞之外, 绝大部分是已经发布过的漏洞. 已知漏洞之所以会影响到嵌入式设备的安全, 是因为大多数的嵌入式设备都存在着研发-生产分离模式: 同一份源代码 (可能存在漏洞) 编译、部署在不同的嵌入式平台 (ARM/MIPS/PPC 等) 上 (也即, 代码重用), 会导致同一厂商不同型号甚至不同厂商不同型号的产品都有可能受到同一已知固件漏洞的影响, 而由于研产分离, 分析人员很难对嵌入式设备固件的内部关联有一个很清晰的认识.

近年来, 随着人工智能技术的发展, 学术界和工业界逐渐尝试探索将应用良好的一些机器学习算法应用到软件的漏洞挖掘/发现^[27-30]上, 并且取得了较大的进展, 其基本思路是利用机器学习/深度学习对软件代码中蕴含的规则进行分析和泛化, 从海量数据中发现和学习规则, 获取最多的与漏洞有关的信息, 以有效地从分析代码中提取并定位漏洞点. 在这种形势下, 研究人员逐渐开始探索将一些应用良好的软件脆弱性检测方法应用到基于二进制相似度的漏洞检测上, 进而扩展到基于二进制相似度的嵌入式设备固件漏洞搜索领域, 取得了较大的进展. 然而, 相比于普通桌面应用程序的二进制相似度分析, 将二进制相似度分析技术应用于嵌入式设备固件的漏洞搜索, 面临着巨大的挑战: (1) 固件源代码通常是无法获取的; (2) 固件的指令集架构多种多样, 所使用的编译器和编译优化级别也不统一, 相同的源代码, 使用不同的编译参数, 甚至是不同的编译器, 针对不同的目标平台可以编译生成不同的二进制文件, 虽然其实现的功能是一样的, 但其语法和结构特征各异. 针对这些挑战, 近年来, 跨平台/跨体系结构的二进制固件漏洞搜索受到了越来越多的关注.

1.2 基于二进制相似度分析的固件漏洞搜索

图 1 给出了嵌入式设备固件漏洞搜索的示意图. 这里以函数为粒度, 给定一个感兴趣的二进制脆弱函数 (受 CVE 影响, 比如包含了 Heartbleed 漏洞^[31]的二进制函数), 本文希望分析一个很大的二进制函数语料库 (获取固件镜像 (固件镜像可以从不同的嵌入式设备中提取、可以从第三方提供的下载链接下载、也可以从设备厂商提供的官方网站或 FTP 站点获取), 解析出其可执行的二进制文件集合, 提取出其中的函数, 形成一个大的二进制函数语料库), 快速且精确地识别出语义上等价或类似于感兴趣的二进制脆弱函数的候选列表, 即为疑似 CVE 的漏洞函数, 然后再通过人工/静态/动态分析的手段进一步确定识别出来的可疑函数是否为真实的漏洞. 这里, 将感兴趣的二进制脆弱函数称为查询函数, 将二进制函数语料库称为目标语料库.

2 关键技术挑战

与源代码的相似度分析不同, 二进制代码相似度分析的唯一输入来源只可能是可执行的二进制程序. 源代码通过编译、链接和优化等步骤编译成可执行的二进制程序, 然而, 任何一个步骤上的些微区别都有可能会导致程序的详细执行行为严重区别于其源代码.

文献 [7] 详细描述了二进制代码的编译流程: 以源代码为输入, 使用选择的编译器 (有 gcc、ICC、clang、VC 等) 和优化设置 (O0、O1、O2、O3、Ox 等), 编译到特定的平台 (操作系统: macOS、Linux、Windows 等可选, CPU 架构: x86、x64、ARM、MIPS 等, 字长: 32 bit、64 bit), 生成目标文件, 再由链接器将这些文件链接到二进制程序中生成最终的可执行文件. 由于编译器、优化选项、CPU 架构、操作系统的不同, 使得相同的源代码经过编译后得到的二进制代码也非常不同, 加之在编译过程中, 混淆技术 (的加入也会对编译结果产生显著的影响, 使得跨越这些差异进行二进制相似度分析成为了一个巨大的挑战. 另一方面, 跨版本的二进制可能会同时改变程序的语法和语义, 也是二进制相似度分析的另一个重要挑战.

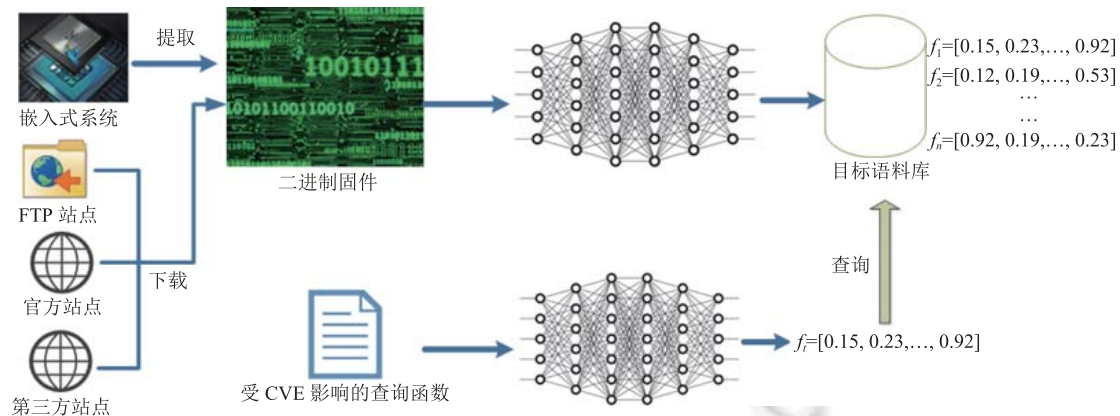


图1 嵌入式设备固件漏洞搜索示意图

以图2中的源代码函数 `ssl_get_algorithm2()` 为例, 该段代码摘自 `openssl-1.0.1a` 中的文件 `ssl/s3_lib.c`. 图2(a)是其源代码, 其中第4行存在一个漏洞, 其漏洞 CVE 编号为: CVE-2013-6449, 这是一个 DoS 漏洞, 存在于 1.0.2 版本之前的 OpenSSL 中. 该函数会从一个错误的数据结构中获取某个特定的版本号, 进而使得远程攻击者通过 TLS 1.2 客户端精心构造的流量触发后台程序发生崩溃, 进而产生 DoS 攻击. 图2(b)、图2(c)、图2(d)分别是使用 IDA Pro (<https://www.hex-rays.com/products/ida/>) 对其二进制版本 (使用 `gcc -O3` 分别针对 x86_64、ARM、MIPS 体系进行编译生成的) 进行反汇编之后的结果.

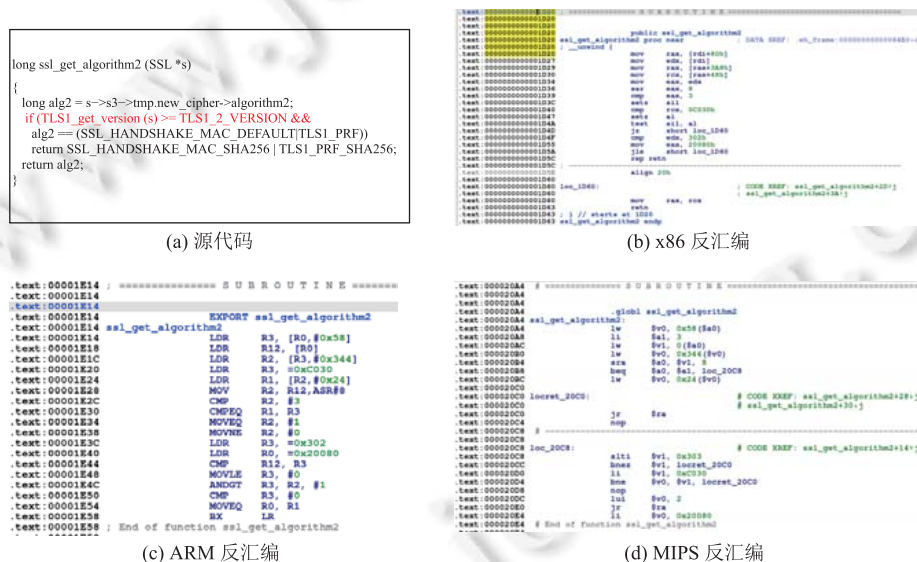


图2 源代码与不同体系架构下的反汇编代码的区别

挑战 1 (C1): 信息损失. 显然, 在编译过程中, 一些在源代码中可用的信息, 比如函数名、变量名等都会丢失. 因此, 分析二进制代码将会比分析源代码更具挑战性和复杂性. 此外, 在缺少调试信息 (比如标识符名称) 的去符号的二进制情况下, 二进制分析任务更具挑战性.

挑战 2 (C2): 指令集架构影响. 显然, x86、ARM 和 MIPS 等架构的指令集是截然不同的. 除此之外, 调用约定、通用和特殊目的 CPU 寄存器 (参数传递或数据存储) 以及内存访问策略也会因体系结构不同而不同. 因此, 即使二进制文件来自于相同的源代码, 不同体系架构下的二进制代码表示也是非常不同的, 因此是难以比较的.

图 3 进一步使用了 4 个部分阐述了同一源代码针对不同 ISA、不同编译器、不同优化选项以及在使用混淆的情况下产生的二进制代码的控制流图的区别。

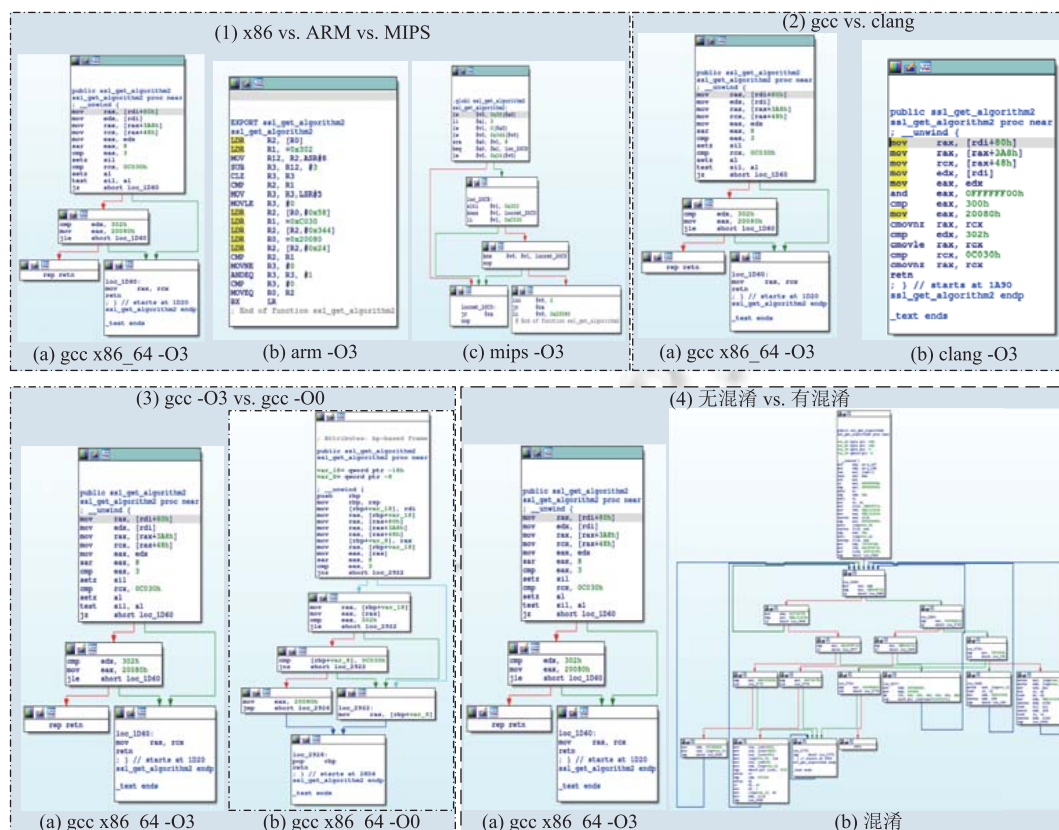


图 3 跨体系、跨编译器、跨优化级别、使用混淆技术的二进制的区别

图 3(1) 中是分别针对 x86_64、ARM 和 MIPS 平台, 使用 gcc 的 -O3 选项编译生成的函数的控制流图. 图 3(2) 显示的是 gcc 和 clang 两种编译器在 x86_64 平台上的编译结果的区别. 图 3(3) 显示的是同一编译器 (gcc), 同一目标平台 (x86_64), 但不同优化选项 (-O3 和 -O0) 的编译结果区别. 图 3(4) 显示的是使用混淆技术和不使用混淆技术时的编译结果区别. 可以看出:

挑战 3 (C3): 编译器影响. 不同编译配置 (编译器、优化选项、目标平台) 的控制流图非常不一致. 以图 3(1) 为例, 尽管其实现的代码语义是相同的, 但同一段源代码编译生成的二进制函数控制流图中 x86_64 包含了 4 个基本块, MIPS 则包含了 5 个, 而 ARM 则仅包含了 1 个.

挑战 4 (C4): 函数内联影响. 作为优化技术的一种, 内联技术可以将一个小的函数嵌入到其调用方函数中. 由于内联函数和函数的其他部分之间没有明显的区分, 因此使得函数内联识别任务非常具有挑战性. 当内联函数的汇编指令由于指令对齐和流水线而不连续时, 这项任务就变得更具挑战性.

挑战 5 (C5): 混淆技术影响. 混淆技术会大大改变原有的控制流图. 如图 3(4) 所示, 在使用了混淆技术 (本文为说明情况, 使用了 -fla, -sub, -bcf 混淆) 之后, 其控制流图发生了显著的变化. 常规的二进制相似度匹配技术是极难将这两个控制流图识别为相似的.

挑战 6 (C6): 脆弱代码逻辑通常分散在多个基本块中, 甚至跨多个函数. 此示例中的漏洞是由该函数的参数的错误版本检查造成的, 从图中可以看出这个代码逻辑跨越了多个基本块, 并且是与此函数外的其他代码逻辑混合在一起的.

图 4 给出了图 2(a) 中的源代码及其补丁代码 (分别是图 4(a) 和图 4(b)), 以及它们经过 gcc -O3 编译选项, 针对 x86_64 平台生成的源代码程序和补丁代码程序的控制流图 (分别是图 4(a1) 和图 4(b1)) 的区别, 可以看出:

```
long ssl_get_algorithm2 (SSL *s)
{
    long alg2 = s->s3->tmp.new_cipher->algorithm2;
    if (TLS1_get_version(s) >= TLS1_2_VERSION &&
        alg2 == (SSL_HANDSHAKE_MAC_DEFAULT|TLS1_PRF))
        return SSL_HANDSHAKE_MAC_SHA256 | TLS1_PRF_SHA256;
    return alg2;
}
```

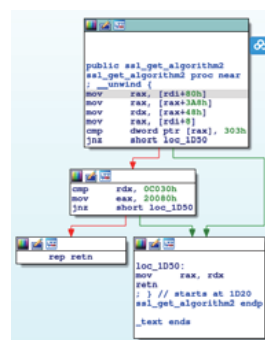
(a) 源代码



(a1) 源代码程序控制流图

```
long ssl_get_algorithm2 (SSL *s)
{
    long alg2 = s->s3->tmp.new_cipher->algorithm2;
    if (s->method->version == TLS1_2_VERSION &&
        alg2 == (SSL_HANDSHAKE_MAC_DEFAULT|TLS1_PRF))
        return SSL_HANDSHAKE_MAC_SHA256 | TLS1_PRF_SHA256;
    return alg2;
}
```

(b) 补丁代码



(b1) 补丁代码控制流图

图 4 源代码程序和补丁代码程序控制流图区别

挑战 7 (C7): 补丁影响. 补丁代码和脆弱代码的控制流图可能不会出现明显的差别. 如图 4(a) 和图 4(b) 所示, 源代码和漏洞代码的差异仅有一行, 由此产生的二进制函数控制流图的差异也非常细微, 常规的二进制相似度分析方法通常会将其识别为相似的, 然而, 显而易见, 这并不正确.

挑战 8 (C8): 精确度. 静态分析方法通常会直接分析目标二进制代码而不会直接执行它, 因此, 可以分析整个二进制代码并且可以探测到所有潜在的执行路径. 然而, 由于其不会实际执行代码, 因此容易出现高误报率 (假的漏洞) 或者高漏报率 (不能发现全部漏洞) 的情况.

挑战 9 (C9): 效率. 如第 1.2 节所述, 二进制固件漏洞搜索不仅要考虑两个二进制代码片段的相似度分值比较, 而且还要高效地从目标语料库中查询出与查询代码相同或相似的代码, 由于目标语料库的规模较大, 因此其查询效率也是一个很大的挑战.

挑战 10 (C10): 可伸缩性. 随着桌面应用程序、IoT 设备、嵌入式系统以及它们之间的互联互通的急剧增长, 部署的软件数量正在呈指数级的增长, 因此, 大规模的漏洞检测是绝对有必要的, 特别是应对嵌入式设备固件时, 所提方法的可伸缩性将是一个巨大的挑战.

3 二进制相似度分析现有工作分类

本文依据提取二进制代码特征的方式将现有的二进制代码相似度分析工作划分为以下几类: 基于静态分析^[3-6,32-37]、基于动态分析^[38-42]、基于混合分析^[39,43-46]和基于机器学习的分析^[46-60], 如图 5 所示.

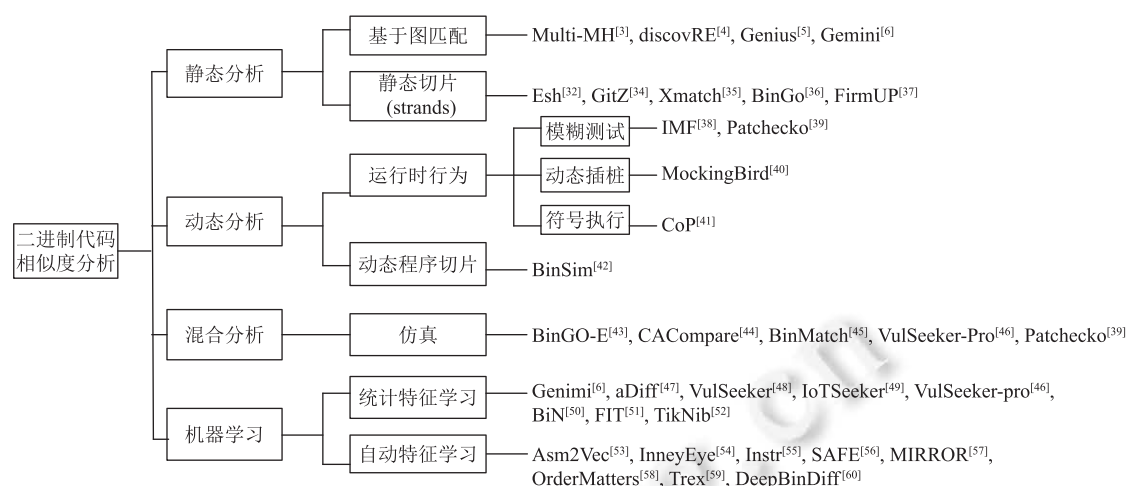


图5 二进制相似度分析技术分类

静态分析方法的一种常用方法是將二进制代码转化为图, 然后再执行比较^[3-5]. 早期的 (2015 年之前) Bindiff 在调用图上执行多对多的图同构检测以匹配函数, 并且利用 CFG 匹配基本块. 在此基础上, Binslayer^[61]使用 Hungarian 算法^[62]增强图匹配以提升匹配的结果. 另一种常用方法是对图进行分解, 生成更小的、可比较的组件, 进行比较^[32-37]. 早期的 Tracelet^[63]將 CFGs 转换为一组固定长度的路径, 称为 tracelets, 然后通过重写技术匹配这些 tracelets.

动态分析方法通过其执行行为来评估一个给定的软件应用程序, 这可以通过在一个真实或仿真设备上通过一组特定输入组成的测试用例来执行/模块执行这个程序以监控其行为, 获取其输入输出信息或程序的行为信息等来实现. 早期的 Blanket Execution^[64]使用相同的输入执行两个输入二进制的函数, 并且比较监控到的行为以计算相似度. BinHunt^[65]和 iBinHunt^[66]使用符号执行和定理证明来验证两个基本块或 strands 是否相等. IMF^[38]和 Patcheko^[39]使用模糊测试技术, MockingBird^[40]基于动态插桩, CoP^[41]基于符号执行来收集行为轨迹, BinSim^[42]依赖于系统调用来执行动态切片, 并且使用符号执行来检查等价性.

混合分析方法將动态分析和静态分析相结合, 同时兼顾代码覆盖率、检测准确率以及方法本身的可伸缩性. BinGo-E^[43]、CACompare^[44]、BinMatch^[45]、VulSeeker-Pro^[46]均使用了仿真的方法. 在静态分析得出的结果之后, 通过仿真模拟执行目标函数以提取各自的语义特征进行相似度比对. Patcheko^[39]则是在第 1 阶段静态检测得出的候选函数的基础上, 执行动态分析, 利用运行时 DLL 二进制注入和远程调试解决方案, 捕获各自的执行轨迹, 以度量两个函数之间的相似性, 进一步确认静态分析报告的正确性.

基于学习的方法. 虽然先前有些二进制相似度分析工作也使用了机器学习算法, 比如 Zeek^[34], 但往往仅为最后的分类器或中间的过滤器使用, 不占主导地位. 自 Gemini^[6]之后, 基于图嵌入、自然语言处理、自注意力网络等深度学习算法才逐渐被广泛应用于二进制相似度分析领域中. 根据基于学习的方法提取特征的方式不同, 可以将其划分为基于统计特征的学习^[6,47-52]和基于自动化学习特征的学习^[53-60]. 基于统计特征的学习主要学习图中节点 (也即, 基本块) 中的一些统计信息, 作为节点的属性, 嵌入到图中, 组成 ACFG, 喂给图神经网络进行学习, 学习出图嵌入向量, 作为函数的向量表示. Gemini^[6]、VulSeeker-Pro^[46]、aDiff^[47]、VulSeeker^[48]、IoTSeeker^[49]、BiN^[50]和 FIT^[51]都对 ACFG 及其变种使用类似 structure2vec^[67,68]和 Siamese 网络^[69]进行有监督的学习. 近年来, 为了消除人工选择特征可能会引入的偏差, 研究人员开始利用自动化的特征学习方法从目标代码自动学习其语义特征, 比如 Asm2Vec^[53]、InnerEye^[54]、Instr^[55]、SAFE^[56]、MIRROR^[57]、OrderMatters^[58]、Trex^[59]、DeepBinDiff^[60]等.

静态分析无需实际执行, 能够覆盖所有的代码, 但其往往只能依赖语法或结构特征, 而无法获取强大的语义信息的支撑, 因此, 检测准确率相对较低. 动态分析及混合分析通过实际执行/模拟执行二进制代码, 获取输入输出信息或者程序的运行时行为信息, 能够获取比静态分析更精确的语义信息来代表二进制代码, 因此, 能发现静态分析

不能发现的漏洞. 然而, 其开销大, 可伸缩性差, 往往需要足够的输入才能保证其代码覆盖率, 而且它们往往只分析执行过的代码, 因此不适用于大规模的嵌入式设备固件漏洞分析. 相比于传统的静态和动态方法, 基于学习的方法: 1) 具有更高的准确度. 它们在分析中充分融合了代码的特征 (语法特征、语义特征或结构特征等). 2) 具有更好的可扩展性. 避免了繁重的图匹配算法或动态执行, 更重要的是, 学习过程可以通过 GPUs 显著加速, 因此吸引了业界的广泛关注, 也是近年来这方面的研究的一个主要发展趋势.

4 二进制代码相似度分析技术特征分析

4.1 统一的“四阶段” workflow 分析框架

为了充分说明二进制相似度分析的相关技术特征, 本文使用一个统一的“四阶段”分析框架来分析、总结本文所涉及的二进制相似度分析工作, 如图 6 所示, 这个“四阶段”分析框架由 4 个阶段组成.

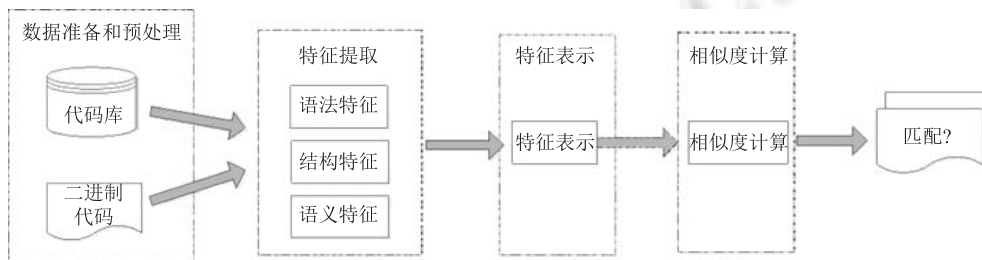


图 6 统一的“四阶段” workflow 分析框架

阶段 1: 数据准备和预处理阶段. 此阶段用于收集大量的用于训练/评估/比对的二进制代码相关数据 (往往是二进制文件), 然后根据要比对的粒度 (程序、函数、基本块) 对数据进行切分. 与源代码的数据切分不同, 二进制代码的数据切分需要分析人员使用相应的工具 (IDA Pro、BAP (binary analysis framework)^[70]、Angr^[71]、Valgrind^[72]等) 对其进行反汇编或动态插桩才能获取.

阶段 2: 特征提取阶段. 此阶段对阶段 1 切分出来的二进制代码片段 (可以是程序、函数或基本块), 提取出其语法/结构/语义信息, 作为反映这个二进制代码片段的特征 (也即唯一表示), 分别称之为语法特征、结构特征、语义特征.

阶段 3: 特征表示阶段. 此阶段会对阶段 2 提取出来的特征进行“向量化”表示, 以方便阶段 4 进行相似度计算. 特征表示阶段的输出结果才是最终计算相似度分值的输入. 可以将其分为使用嵌入的特征表示, 以及不使用嵌入的特征表示. 本文将在第 4.4 节中详细描述这两种类型的特征表示方法.

阶段 4: 相似度计算阶段. 相似度计算阶段所使用的比较方法会根据所使用的特征表示的不同而有所不同, 比如, 分析人员可以计算特征集合之间的 Jaccard 距离^[73]、计算 CFGs 之间的图编辑距离^[74]或者甚至结合深度学习算法进行聚类^[34]. 然而, 任何算法的成功很大程度上都依赖于所选择的特征.

接下来, 本节将从这个“四阶段”分析框架出发, 对每一个阶段的技术特征进行详细分析.

4.2 阶段 1: 数据准备和预处理

4.2.1 阶段 1.1: 数据准备

数据准备阶段的目的是准备后续进行处理、学习、训练及评估的数据, 根据数据来源不同, 可以将其分为 3 类.

I 型数据集: 基线数据集. 由于目标设备固件的闭源属性, 要逆向目标设备固件中的二进制程序所使用的编译器、优化选项以及混淆技术等是极其困难的. 因此, 分析人员通过构造一些数据集以进行分析和评估. 如图 7 的上半部分虚线框内的流程所示, 在数据准备阶段, 需要收集一些具有代表性的源代码 (固件中常用的应用程序或库), 使用不同的编译器以及不同的优化选项, 设置不同的操作系统、CPU 体系, 针对不同的目标平台进行编译, 生成二进制文件, 作为基线数据集.

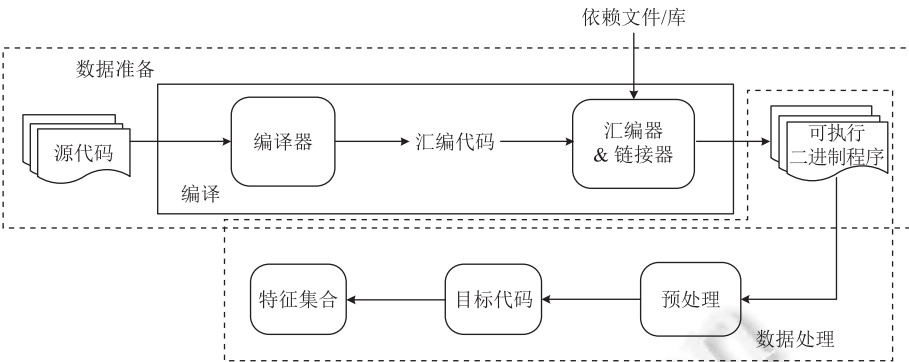


图 7 数据准备和预处理

II 型数据集: 固件镜像数据集. 固件镜像数据集的来源有以下途径: (1) 从真实的物理设备中获取; (2) 使用公开可用的固件镜像数据集^[11]; (3) 从设备厂商/第三方提供的 Web 站点或 FTP 站点获取. 然后, 使用 Binwalk (<https://code.google.com/p/binwalk/>) 对固件镜像进行解包, 从中提取出二进制文件, 然后经过数据预处理、特征提取和特征表示, 形成相应的特征向量, 组成目标语料库 (也称目标搜索数据集).

III 型数据集: 漏洞数据集. 漏洞数据集是漏洞编号 (通常是 CVE) 和实际引入这个漏洞的对应函数之间的一个映射. 漏洞数据集是对目标搜索数据集进行搜索的查询函数集合. 其构建方式如下: 首先, 收集对应 CVE 编号的漏洞列表, 然后, 下载存在漏洞的软件源代码, 对其进行编译, 并使用符号名从二进制代码中提取出脆弱函数, 存储在漏洞数据集中. 后续会与基线数据集一样, 经过特征提取和特征表示, 形成相应的特征向量.

4.2.2 阶段 1.2: 数据预处理

数据预处理阶段的主要目的是根据要匹配的粒度从可执行二进制程序中提取出待匹配的函数/基本块/指令, 其处理方式可以分为以下几种.

(1) 转化成中间表示 (IR)

大多数处理二进制可执行文件的工具使用的第一步就是将可执行文件程序提升到某种 IR 以缩小不同指令集架构的二进制可执行文件的语法表示的差距, 使得分析人员可以关注二进制指令中表达的语义, 而不必关注编译器发出二进制指令的方式或链接器的排列方式. 表 1 给出了现有的二进制相似度分析工具中用来将二进制代码转换成 IR 的分析平台以及其 IR 表示形式.

表 1 二进制相似度分析方法所使用的分析平台及 IR 表示

方法	分析平台	IR 表示
Multi-MH ^[3]	Valgrind ^[72]	VEX-IR
GitZ ^[33]	Valgrind, 自定义的转换器	VEX-IR, LLVM-IR
FirmUp ^[37]	Valgrind	VEX-IR
CACompare ^[44]	Valgrind	VEX-IR
BinMatch ^[45]	Valgrind	VEX-IR
VulSeeker-pro ^[46]	Valgrind	VEX-IR
BinGo ^[36]	—	REIL
Esh ^[32]	BAP ^[70] , SMACK ^[75]	LLVM-IR, Boogie IVL
CoP ^[41]	BAP	BIL
ImOpt ^[76]	BAP	BIL
Xmatch ^[35]	McSema ^[77]	LLVM-IR
BinSim ^[42]	TEMU ^[78]	VineIL

如表 1 所示, Multi-MH^[3], GitZ^[33], FrimUp^[37], CACompare^[44], BinMatch^[45]和 VulSeeker-pro^[46]均使用了 Valgrind 作为分析平台, 将汇编指令提升为 VEX-IR 形式的中间表示. Multi-MH^[3]使用了与 VEX-IR 绑定的 API 的 Python 项目 PyVex (<https://github.com/angr/pyvex>) 实现自动化静态二进制代码的转译. GitZ^[33]在将二进制代码转换成 VEX-IR 之外, 又重新实现了一个自定义的转换器, 将 VEX-IR 转换为 LLVM-IR. FrimUp^[37]首先对二进制过程进行基本块级别的切片, 获得 strands, 然后使用 VEX-IR 将 strands 提升为 IR 表示. CACompare^[44]在将二进制代码转换为 VEX-IR 之后, 将翻译得到的 IR 语句存储在自定义的数据结构中以备后续语义特征的提取. BinMatch^[45]会在转换后的 IR 上插桩监控代码, 再将插桩后的 IR 转换成二进制代码之后执行. VulSeeker-pro^[46]则是在 Vulseeker 的图嵌入算法筛选出候选函数之后, 通过将候选函数提到 IR 并进行仿真, 提取函数签名, 进行进一步的筛选, 来提升漏洞搜索的精确度. BinGo^[36]将汇编指令提升为中间表示 REIL. 在其扩展版本 BinGo-E^[43]中, 由于不再需要基于 REIL 的约束求解, 因此不再使用中间语言. Esh^[32], CoP^[41], ImOpt^[76]使用了 BAP 作为二进制转换平台将二进制代码提升为 LLVM IR 表示. 不同之处在于, Esh^[33]会在将代码提升为 LLVM IR 之后, 再使用 SMACK (bounded software verifier)^[75]转换器将 LLVM IR 转换为 Boogie IVL. ImOpt^[76]会在提升后的 IR 基础上利用即时 SSA 转换算法^[79], 规范化并消除垃圾代码, 以减轻优化和混淆带来的干扰, 进一步提升二进制相似度评估的精确度. CoP^[41]则会在提升后的 IR 基础上, 直接构建 CFGs 和 CGs, 以生成 ICFG. Xmatch^[35]扩展了 McSema^[77], 首先将目标代码转换为 LLVM IR, 然后再从转换后的函数级 IR 中提取出条件公式, 进行进一步的条件公式匹配, 进而是函数级的匹配. BinSim^[42]在获取 TEMU^[78]输出的原始 trace 数据之后, 会首先将 x86 指令提升为 Vine IL, Vine IL 的 SSA 可以在执行后向切片时辅助追踪 use-def 链, 然后从 Vine IL 上提取出系统调用序列, 通过系统调用对齐算法定位匹配的系统调用对.

IR 的引入统一了二进制代码的表示, 为函数语义特征提取带来了极大的便利, 比如所有的内存读和内存写都统一用 VEX-IR 的操作码 load 和 store 表示, 分析人员只需专注于 IR 某些特定的操作即可, 而无需在不同指令架构集的多种表达上耗费过多精力. 因此, 极大促进了跨体系架构二进制相似度分析工作.

(2) 转化成图表示

图是二进制代码结构特征的直观表示, 其在形式上可以表示为控制流图 (control flow graph, CFG)、数据流图 (data flow graph, DFG) 以及函数调用图 (function call graph, FCG) 等. 表 2 列出了使用 IDA Pro 从二进制代码中提取出图 (CFG, DFG, FCG) 并且转化为相应的图表示 (ACFG (attributed control flow graph), LSFG (labeled semantic flow graph), ICFG (inter-procedural control flow graph) 以及 3LACFG (3-level-features attributed control flow graph)) 的相关工作.

表 2 二进制相似度分析方法使用的图表示类型

方法	图类型			表示
	CFG	DFG	FCG	
discovRE ^[4]	√	—	—	—
Genius ^[5]	√	—	—	ACFG
Gemini ^[6]	√	—	—	ACFG
Patchcko ^[39]	√	—	—	ACFG
VulSeeker-pro ^[46]	√	—	—	ACFG
α Diff ^[47]	—	—	√	—
VulSeeker ^[48]	√	√	—	LSFG
BiN ^[50]	√	√	√	ACFG
FIT ^[51]	√	—	—	3LACFG
OrderMatters ^[58]	√	—	—	—
DeepBinDiff ^[60]	√	—	√	ICFG
CVSSA ^[80]	√	—	—	ACFG

Genius^[5]最早提出以 ACFG 的形式从二进制函数中提取出其原始特征, 之后, ACFG 的概念被广泛使用在 Gemini^[6], Patchcko^[39], VulSeeker-pro^[46], BiN^[50], CVSSA^[80]等工作中. Genius^[5], Gemini^[6]和 VulSeeker-pro^[45]使用了同样的 8 种属性作为基本块的属性. CVSSA^[80]同样使用了 ACFG, 只是在预处理阶段, 它不仅提取了函数级的特征, 同时也提取了基本块级的特征作为训练的输入. Patchcko^[39]提取了函数级、基本块级和 inter-block 级共 48 个不同的属性作为基本块的属性形成 ACFG. VulSeeker^[48]在 Genius 的基础上, 同时提取了 CFG 中数据的流动 (DFG), 共同形成 LSFG(同时 CFG 和 DFG) 作为提取语义特征的基础. FIT^[51]使用了一个 3LACFG 的图表示方法表示其二进制函数, 3LACFG 同时包括指令级、基本块级和函数级. 同样, discovRE^[4], CVSSA^[80]和 OrderMatters^[58]同样使用 IDA Pro 提取了函数的 CFG 以作为其结构特征的代表, 然而, 与 Gemini 不同的是, OrderMatters^[58]使用 BERT 模型^[81]生成基本块的属性. α Diff^[47]直接应用了一个深度神经网络从每一个函数的原始字节中提取 intra-function 特征, 但是它使用了 CG 来表示 inter-function 语义特征, 并且使用了输入的函数调用关系来刻画 inter-module 语义特征. 通过结合每一个函数的 CFG 和 CG, DeepBinDiff^[60]利用 IDA Pro 来提取基本块信息, 并且生成 ICFG, 提取程序范围的上下文信息, 然而, 除了 ICFG 携带的控制依赖信息外, DeepBinDiff^[60]还通过为每一个基本块生成特征向量考虑了语义信息.

(3) 生成 trace

执行 trace 一般是指 CFG 中连续的基本块组成的序列, 用于表示程序的执行轨迹, 携带了程序的部分控制流信息. BinGo^[36]中使用了与 Tracy^[63]中一样的 Tracelet 来表示二进制的语义特征, 但与 Tracy 不同的是, BinGo 使用的是变长的 Tracelet, 提取符号表达式, 并且使用 Z3^[82]来为基本块生成输入, 并且剪去不可行的执行 trace, 通过输入输出比较基本块的相似度. IMF^[38]则使用 in-memory fuzzing 技术, 对二进制函数进行模糊测试以提升其各种程序行为, 并组成行为轨迹, 作为其向量特征生成的依据. 与之类似, BinMatch^[45]使用动态插桩技术, 通过测试用例的执行来捕获语义签名和函数的运行时信息. Asm2vec^[53], DeepBinDiff^[60]等则通过随机游走 (random walk)^[83]来获取用于训练的执行 trace. 二者均是在图上执行随机游走算法, 得到一些序列, 而这些序列就可以看作是程序的执行 trace. 在最新的补丁检测工作 BinXray^[84]中, 同样使用了执行 trace. 为了精确地表达给定脆弱函数和补丁函数的补丁特征, BinXray 生成两组有效 trace, 一组用于脆弱函数, 一组用于补丁函数. 构建有效 trace 的方法, 是将所有的更改基本块和边界基本块放在一个连通图中, 有效 trace 是图中所有路径. Trex^[59]扩展了 Godefroid 的 micro-execution^[85], 以跨多个指令集架构生成函数的 micro-traces, 作为模型训练的基础, 与先前的静态 trace 不同, micro-trace 不仅包含了一组对齐的指令序列而且还包含了相应的程序状态值.

(4) 汇编代码序列

discovRE^[4]和 Genius^[5]构建描述性的特征, 比如算术汇编指令的比率、传输指令的数目、基本块的数目等来表示函数的特征, 然而, 所有这些方法都假设一种特征或范畴是一个独立的维度, 并没有考虑汇编指令之间的关系. 为此, 相关工作提出将文本语义关系纳入到特征工程过程中, 提出直接从普通的汇编代码中学习更丰富的语义信息, 给定一个二进制文件, 使用 IDA Pro 这样的反汇编器提取出一组汇编函数、其基本块及控制流图列表 (均是由汇编指令及其序列组成), 作为后续特征提取的基础.

如表 3 所示, 文献 [51, 53–60] 等都是以汇编代码序列作为其要分析的目标. Asm2Vec^[53]和 DeepBinDiff^[60]通过随机游走的方式分别从 CFG 和 ICFG 中获取指令序列. MIRROR^[57]通过 llvm pass 编译生成相似/不相似的基本块对作为模型训练的输入. Trex^[59]则是通过 Godefroid^[85]的微执行获取其 micro-trace, 作为指令序列进行分析. 文献 [51, 54–58] 则是通过反汇编提取出其相应粒度 (基本块或函数) 的指令序列.

在 tokens 的粒度方面, Asm2Vec^[53]、MIRROR^[57]和 DeepBinDiff^[60]将操作码和操作数作为单独的 token. Trex 将指令中的所有符号, 包括标点符号 (比如, “,” “[” “]”) 都视作 tokens, 因为 “,” 暗示在其之前和之后的标记分别为 mov 的目标和源 (在 Intel 语法中), 并且 “[” 和 “]” 表示取两者之间的操作数的地址. FIT^[51], InnerEye^[54], Instr^[55], SAFE^[56], OrderMatters^[58]则是将整条指令视作一个 token.

在指令清洗上, FIT^[51]和 Asm2Vec^[53]未做任何操作. InnerEye^[54]和 Instr^[55]使用了同样的指令清洗方法, 使用 0 替换常量值, 使用 <STR> 替代字符串, 使用 FOO 替代函数名, 使用 <TAG> 替代其他标签. TREX^[59]使用 num 替代数

值, 其他不变. DeepBinDiff^[60]使用 `im` 替代数值常量, 根据通用寄存器的长度重新命名通用寄存器, 使用 `str` 替代指针. MIRROR^[57]复杂一些, 它将所有的汇编指令都进行了规一化处理, 分别使用 IMM、ADDRESS、VAR、FUNC 和 BB 替代立即数、地址、变量名、函数名和基本块标记. 寄存器也会根据其使用情况进行归一化, 对于 x86 指令集, 将寄存器分为 14 类: 指针寄存器、浮点寄存器、4 种通用寄存器、4 种数据寄存器和 4 种地址寄存器 (根据其存储数据的长度 (8 bit, 16 bit, 32 bit 和 64 bit) 划分), 对于 ARM 中的寄存器, 归一化为两种: 通用寄存器和指针寄存器. 图 8 给出了几种指令清洗的示例.

表 3 汇编指令序列以及指令清洗方法

方法	粒度	序列生成	tokens	清洗
FIT ^[51]	函数	反汇编	指令	原始指令, 未做清洗
Asm2Vec ^[53]	函数	随机游走	操作码, 操作数	原始指令, 未做清洗
InnerEye ^[54]	基本块	反汇编	指令	常量值->0, 字符串-><STR>, 函数名->FOO, 其他标签-><TAG>
Instr ^[55]	指令	反汇编	指令	同InnerEye
SAFE ^[56]	函数	反汇编	指令	基内存地址->MEM, 绝对值超过5000的立即数->IMM
MIRROR ^[57]	基本块	反汇编	操作码, 操作数	变量->VAR, 立即数->IMM, 地址名->ADDRESS, 函数名->FUNC
OrderMatters ^[58]	函数	反汇编	指令	原始指令序列, 添加特殊token [CLS]和[SEP]
TREX ^[59]	函数	micro-trace	所有符号	数值->num
DeepBinDiff ^[60]	基本块	随机游走	操作码, 操作数	数值常量->im, 通用寄存器根据其长度重命名, 指针->str

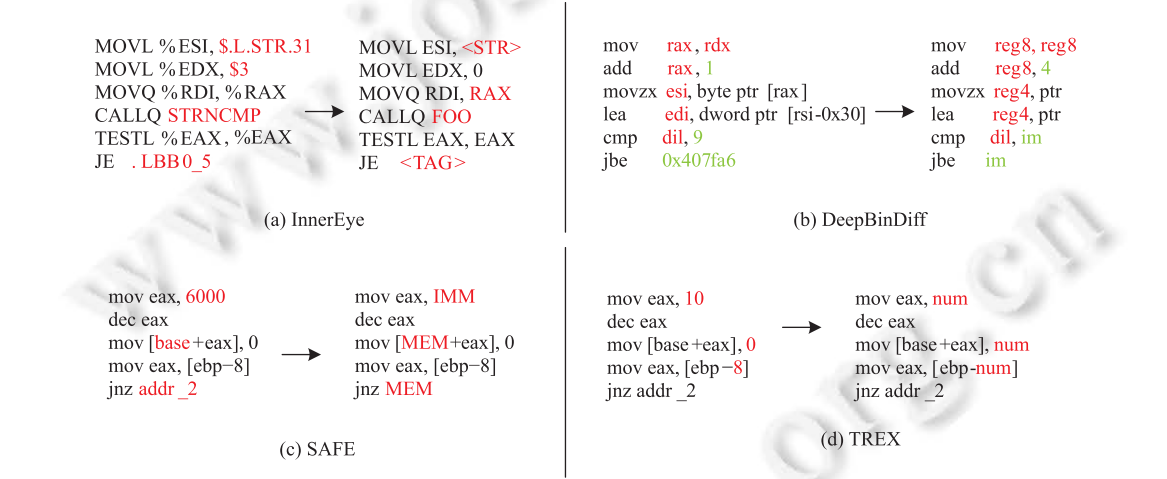


图 8 指令清洗示例

4.3 阶段 2: 特征提取

特征提取阶段是从预处理之后的数据中提取出能够表示程序/函数/基本块语义信息的特征, 以作为后续相似度评估的计算单位. 根据特征提取的智能化水平, 本文将特征提取方法划分为: 基于人工逆向的统计方法, 基于动态执行的语义特征提取和基于学习的自动化特征提取方法.

(1) 人工逆向的统计方法

很多研究使用逆向的方法从目标代码片段的每一个基本块中提取数值特征或结构特征作为目标代码的标识. 数值特征包括指令数目、局部变量的大小、基本块的数量等. 结构特征包括函数的 CFG 以及其基本块的其他特征等. 这种数值形式或结构形式的特征可以直接作为相似度比较的基础, 也可以通过机器学习做后续处理. Genius^[5]使用 IDA Pro 逆向出基本块级别的统计特征 (字符串常量、数值常量、传输指令数量、调用数据、指令数量以及算术指令数量) 和结构特征 (offspring 的数量以及 Betweenness) 作为其 CFG 中结点 (也即基本块) 的

特征, 形成 ACFG 作为其相似度比较的基础. Gemini^[6]的特征提取使用了与 Genius^[5]一样的方法, 只是后续它将形成的 ACFG 作为图嵌入生成模型的输入喂给了模型以进一步训练出图嵌入作为函数的特征进行比较. 在 Genius 的基础上, discovRE^[4], Patchchecko^[39], Vulseeker-pro^[46], aDiff^[47], Vulseeker^[49]等还考虑了来自调用图的数值特征, 主要是度量 callers 和 callees 的数量. 提取这些特征时, 分析人员选择性地应用过程间分析方法, 使用相同库中内部 callees 的入度/出度以及输出库的外部 callees 的速率^[39], 这一点类似于耦合的概念, 它可以分析软件模块之间的相互依赖. 同时, 提取出来的特征也可以使用机器学习进行后期处理^[47]. TiKNiB^[52]结合 IDA Pro, NetworkX^[86]和 Capstone (<https://www.capstone-engine.org/>) 从二进制函数中提取出来 41 种 CFG 级别和 6 种 CG 级别的数值特征, 直接作为后续比较的基础.

(2) 基于动态执行的语义特征提取

为了获取语义特征, 需要执行较为复杂的分析, 比如符号执行, 动态仿真或基于机器学习的嵌入学习. 这里将前两种归纳为动态特征提取. 动态提取出来的特征常与其他的特征一起使用. 一种直接的方法是使用符号约束条件来表示给定的代码片段. 符号约束条件可以表达基本块^[38]、程序切片^[33,35,42]或路径^[41]的输出变量或状态. 另一种直接的方法是使用代码片段的运行时行为来直接表达其语义, 可以通过真实运行或仿真运行来实现. 通过在相同执行环境内执行两个目标函数, 研究人员可以直接比较目标函数所执行的指令序列, 或者访问过的 CFG 边 (也即, 执行过的执行路径). 根据“两个相同的代码片段会产生一致的输入/输出样本”这样的一种直观结论, 研究人员还可以使用输入/输出样本来表示代码片段的语义特征, 生成输入/输出样本的手段有多种: 可以提供随机输入给代码片段产生输入/输出样本, 也可以在代码片段的符号约束之上应用一个 SMT 求解器以生成输入/输出样本, 甚至可以采用过程间分析来精确地模拟输入/输出样本, 如果目标代码包含了一个函数调用的话. BinGo-E^[43]使用 Unicorn^[87]来虚拟执行从初始步骤中提取出来的部分 trace, 并且从中识别出输入/输出值. VulSeeker-pro^[46]在 VEX-IR 上对过滤后的函数执行函数仿真, 然后提取出其输入值、输出值、比较操作码/操作数以及库函数调用作为仿真函数的语义数字签名.

(3) 基于学习的自动化学习方法

近年来, 开始有一些基于机器学习的工作利用机器学习网络从原始二进制代码中提取语义信息以构建特征向量表示. 基于学习的特征学习方法可以构建于人工逆向的统计特征或者通过符号执行/仿真计算出来的动态特征之上, 也可以直接从原始的二进制代码中自动学习. 在 Genius^[5]构建完 ACFGs 之后, 研究人员应用频谱聚簇技术对多个 ACFGs 进行了分组; 研究人员也可以利用流行的编码技术 (比如 Word2Vec^[88], CNN^[89]等) 将其嵌入为一个向量, 比如 Gemini^[6]中使用图嵌入网络 Structure2Vec^[67]将其嵌入为一个向量. VulSeeker^[48]中应用了相同的技术, 只是其操作对象是程序依赖图. 从 2018 年开始, 逐渐兴起了将最新的 NLP 技术^[90]应用到二进制代码相似度分析的工作中, 其核心思想是从原始字节或汇编指令中自动学习出代表其语义及语义关系的特征向量 (也即嵌入), 然后根据不同的粒度再进一步提取相应程度的特征. 比如, Instr^[55]利用 Word2Vec^[88]自动学习指令嵌入, InnerEye^[54]结合 Word2Vec^[88]和 LSTM^[91]自动学习基本块嵌入, Asm2Vec^[53]利用 PV-DM 模型^[92]来学习函数嵌入, SAFE^[56]则是利用自注意力的网络^[93]自动学习函数嵌入. 基于学习的自动化特征提取方法提取出来的特征不依赖于人工逆向的水平, 脱离了人工逆向可能会带来的偏差, 因此, 更能有效地表示目标代码的特征, 进而在最后的相似度比较阶段的精确度也就更高.

4.4 阶段 3: 特征表示

特征提取的目的是将目标代码提升为可进行比较的形式, 这里将其称为特征表示, 可以划分为: 非嵌入表示和嵌入表示两种形式, 如表 4 所示.

(1) 非嵌入表示

1) 输入输出对. 直观上语义相似的代码片段, 给定相同输入, 其行为也会相似. Multi-MH^[3], BinGo^[36], BinGo-E^[43]和 CACompare^[44]均使用输入输出对来表示代码片段. CACompare^[44]不仅提取了输入输出值, 而且还提取了比较操作数和条件代码、库函数调用等作为函数签名. 输入输出能更准确地衡量函数的语义, 也能一定程度上抵御混淆和编译器优化, 但对于每个代码片段都要进行计算, 因此复杂度过高.

2) 执行 trace/运行时行为. 执行 trace 表示程序的执行轨迹, 携带了程序的部分控制流信息. 因此, 常用来表示目标代码. IMF^[38]使用 in-memory fuzzing 技术对二进制函数进行 fuzzing 以提取其程序行为, 组成执行 trace, 作为特征向量, 喂给机器学习模型. MockingBird^[40]基于动态插桩机制, 在程序运行时收集比较操作数对和系统调用属性作为函数的语义特征. BinSim^[42]通过 TEMU^[78]获取程序的执行 trace, 在其之上, 执行增强的动态切片和符号执行提取出其系统调用序列作为其比较相似度的基础. BinMatch^[45]同样利用动态分析从查询函数获取函数语义, 作为其判别相似性的标准. BinXray^[84]同样使用了执行 trace 来代表目标函数的语义.

3) strands. 程序切片是指从程序中提取满足一定约束条件的代码片段. strands 是指通过某个变量向后切片, 获取与计算该变量相关的指令序列. 它在某种程度上代表了部分数据流信息. 文献 [32–34,37] 都使用了程序切片技术, 在基本块级别向后切片, 获取 strands, 作为可比较的代码片段.

4) 符号表达式. Xmatch^[35]提出从原始二进制代码中提取条件公式作为高级别的语义特征来构建代码搜索. 条件公式可以明确地捕获漏洞的两个主要因素: 1) 错误的数据依赖; 2) 缺少或无效的条件检查. 因此, 在条件公式上的二进制代码搜索可以产生更高的精确度, 并且可以提供有意义的证据给分析人员以进行进一步分析搜索结果.

5) 图表示. 对于二进制相似度分析来说, 最直观的便是对结构信息即 CFG 进行比较. discovRE^[4]在 CFGs 上采用了一个过滤器, 以缩小 GI (graph isomorphism)^[94]比较的数量. 它从 CFGs 上提取一些数值特征 (比如, 指令或基本块的数量), 并且使用 kNN 算法^[95]以预过滤相似的 CFGs. Genius^[5]从函数的基本块中提取类似的数值信息, 并且使用来增强 CFG 的顶点, 形成 ACFG, 以表示函数. Gemini^[6]则在 ACFG 的基础上使用了一个端到端的神经网络, 将其嵌入为一嵌入 (向量) 来表示函数. VulSeeker^[48]和 VulSeeker-pro^[46]构建于 Gemini 之上, 将 ACFG 扩展到了 LSFG (使用数据流图的边扩展了 CFG 的图).

表 4 二进制代码相似度分析中使用的特征表示方法

特征表示方法	方法
输入输出对	Multi-MH ^[3] , BinGo ^[36] , BinGo-E ^[43] , CACompare ^[44]
执行 trace	IMF ^[38] , MockingBird ^[40] , BinSim ^[42] , BinMatch ^[45] , BinXray ^[84]
非嵌入表示	strands Esh ^[32] , GitZ ^[33] , Zeek ^[34] , FirmUP ^[37]
符号表达式	Xmatch ^[35]
图	discovRE ^[4] , Genius ^[5]
嵌入表示	Gemini ^[6] , VulSeeker-pro ^[46] , aDiff ^[47] , VulSeeker ^[48] , BiN ^[50] , FIT ^[51] , Asm2vec ^[53] , InnerEye ^[54] , Instr ^[55] , SAFE ^[56] , MIRROR ^[57] , OrderMatters ^[58] , Trex ^[59] , DeepBinDiff ^[60] , ImOpt ^[76]

(2) 嵌入表示

“嵌入”^[96]在数据上表示为一个映射: $f: X \rightarrow Y$, 也就是一个函数. 其中, 该函数是一个单射函数 (每一个 Y 只有一个唯一的 X 对应, 反之亦然) 且是结构保留的 (比如在 X 所在空间存在 $X1 < X2$, 那么映射后, 在 Y 所属空间 $Y1 < Y2$). 在这里, 嵌入表示即是: 使用某种嵌入方法将从二进制函数中提取出来的特征映射到另外一个空间, 其中这个映射具有单射性和结构保留的特点. 图 9 给出了 Gemini 所使用的 ACFG 图及图嵌入示意图, 其中图 9(a) 的 ACFG 即是从二进制函数中提取出来的图表示特征, 图 9(b) 以图 9(a) 中的 ACFG 为输入, 经过神经网络的 T 次迭代之后, 生成一个特征向量 μ , 即为图嵌入.

如表 5 所示, 在二进制相似度分析工作中, 使用的嵌入可以根据其粒度划分为函数嵌入、基本块嵌入和指令嵌入. Gemini^[5], VulSeeker-pro^[46], VulSeeker^[48], IoTSeeker^[49], BiN^[50]均是以函数为粒度, 使用 Structure2Vec 模型^[67]将 ACFG 嵌入生成一个图嵌入. aDiff^[47]: (1) 使用 CNN 模型学习 intra-function 的语义特征, (2) 提取函数调用图中节点 (也即函数) 的入度和出度, 形成一个 2 维向量作为其 inter-function 的语义特征, (3) 使用公式 $h(set, superset) = \langle x_1, x_2, \dots, x_N \rangle$ 将函数的输入函数集合嵌入为一个向量表示其 inter-module 的语义特征, 然后使用这 3 个特征向量的和作为其函数嵌入. FIT^[51]的函数嵌入是一个含有 3 级特征的函数嵌入: 它首先使用 SGNS (skip-gram negative sampling) 模型^[97]生成词嵌入, 然后使用 LSTM^[91]生成基本块嵌入, 最后使用 Structure2Vec 生成图嵌入. Asm2Vec^[53]使用 PV-DM 模型^[92]同时学习 token 级的嵌入, 然后再使用 Structure2Vec 模型在 token 级的嵌入之上

生成函数嵌入。与之类似, InnerEye^[54]首先使用 Word2Vec 学习词嵌入, 然后以学习到的词嵌入序列作为输入, 使用 LSTM 将基本块编码为基本块嵌入。InnerEye 是首个基本块级别的嵌入生成方法。Instr^[55]则是首个指令级别的嵌入生成方法, 与 InnerEye 相同, 它将整条指令视为一个词, 使用 CBOW 模型^[98]学习词嵌入, 同时利用联合学习 (joint learning)^[99]方法学习跨体系结构的指令嵌入。SAFE^[56]同样将指令视为词, 使用 skip-gram 模型^[100]生成词嵌入, 并且针对 InnerEye 所使用的 LSTM 不能很好地处理长序列的问题, SAFE 引入自注意力网络 (self-attention network, SAN)^[93]生成函数级别的嵌入。MIRROR^[57]基于 NMT (neural machine translation)^[101]的 encoder 和 decoder 机制实现了一种跨 x86-ARM 的基本块嵌入生成方法, 使得不同平台基本块的丰富语义信息映射到相同的向量空间, 并且相同语义的 x86 基本块和 ARM 基本块会尽可能的近, 且不同语义的 x86 基本块和 ARM 基本块会尽可能的远。OrderMatters^[58]使用 BERT^[81]学习词和基本块嵌入, 使用带有 GRU^[102]更新函数的 MPNN^[103]学习图嵌入, 使用一个类似于 ResNet 网络的 CNN^[104]学习 order 信息, 最终将图嵌入和 order 嵌入组合在一起形成函数嵌入。Trex^[59]使用 BERT 模型的 MLM 任务学习微迹中的指令之间的关系, 并使用一个 2 层的多层感知机 (multi-layer perceptions, MLP) 模型在其基础上生成嵌入。DeepBinDiff^[60]首先计算出二进制文件的程序范围内的过程间控制流图, 然后, 基于 TADW 算法^[105]来自动生成基本块级的嵌入。

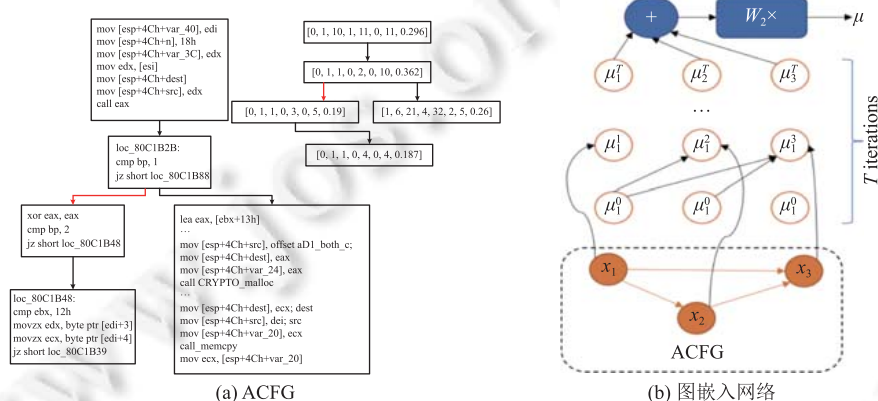


图9 ACFG 和图嵌入示意图 (摘自 Gemini^[6])

表5 基于学习的方法中嵌入生成所使用的模型

方法	嵌入粒度	嵌入生成模型
Gemini ^[5]	函数	Structure2Vec ^[67]
VulSeeker-pro ^[46]	函数	Structure2Vec
α Diff ^[47]	函数	Structure2Vec
VulSeeker ^[48]	函数	Structure2Vec
IoTSeeker ^[49]	函数	Structure2Vec
BiN ^[50]	函数	Structure2Vec
FIT ^[51]	函数	SGNS ^[97] , LSTM ^[91] , Structure2Vec
Asm2Vec ^[53]	函数	PV-DM ^[92] , Structure2Vec
InnerEye ^[54]	基本块	Word2Vec ^[88] , LSTM
Instr ^[55]	指令	CBOW ^[98] , joint learning ^[99]
SAFE ^[56]	函数	skip-gram ^[100] , Self-attention network ^[93]
MIRROR ^[57]	基本块	NMT (Transformer) ^[101]
OrderMatters ^[58]	函数	BERT ^[81] , MPNN ^[103] , CNN ^[104]
Trex ^[59]	函数	BERT, 2-layer MLP
DeepBinDiff ^[60]	基本块	CBOW ^[98] , TADW ^[105]

4.5 阶段 4: 相似度匹配

如表 6 所示, 二进制相似度分析技术所使用的相似度匹配算法可以划分为基于哈希匹配、基于统计相似性、基于图匹配、基于学习以及综合相似度的匹配算法.

表 6 二进制相似度分析方法所使用的相似度匹配算法

相似度匹配算法		二进制相似度分析方法	相似度计算公式
基于哈希匹配	BHB	Multi-MH ^[3]	未明确
	strands归一化	GitZ ^[33]	公式(1)
	JCS ^[106]	BinGo ^[36] , BinGo-E ^[43]	公式(2)
	Jl ^[107]	IMF ^[38] , MockingBird ^[40] , CACompare ^[44]	公式(3)
	SimHash ^[108] 的Hamming距离 ^[109]	BinMatch ^[45]	公式(4)
	指令间距离	BinXray ^[84]	公式(5), 公式(6)
	线性求解	Xmatch ^[35]	公式(7)
统计相似性	约束求解	Esh ^[32]	SMT求解
	平均值	BinSim ^[42]	公式(8)
图匹配	MCS ^[110]	discovRE ^[3]	公式(9), 公式(10)
	bipartite图匹配 ^[111]	Genius ^[5] , CVSSA ^[80] , BiN ^[50] , FIT ^[51]	公式(11), 公式(12)
基于学习	cosine距离 ^[112]	Gemini ^[6] , VulSeeker ^[48] , VulSeeker-Pro ^[46] , SAFE ^[56] , Trex ^[59]	公式(13), 公式(14)
	嵌入距离	OrderMatters ^[58] , Asm2Vec ^[53]	未明确
	深度学习分类器	Zeek ^[34]	未明确
	Manhattan距离 ^[113]	InnerEye ^[54]	公式(15), 公式(16)
	Euclidean距离 ^[113]	MIRROR ^[57]	公式(17), 公式(18)
	k-Hop贪婪算法	DeepBinDiff ^[60]	未明确
综合相似度	相似度组合	α Diff ^[47]	公式(19)
	相似度平均	Patchcko ^[39]	公式(20), 公式(21)
	相对差的平均值	TiKNib ^[52]	公式(22)

注: BHB: best-hit-broadening, JCS: Jaccard containment similarity, Jl: Jaccard index, MCS: maximum common subgraph isomorphism

(1) 基于哈希的匹配

Multi-MH^[3]使用 MinHash^[114]对待比较的 I/O 对 (表示基本块) 计算出一个语义哈希, 然后通过比较两个基本块的语义哈希, 而不是直接比较 I/O 对的集合来提升基本块相似度比较的效率. Multi-MH 使用 BHB 算法来比较两个基本块图之间的相似度. 同样, GitZ^[33]使用 $R(p)$ 表示给定程序 p 经过切片且归一化后的 strands 的 MD5^[115] 的哈希值: $R(p) = \{MD5(Canonicalize\&Normalize(s_p)) | s_p \in p\}$, 然后 GitZ 将 q 和 t 之间的相似度分值计算为 q 和 t 之间共享的所有 strands 的逆向频率的总和和被 P 中所有特有的 strands 的数量归一化后的值:

$$S_P(q, t) = \sum_{s \in (R(q) \cup R(t))} \frac{|P|}{f(s)} \tag{1}$$

BinGo^[36]和 BinGo-E^[43]则使用 JCS^[106]来计算两个不同函数模型之间的相似度分值 (M_{sig} 和 M_{tar} 分别是签名和目标函数生成的函数模型):

$$Sim(M_{sig}, M_{tar}) = \frac{M_{sig} \cap M_{tar}}{M_{sig}} \tag{2}$$

IMF^[38], MockingBird^[40]和 CACompare^[44]则采用了 Jaccard 系数 $Jl^{[107]}$ 来计算之前提取出来的特征序列的相似度:

$$J(T_1, T_2) = \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|} = \frac{|T_1 \cap T_2|}{|T_1| + |T_2| - |T_1 \cap T_2|} \quad (3)$$

其中, $|T_1 \cap T_2|$ 表示 T_1 和 T_2 之间的 LCS 的长度, 而 $|T_1|$ 和 $|T_2|$ 分别表示 T_1 和 T_2 的长度. Jaccard index 越接近于 1, 两个序列越被认为是相似的. 不同的是 CACompare^[44] 中采用了一个阈值来决定使用 MinHash 估算 Jaccard 系数的时机. 当要比较的两个特殊序列的长度超过阈值时, 便使用传统的最长公共子序列算法作序列比较. 而 IMF 则是将计算出来的 Jaccard 系数作为一个特征值, 组成特征向量, 喂给一个基于树的模型进行训练, 最终输出一个标签, 以识别两个函数是否相似.

BinMatch^[45] 则使用两个集合的 SimHash 值^[108] 的 Hamming 距离^[109] 来比较其相似度 ($SH_F(S_r), SH_F(S_t)$ 分别是两个集合的 SimHash 值, 其值范围在 $0 \sim 2^F$ 之间, F 表示 SimHash 值的尺寸):

$$Sim(S_r, S_t) = 1 - \frac{HD[SH_F(S_r), SH_F(S_t)]}{F} \quad (4)$$

虽然 BinXray^[84] 也使用了有效 trace 来表示其函数, 然而其相似度比较方法与上面几种基于 trace 的相似度比较方法不同, 它使用了精确到指令间距离的相似度比较方法:

$$Sim(T_1, T_2) = \sum_{t_1 \in T_1, t_2 \in T_2} \frac{Sim(t_1, t_2) * (len(t_1) + len(t_2))}{|T_1| * len(T_2) + |T_2| * len(T_1)} \quad (5)$$

其中, $Sim(t_1, t_2)$ 表示每一个 trace 对的相似度分值:

$$Sim(t_1, t_2) = \frac{\max(len(t_1), len(t_2)) - edit(t_1, t_2)}{\max(len(t_1), len(t_2))} \quad (6)$$

Xmatch^[35] 将条件公式匹配建模为一个线性分配问题, 利用整数规划技术来查找最优解, 其相似度计算公式下:

$$Sim(f_1, f_2) = 1 - \frac{dist(f_1, f_2)}{\sum_{i=1}^n len(cf_i) + \sum_{j=1}^m len(\widehat{cf_j})} \quad (7)$$

其中, cf_i 表示第 i 个条件公式, $len(cf_i)$ 是对每一个条件公式的字符串长度进行计数.

(2) 基于统计相似性的匹配

虽然 Esh^[32] 与 GitZ^[33] 一样都是将程序分解为可比较的 strands, 但 Esh 使用程序验证程序 SMT 求解器^[82] 来检查两个 strands 在语义上是否等价 (而不是对 strands 做哈希), 因此其性能严重受限. 具体方法上是假设输入等价, 检查中间值和输出值是否相同. 若不等价, 则定义一个基于匹配值与 strands 中值总数的比例的 strands 相似性的定量概念.

BinSim^[42] 的相似度计算公式为:

$$Sim(a, b) = \frac{\sum_{i=1}^n SimilarityScore}{Avg\{|T_a|, |T_b|\}} \quad (8)$$

其中, $\sum_{i=1}^n SimilarityScore$ 是所有对齐的系统调用对的相似度分值的总和. T_a 和 T_b 是从程序 a 和 b 收集来的系统调用序列, n 是对齐的系统调用的数量. $Sim(a, b)$ 的取值范围是 0.0 到 1.0, $Sim(a, b)$ 的值越高, 表示两个 traces 越相似.

(3) 基于图匹配的匹配

图匹配试图在两个或多个图结构之间, 建立结点与结点之间的对应关系. 在计算机视觉领域, 图匹配算法通常被用于求解多个图像之间, 关键点到关键点的匹配关系. 相比于只考虑节点与节点之间的一阶相似度关系的点匹配, 图匹配还考虑了图结构中边到边的二阶相似度.

discovRE^[3] 使用的图匹配算法基于 MCS^[110] 来发现两个图之间的最大子图同构, 考虑到两个基本块之间的相似度, discovRE 将距离函数扩展为以下形式:

$$d_{mcs}(G_1, G_2) = 1 - \frac{|mcs(G_1, G_2)| - \sum d_{BB}(b_i, b_j)}{\max(|G_1|, |G_2|)} \quad (9)$$

其中, d_{BB} 表示基本块距离:

$$d_{BB} = \frac{\sum a_i |c_{if} - c_{ig}|}{\sum a_i \max(c_{if}, c_{ig})} \quad (10)$$

Genius 和 CVSSA 则使用了 bipartite 图匹配算法^[111], 两个图之间的 ACFG 相似度表示为:

$$k(g_1, g_2) = 1 - \frac{\text{cost}(g_1, g_2)}{\max(\text{cost}(g_1, \Phi), \text{cost}(\Phi, g_2))} \quad (11)$$

其中, $\text{cost}(g_i, g_j)$ 表示两个图 g_1 和 g_2 之间的最优 bipartite 匹配, 转化为查找两个基本块之间的距离计算问题:

$$\text{cost}(v, \hat{v}) = \frac{\sum_i \alpha_i |a_i - \hat{a}_i|}{\sum_i \alpha_i \max(a_i, \hat{a}_i)} \quad (12)$$

其中, a_i 和 $\hat{a}_i$ 分别是两个基本块 v 和 $\hat{v}$ 的特征向量的第 i 个特征, α_i 是相应的特征权重. 如果特征是一个常量的话, $|a_i - \hat{a}_i|$ 就是其区别. 如果特征是一个集合, 则使用 Jaccard 系数来计算集合的区别.

(4) 基于学习的匹配

基于学习的方法通过将训练样本 (f_1, f_2, y) 或者 (B_1, B_2, y) 喂给学习模型, 训练模型识别样本的相似/不相似的能力, 使得相似样本在向量空间越来越接近, 不相似的样本越来越远. 由于样本最终都会通过某种方式转化为向量, 因此, 样本之间的相似度比较, 就转化成了向量之间的相似度计算.

Gemini^[6], VulSeeker^[47], VulSeeker-Pro^[45], SAFE^[53], Unsupervised^[116]均使用 Cosine 距离^[112]来计算其图嵌入向量 ($\tilde{\mu}$ 和 $\tilde{\nu}$ 分别表示函数 f 和 f' 的嵌入向量) 之间的相似度:

$$\text{Sim}(f, f') = \cos(\tilde{\mu}, \tilde{\nu}) = \frac{\tilde{\mu} \cdot \tilde{\nu}}{\|\tilde{\mu}\| \|\tilde{\nu}\|} \quad (13)$$

为了训练模型参数, 它们将问题转化为下面的目标函数:

$$\min_{W_i, W_f, \dots, v_o} \sum_{i=1}^N (y_i - \text{Sim}(f_i, f'_i))^2 \quad (14)$$

Asm2Vec^[53]和 OrderMatters^[58]并未明确说明其相似度匹配所使用的算法, 然而, 虽然 OrderMatters 使用了 3 种级别的学习模型来学习函数的特征, 但是它最终组合在了一个函数嵌入中. Asm2Vec 则是训练一个模型, 学习汇编指令及指令序列的特征向量, 以形成一个嵌入, 因此, 本文估计其使用的与上面几种方法一样的相似度计算算法.

Zeek^[34]将具有相同语义但不同语法的 strands 规一化为相同的文本表示, 并在其之上应用 b -bit 的 MD5 哈希, 得到一个 $\{0, 2^b - 1\}$ 范围内的整数, 随后使用上一步产生的整数集合作为索引构建一个长度为 2^b 的向量, 最后, 使用一个深度学习分类器 (skeleton)(由 1 个输入层 (2×2^b 个神经元), 两个全连接隐藏层和一个 Softmax 输出层组成) 来学习两个输入向量 (表示两个程序) 之间的相似度.

InnerEye^[54], MIRROR^[57]和 DeepBinDiff^[60]是基本块级别的嵌入生成和相似度比较工作. 但是其各自使用的基本块相似度比较算法则不同. InnerEye 使用 Manhattan 距离^[113]来计算基本块 B_1 和 B_2 之间的相似度 (其中, $\mathbf{h}_T^{(1)}$ 和 $\mathbf{h}_S^{(2)}$ 分别是 B_1 和 B_2 的向量, 取自 LSTM 的最后一层的最后一个隐藏状态):

$$\text{Sim}(B_1, B_2) = \exp(-\|\mathbf{h}_T^{(1)} - \mathbf{h}_S^{(2)}\|_1) \quad (15)$$

与 Gemini^[6]等工作一样, 为了训练网络参数, InnerEye 使用了随机梯度下降 (stochastic gradient descent, SGD) 算法^[117]来最小化 loss 函数 (其中, y_i 是 $\langle B_1^i, B_2^i \rangle$ 对的相似度真实情况, N 是训练数据集中的基本块对的数量):

$$\min_{W_i, W_f, \dots, v_o} \sum_{i=1}^N (y_i - \text{Sim}(B_1^i, B_2^i))^2 \quad (16)$$

MIRROR^[57]使用 Euclidean 距离^[113]来度量基本块的相似度:

$$\text{Sim}(B_1, B_2) = \exp\left(-\frac{D(E_1, E_2)}{d}\right) \quad (17)$$

其中, E_1 和 E_2 分别是基本块 B_1 和 B_2 的嵌入, 很明显, Euclidean 距离越小, 两个基本块越相似. 由于其引入了正负样本来训练和微调模型, 因此, 它使用了一个基于 margin 的三元损失函数^[118]来训练模型 ($\gamma > 0$ 是 margin 参数):

$$L = \max \{D(E_1, E_2) - D(E_1, E_3) + \gamma, 0\} \quad (18)$$

DeepBinDeep^[60]引入了一个 k-Hop 贪婪匹配算法: k-Hop 贪婪匹配算法的上层思想是受益于 ICFG 上下文信息, 根据已经匹配到的基本块的 k-Hop 邻居内的基本块嵌入计算出来的相似度来匹配基本块. 由于这个 k-Hop 贪婪匹配算法比较复杂, 所以本文并没有列出其具体的公式.

(5) 基于综合相似度的匹配

由于 αDiff ^[47]使用了 3 种特征来表示两个函数, 因此, 它使用了 3 个特征之间的距离来计算两个函数之间的总的相似度, 其计算公式为 (其中, D_1, D_2, D_3 分别表示函数内, 函数间及模块内特征之间的距离):

$$D(I_q, I_t) = D_1(I_q, I_t) + (1 - \mathcal{Z}^{D_2(I_q, I_t)}) + D_3(I_q, I_t) \quad (19)$$

同样, Patchchecko^[39]使用动态执行行为来代表函数的特征向量, 并且使用多个执行环境中的相似度的平均值来作为每一个函数对 (f, g) 的相似度:

$$\text{Sim}(f, g) = \frac{1}{K} \sum_{i=1}^K d_k(f, g) \quad (20)$$

其中, f 表示 CVE 函数, g 表示目标固件中的候选函数, K 为执行环境总数, k 为所使用的第 k 个执行环境. $d_k(f, g)$ 表示第 k 个执行环境的相似度距离:

$$d_k(f, g) = \left(\sum_{i=1}^n |x_i - y_i|^p \right)^{1/p} \quad (21)$$

TiKNib^[52]取两个函数的所有特征之间的相对差的平均值来计算两个函数 A 和 B 之间的相似度分值 (其中, δ 表示两个特征值之间的相对差):

$$s(A, B) = 1 - \frac{(\delta(A_{f_1}, B_{f_1}) + \dots + \delta(A_{f_N}, B_{f_N}))}{N} \quad (22)$$

4.6 二进制相似度分析技术评估

4.6.1 评估数据集和评估标准

(1) 评估数据集

如第 4.2.1 节所示, 二进制相似度分析技术用于评估其所提方法的数据集通常来源于 3 个方面: 基线数据集、固件数据集和漏洞数据集. 基线数据集常常是由一些开源项目通过不同的配置场景 (编译器、优化选项、混淆、平台等) 编译而来的二进制可执行文件组成的数据集, 现有工作中常用到的开源项目有: OpenSSL (<https://github.com/openssl/openssl.git>), BusyBox (<https://git.busybox.net/busybox>), Coreutils (<https://www.gnu.org/software/coreutils/>) 等, 这些都是嵌入式设备固件中常用到的库函数. 固件数据集通常是从厂商 Web 站点或 FTP 站点爬取的固件组成的固件镜像集合, 通过 Binwalk 解包获取的现实二进制文件数据集, 漏洞数据集通常是根据 CVE 编号从 CVE 站点搜集而来的漏洞, 编译形成的二进制文件集合. 本文将在第 5 节详细讨论现有的基于二进制相似度的固件漏洞搜索工作中应用的数据集.

(2) 评估标准

在评估标准方面, 现有工作大多从: (1) 精确度, (2) 性能, (3) 漏洞搜索通力, (4) 跨编译器/跨体系结构/跨优化选项/跨版本的相似度比对能力等几个方面进行实验验证. 精确度常用来衡量所提方法在验证数据集上的表现. 不同的方法会使用不同的评估指标, 常用的方法有: 准确率 (accuracy), 即模型预测正确数量所占总量的比例; 精确率 (precision), 即识别为正类别的样本中, 为正类别的比率; 召回率 (recall), 即所有正类别的样本中, 被正确识别为正类别的比率; F1-Measure, 又称 F-Score, 是 precision 和 recall 的加权调和平均, 常用于评估分类模型的好坏; 还有一些工作, 使用 precision@k 或 recall@k 来度量正确结果是否出现在 top-K 中. 性能评估通常是计算所提方法各步骤的时间效率、数据体量、参数选择以及模型选择等对精确度和时间的影响等等, 是大规模漏洞搜索要考虑的因素. 漏洞搜索通力通常是用来评估所提方法在实际场景中的应用能力, 以证明所提方法的有效性. 跨编译器/跨体系结构/跨优化选项/跨版本的相似度比对能力是检测所提方法面对输入变化时的能力, 如第 3 节所示, 相同的源

代码在不同的配置场景下所生成的控制流程图或指令序列是不同的,“完美”的方法应该能够满足所有应用场景,然而现实中并不存在这样的“完美”方法.到目前为止,尚不存在一种方法可以处理所有的应用场景.

4.6.2 现有二进制相似度分析工作在评估方面的总结

本文对所研究的论文列表中用于评估的基准数据集进行了详细分析,得出了以下结论.

(1) 到目前为止,尚不存在任何两个工作使用了相同的评估基准数据集(除了同一作者的改进工作之外,比如 VulSeeker^[48]和 VulSeeker-pro^[46]使用了相同的数据集),这里所说的“相同”是指使用的源代码(版本也相同)、编译器、优化选项、有无使用混淆技术等都是一样的.其中一些工作虽然使用相同的源码包,比如 OpenSSL,但由于版本、编译器、优化选项等的不同,所以生成的二进制文件也是不同的.这很大程度上是因为制定统一的基准测试集比较繁琐且很困难,在环境设置方面需要很大的努力,特别是交叉工具链的构建、交叉编译不同的架构、使用不同的编译器进行交叉编译等等.因此,论文中提到的“与先前工作的对比”往往是针对先前工作自己得出来的结论进行的比较,除非“先前工作”开源了其所使用数据集的具体配置环境和编译条件,可供研究人员使用与其完全一样的编译环境编译出与其所用数据集完全一样的数据集.

(2) BINKIT^[52]构建了一个可解释的模型,基于新的相似度评分标准来评估每一个特征,然后直接度量两个不同特征值之间的区别,也即,它使用了一个可解释的模型捕获了每一个特征在不同编译选项、不同体系架构、不同优化选项、不同混淆技术之间的区别有多大.然而,此研究中使用的特征是 presemantic 特征(语义前特征,即语法或结构特征),这些特征是通过人工逆向从函数的 CFG 或 CG 中提取出来的,比如,基本块的数量、边的数量、函数调用入度、函数调用出度等这样的特征,与之前的所有人工选择出来的特征一样,不足以充分表示函数的语义的有效性,因此,这个结论的可靠性有待进一步论证.但是,它提供了一套完整的 benchmark 编译脚本(<https://github.com/SoftSec-KAIST/binkit>),可以帮助自动化编译、复现及扩展不同的研究目标,为构建统一的评估数据集提供了很好的支撑条件.

(3) 正如第(1)点所述,现有二进制相似度分析工作所使用的基准数据集、评估标准并不完全相同,因此,很难横向比较现有工作的优劣程度(也无法对评估标准的准确性进行评估),加之篇幅的限制,所以,本文并未对现有工作的评估标准和评估结果进行详细的阐述.

4.7 现有二进制相似度分析工作的综合比较

后文表 7 列出了本文研究范围内的二进制相似度技术的一个纵向比较,具体表现在能够处理的操作系统类型(Windows 和 Linux, C1)、处理器类型(x86, ARM, MIPS, C1, C2)、所使用的编译器(gcc 和 clang, C3)及其优化选项(O0-O3, D: 缺省, A: 全部, C3)、是否使用混淆技术(C5)、是否考虑了函数内联(C4)、分析对象(二进制或者固件, C10)、分析粒度(程序、函数、基本块, C6, C8)、所使用的预处理工具(IDA pro, BAP, Angr, 或动态插桩工具)、是否将二进制转化为了中间语言(C2, C10)、特征提取方法(统计、动态提取、基于学习, C8, C9)、特征表示方法(非嵌入: 输入输出对、符号表达式、strands、traces 或者图, 嵌入, C8, C9, C10)、特征编码方法以及所使用的相似度算法等.

如表 7 所示, 100% 的二进制相似度分析工具都提供了对 Linux 平台的支持, 20.6% (7/34) 的工具支持 Windows. 支持 x86、ARM 和 MIPS 处理器体系的工具分别占: 97.1% (33/34), 73.5% (25/34) 和 70.6% (24/34), 其中只有 BinARM^[119]是专用于 ARM 型 IED 固件的. 85.3% (29/34) 的工具考虑了编译器的优化级别, 14.7% 的工具使用了编译器的缺省优化, 还有一些工作考虑了更多的优化选项, 比如 discovRE^[13]考虑了标准的编译器优化选项, 总共有 1711 种不同的编译器配置, Patcheko 的优化级别除了 O0-O3 之外, 还考虑了 Oz 和 Ofast 两种优化级别. 除了 BinARM^[119]是专门针对固件进行分析的工具, 其余的 97% (33/34) 都是以二进制作为分析对象, 其中 38.2% (13/34) 考虑了固件或者使用了固件作为 real-world 的案例分析. 91.2% (31/34) 的工具以函数为分析粒度, 以基本块为分析粒度的工具占比 20.1% (7/34), 以程序为分析粒度的工具最少, 占比仅 9.7% (3/34). 23.5% (8/34) 的工具考虑了使用 obfuscator-llvm^[120]的混淆, 17.6% (6/34) 的工具在其工具设计中考虑了二进制代码的函数内联情况, 14.7% (5/34) 的工具考虑了补丁识别情况.

在预处理过程中, 79.4% (28/34) 的工具使用了 IDA Pro 作为其静态反汇编工具, 11.8% 的工具使用了 BAP^[70], 8.8% 的工具使用了 Angr^[71], 还有一些工具由于要收集动态 trace, 因此使用了一些动态平台, 比如 Valgrind^[72],

TEMU^[78], PIN^[121]等. 这类工具的占比约 14.7%, 比如, IMF^[38]使用 PIN 作为插桩工具, MockingBird^[40]使用 Valgrind 作为插桩工具, Patchchecko^[39]的动态插桩则有两种, 一种是对 Android 和 Android Things 平台, 实现了一个基于 GDB^[122]和 GDBServer^[123]的动态插桩插件, 另一种是对 iOS 平台, 实现了一个基于 IDA Pro 和 debugserver^[124]的动态插桩插件. 35.3% (12/34) 的工具会首先将二进制代码转化为 IR, 第 4.3.2 节详细描述了相关工具使用了什么样的二进制分析平台将目标二进制代码转化为 IR, 在此不再赘述.

在特征提取方法中, 逆向统计、静态分析、动态提取和基于学习的占比差不多, 分别为 35.3%, 23.5%, 26.5% 和 26.5%. 在特征表示中, 以嵌入为表示方法的工具最多, 占 47%, 并且大多分布在 2018–2020 年, 这也充分说明了近年来, 深度学习技术在二进制相似度分析领域逐渐得到了广泛的应用.

5 二进制代码相似度分析在嵌入式设备固件漏洞搜索中的应用研究

作为一种典型应用, 二进制代码相似度分析特别是跨体系架构的二进制代码相似度分析技术常被用于嵌入式设备固件漏洞搜索. 尽管理论上来说, 跨体系结构的二进制相似度分析技术都可以应用到嵌入式设备固件的漏洞搜索中, 然而, 在本文分析的文献列表中, 只有 13 篇论文 (约 38.2%) 将在固件中的漏洞搜索作为其典型应用进行了阐述. 本文单独抽取了这 38.2% 的工具来阐述二进制代码相似度分析在嵌入式设备固件漏洞搜索中的应用, 如后文表 8 所示. 表 8 列出了工具名称、所使用的数据集、漏洞搜索的 CVE 编号、针对 CVE 编号进行编译的开源库以及搜索精确度以及搜索所需的时间开销.

(1) 2015–2020 年间, 基于二进制代码相似度分析的嵌入式设备固件漏洞搜索包含但不仅限于这 38.2% 的分析工具. 比如, Costin 等人^[9]首次提出基于模糊哈希的大规模嵌入式固件安全分析方法, 通过固件模块关联的方法, 采用模糊哈希算法 (ssdeep^[125]和 sdhash^[126]) 扫描不同固件镜像之间存在的类似漏洞. 之后, 李登等人^[127]提出基于第三方库的同源性分析结果对同类固件的漏洞检测也是基于模糊哈希算法实现的, 能够准确地对不同类型、架构、版本的固件进行漏洞检测. 同样, IHB^[128]设计并实现了一种时间复杂度为 $O(\lg N)$ 同源二进制文件检索方法. 它通过深度学习网络 Bi-GRU^[129]对二进制文件中的可读字符串进行编码, 然后对编码向量生成局部敏感哈希 (locality sensitive hashing, LSH) 从而实现快速检索. 这种基于模糊哈希的方法粒度是基于文件级的, 精确度不高, 因此本文并没有将它们列入对比的行列.

(2) 表 8 中列出了 12 篇论文, 其中 IoTSeeker 是表 7 中的 VulSeeker 和 VulSeekerpro 工作的延伸.

5.1 基于二进制代码相似度分析技术的固件漏洞搜索现有工作分析

如表 8 所示, 在涉及固件漏洞检测的二进制相似度分析工作中, 数据集除了第 4.2 节所阐述的 3 种类型: 基线数据集、固件数据集和漏洞数据集之外, 还可能包含其他数据集. 在 12 项工作中, 其中有 2 项工作使用了其他数据集: 为了评估工具的效率, Gemini^[6]使用不同尺寸 (也即, 图中的顶点的数量不同) 的 ACFGs 构建了一个数据集, 最终获取了一个由 3 037 个 ACFGs 组成的数据集, 在表 8 中记为 IV-1. 为了训练指令嵌入模型, FIT^[51]引入了一个块内指令序列 (in-block instruction sequence, IBIS) 数据集, 它由 3 个子集组成, 分别包含 x86, MIPS 和 ARM 形式的 OpenSSL 二进制文件中的 493 540 510 772 和 475 927 个 IBISs, 在表 8 中记为 IV-2. 此外, 为了进行图匹配效果的评估, 包括跨体系结构和跨版本的评估, FIT 还引入了另一个数据集, 它是由 CoreUtils (v6.6, v8.2, v8.31), 使用 gcc (v5.4/v7.5), 4 种优化级别 (O0–O3), 针对 3 种体系结构 (x86, MIPS, ARM) 编译而成的, 在表 8 中记录为 IV-3.

在这 12 项工作中, 有 2 项工作 (2/12, 16.7%) 没有使用基线数据集: BinARM^[119]和 FirmUp^[37]. 这是因为 BinARM^[119]和 FirmUp^[37]都不是基于学习的方法, 不需要进行模型的预训练和评估, 因此不需要构建基线数据集. 1 项工作 (1/12, 8.3%) 使用了 Android 库形成的基线数据集: Patchchecko^[39]. 剩余的 9 项工作 (9/12, 75%) 均使用了嵌入式设备固件中常用的一些库的源代码, 进行编译生成基线数据集. 在所使用的开源项目中, OpenSSL 和 BusyBox 应用最广泛, 有 8 项 (8/9, 88.9%) 使用了 OpenSSL, 有 7 项 (7/9, 77.8%) 使用了 BusyBox. Gemini^[6]仅使用了 OpenSSL, 没有使用 BusyBox. discovRE^[4]既没有使用 OpenSSL 也没有使用 BusyBox. 在构建基线数据库时的编译器和优化选项方面, gcc 应用最多 (9/9, 100%), 优化选项 O0–O3 应用最多 (7/9, 77.8%).

表 7 二进制相似度分析技术比较

时间	方法	OS	处理器		编译器	优化	混淆	函数内联	补丁	分析对象		分析粒度			预处理工具				特征提取方法			特征表示					特征编码	相似度算法			
Windows	Linux	X86	ARM	MIPS	clang	gcc				二进制的	固件	程序	函数	基本块	IDA Pro	BAP	Angr	动态插桩	IR	逆向统计	动态提取	基于学习	I/O 对	符号表达式	strands	traces	图	嵌入			
2015	Multi-MH ^[3]																													MinHash	BHB
2016	discoverE ^[4]																													kNN	MCS
	Genius ^[5]																													LSH	JD
	Esh ^[32]																													VCP	GES
	BinGo ^[36]																													SE	JCS
	MockingBird ^[40]																													MD5	JI
2017	Xmatch ^[35]																													CF	自定义
	Gemini ^[7]																													DNN	cosine
	GitZ ^[33]																													MD5	自定义
	BinSim ^[42]																													-	自定义
	IMF ^[38]																													JIR	Extra-trees
	CACCompare ^[44]																													MinHash	JI
	Cop ^[41]																												SF	LCS	
2018	BinArm ^[119]																													多维特征	自定义
	BinGo-E ^[43]																													多维特征	JCS, FDD
	BinMatch ^[45]																													SimHash	JI
	Zeek ^[34]																													prov2vec	NNC
	FirmUP ^[37]																													strand	SGA
	α Diff ^[47]																													CNN	嵌入距离
	VulSeeker ^[48]																													DNN	cosine
	VulSeekerPro ^[46]																													DNN	cosine, JSC

表 7 二进制相似度分析技术比较 (续)

时间	方法	OS		处理器		编译器	优化	混淆	函数内联	补丁	分析对象		分析粒度			预处理工具				特征提取方法		特征表示					特征编码	相似度算法																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																					
		Windows	Linux	X86	ARM	MIPS					clang	gcc	程序	函数	基本块	IDA Pro	BAP	Angr	动态插桩	IR	逆向统计	动态提取	机器学习	I/O 对	符号表达式	strands			traces	图	嵌入																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																		
2019	InnerEye ^[54]	-	o	o	o	o	o	o	o	-	o	-	-	o	o	o	-	-	-	o	o	-	-	-	-	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o

注: JD: Jaccard distance, GES: global evidence score, SE: symbolic expressions, JIR: Jaccard index rates, FDD: function difference degree, NNC: neural network classifier, SGA: similarity game algorithm, JSC: Jaccard similarity coefficient, MD: Manhattan distance, LCS: longest common subsequence, RN: residual network, LCGS: local call graph similarity, SAN: self-attention network, TADW: text-associated DeepWalk, k-Hop GMA: k-Hop greedy matching algorithm, ED: Euclidean distance, SF: symbol formula
“o”表示是, “-”表示否, “A”表示全部(all), “D”表示缺省(default)

表 8 二进制相似度分析技术在嵌入式设备固件漏洞搜索中的应用

工具	数据集					CVE	查询样本开源库	受影响的设备固件	查询结果	时间开销	开源
	基线数据集(项目;编译器;优化选项;处理器;最后生成的数据集规模)	漏洞数	其他数据集	漏洞数	其他数据集						
Multi-MH ^[3]	OpenSSL:v1.0.1f, BusyBox:v1.21.1; gcc v4.6.2/v4.8.1, O0-O3; x86, MIPS, ARM; 60个二进制	8	无	无	无	2014-0160 2013-1813 SerComm后门 2013-6484	openssl-1.0.1f busybox-1.20.0 socket_read() Pidgin-2.10.8	DD-WRT NetgearNAS v6.1.6 Netgear Windows, macOS x	成功 成功 后门 成功	未明确	否
	OpenSSL:v1.0.1f, v1.0.1a, BusyBox:v1.21.1, v1.20, coreutils:v6.5, v6.7; gcc v4.6.2/v4.8.1, clang v3.0:O0-O3:x86, MIPS, ARM; 568 134个函数	8 128	154	无	无	2015-1791 2014-3508	基线数据库 基线数据库	D-Link, Belkin CenturyLink, D-Link, Actiontec	14/50 (28%) 24/50 (48%)	1.8 · 10 ⁻⁶ s	部分
Genius ^[5]	OpenSSL: v1.0.1f, 1.0.1u; gcc v5.4; O0-O3; x86, MIPS, ARM; 18 269个二进制文件, 129 365个ACFGs	8 128	154	IV-1	无	2015-1791 2014-3508	基线数据库 基线数据库	固件数据库 固件数据库	42/50 (84%) 41/50 (82%)	未明确	部分
Gemini ^[6]	OpenSSL:v1.0.1a, v1.0.2d, BusyBox:v1.19.0, 1.20.0;gcc v4.8.4/4.8.1, clang v3.4;缺省优化级别; x86, MIPS; 72 个二进制文件, 216 000个函数	1	10	无	无	2013-6449	基线数据库	DD-WRT(r21676)	87%	未明确	否
Xmatch ^[35]	OpenSSL: v1.0.1f, 1.0.1u, BusyBox:v1.21.0, coreutils:v6.5, v6.7;gcc:v4.9, v5.5; O0-O3;x86, x64, MIPS32, MIPS64, ARM32, ARM64; 240 个二进制, 735 540个函数	4 643	无	无	无	2015-1791 2014-3508	基线数据库 基线数据库	未知 未知	39/50 (78%) 41/50 (82%)	0.2s/函数 对	否
IoTSeeker ^[40]	BitDHT, GnuPG, ImageMagick, MAME, OpenCV, SQLite, stunnel; gcc 4.8.2, clang 3.3, ICC14.0.1, VC16.00; 排除函数内联; x86, x64, ARM v7, MIPS; 31 000个函数	3	无	无	无	2014-0160 2014-3566	openssl-1.0.1e openssl-1.0.1e	DD-WRT ReadyNAS	heartbleed POODLE	4.1 · 10 ⁻⁴ s	否
discoverRE ^[4]	BinARM ^[119]	2 628	235	无	无	漏洞库脆弱函数	无	固件数据集	精确度: 92%	0.00027 s	否
FirmUP ^[37]	无	~2 000	7	无	无	2011-0762 2009-4593 2012-0036 2013-1944 2013-2168 2014-4877 2016-8618	vsftpd bftpd libcurl libcurl dbus Wget 1.15 libcurl	Asus, Netgear Netgear Netgear Asus, D-Link D-Link, Netgear Asus, Netgear Asus, D-Link, Netgear	在固件库中发现了 373个脆弱函数, 成功率高达93%	未明确	否

表 8 二进制相似度分析技术在嵌入式设备固件漏洞搜索中的应用 (续)

工具	数据集					CVE	查询样本开源库	受影响的设备固件	查询结果	时间开销	开源
	基线数据集(项目;编译器;优化选项;处理器;最后生成的数据集规模)	漏洞数	其他数据集	固件(#)	数据集(#)						
FIT ^[51]	OpenSSL:v1.0.1f, v1.0.1u,					2016-5681	main()				
	BusyBox:v1.27.2, findutils:v4.6.0; gcc	IV-2		8 122	无	2013-1813	BusyBox	固件数据集中的	最高97.79%	3.61·10 ⁻³ s	部分
	v5.4;O0-O3;x86, MIPS, ARM;72个二进制	&IV-3				2014-3508	OpenSSL	固件		/1000基	
	文件, 254 391个3LACFGs					2017-15 873	BusyBox			本块	
Bin ^[50]	OpenSSL:v1.0.1e, v1.0.1u,			8 753	84	2015-1791	OpenSSL	D-Link, TP-Link,	42.6/50 (85.2%)	未明确	否
	BusyBox:v1.27.2; gcc v5.4; O0-O3;x86, AMD64, MIPS, MIPS64, ARM, ARM64;--		无			2014-3508	OpenSSL	Netgear, Cisco	87.9/100 (87.9%)		
Patchchecko ^[39]	100个Android库(来自Android 8.1.0 clang; O1, O2, O3, Oz, Ofast; x86, AMD64, ARM32, ARM64;2 108个库二进制, 2 037 772个函数	Android和 iOS固件	无	2 076		25个CVE	CVE脆弱函数	Andorid Things, Google Pixel	训练: 96%, 检测: 93%, 动态分析: ~100%	检测约3s/函数	否
Trex ^[59]						2019-1563	OpenSSL				
						2017-16544	BusyBox				
						2016-6303	OpenSSL				
						2016-6302	OpenSSL				
						2016-2842	OpenSSL				
						2016-2182	OpenSSL				
						2016-2180	OpenSSL				
	OpenSSL:v1.0.1f, v1.0.1u, binutils:2.34, coreutils:8.32, curl:7.71.1, diffutils:3.7, findutils: 4.7.0, GMP-6.2.0,					2016-2178	OpenSSL	Ubiquiti sunMaxTP-Link Deco-			
	ImageMagick:7.0.10, libmicrohttpd:0.9.71, libTomCrypt:1.18.2, PuTTY-0.74, SQLite-3.34.0, Zlib-1.2.11;gcc-7.5. O0-O3; x86, x64, ARM (32-bit), MIPS (32-bit);			180	16	2016-2176	OpenSSL	M4Netgear R7000Linksys RE7000	AUC>0.958	未明确	部分
	1 472 066个函数					2016-2109	OpenSSL				
						2016-2106	OpenSSL				
						2016-2105	OpenSSL				
						2016-0799	OpenSSL				
						2016-0798	OpenSSL				
						2016-0797	OpenSSL				
						2016-0705	OpenSSL				

在这 12 项工作中, Patcheko^[39]使用的开源项目最多, 它使用了 100 个 Android 库, 使用 clang 编译器, 5 种优化选项, 针对 4 种体系架构, 共编译生成了 2 037 772 个函数, 成为了最大规模的基线数据集. Trex^[59]次之, 它使用了 13 个开源项目, gcc 编译器, 4 种优化选项, 针对 4 种体系架构, 共编译生成了 1 472 066 个函数.

在固件数据集方面, BiN^[50]使用的固件数据集规模最大 (含 8 753 个固件), Genius^[5]和 Gemini^[6]次之 (含 8 128 个固件), FIT^[51]居第 3 (含 8 122 个固件). Xmatch^[35]、discovRE^[4]和 Multi-MH^[3]分别使用了 1、3 和 8 个固件来验证其工具的跨平台搜索能力, 其中, 三者均使用了 MIPS 架构的 DD-WRT 路由器固件 DD-WRT(r21676). 除了 DD-WRT (r21676) 之外, discovRE^[4]还使用了 ARM 架构的 NAS 设备 (Netgear ReadyNAS v6.1.6) 以及一个为 HTC Desire 编译的 Android ROM 镜像 (version 4.1.1) 来验证其在跨体系结构的漏洞搜索方面的能力. 除了 DD-WRT(r21676) 和 Netgear ReadyNAS v6.1.6 之外, Multi-MH^[3]还使用 MikroTik 的路由器 v5 和 v6 来验证 RouterOS 漏洞, 以及使用了 DGN1000, DGN3500, DM111Pv2, JNR3210 来验证 SerComm Backdoor 函数 socket_read().

在漏洞数据集方面, Patcheko^[39]收集到的漏洞数据集规模最大, 它收集了 2016 年 7 月到 2018 年 11 月期间来自 Android Security Bulletins (<https://source.android.com/security/bulletin>) 的漏洞, 共 2 076 个, 其中包含 1 351 个高危漏洞和 381 个关键漏洞. BinARM^[119]的漏洞数据集规模次之, 共 235 个, 它主要是针对 ARM 架构的 IED 固件收集而来的. Genius^[5]和 Gemini^[6]的漏洞数据集规模居第 3, 共 145 个. Multi-MH^[3]、discovRE^[4]、IoTSeeker^[49]和 FIT^[51]没有明确说明其所使用的漏洞数据集.

在用于搜索的 CVE 方面, 表 8 中的第 6 列、第 8 列和第 9 列分别列出了用于搜索的 CVE 编号、受影响的固件设备以及搜索查询的结果. 由于进行搜索的 CVE 不同、被搜索的固件数据集不同以及进行评估的准则不同等等因素, 很难对其进行一个综合评估. 因此, 本文仅列出了论文中所使用的 CVE 编号及其搜索的结果, 不对其进行详细的评估. 此外, 从第 6 列中, 还可以看出, 除了 Trex^[59]之外, 其余的工作使用的基本都是 2016 年之前分配的 CVE. Trex 使用 16 个 2016 年之后的 CVE 对一些最新版本的固件进行了漏洞搜索, 且其在实验中证明了其在跨体系结构搜索时的表现优于当前最先进的工具——SAFE^[56] 25.3%, 充分证实了 Trex 所提方法在应对大规模固件漏洞搜索时的实用性.

在时间开销方面, 由于其检测的粒度、数据集规模等方面存在差异, 很难使用一种统一的评估标准来评估不同的工具在时间开销方面的优劣度, 本文在表 8 中的第 10 列列出了各工具在其论文中明确指出的时间开销, 未明确指出的标注为“未明确”.

表 8 中的第 11 列列出了现有工作的开源情况, 其中只有 4 项工作 (4/12, 33.3%) 部分开放了其代码, 所谓“部分”开放其代码是指仅开源了其实现的源代码以及经过清洗后的数据集, 而没有给出其所使用的基线数据集、固件数据集以及漏洞数据集. 至目前为止, 尚没有一项工作全部开放了其所使用的数据集以及其进行数据清洗的相关代码.

6 总结与挑战

6.1 现有二进制相似度分析技术工作总结

本文通过对现有二进制相似度分析工作的总结与对比, 发现: 随着 AI 的发展, 基于机器学习的漏洞检测已经成为安全领域的研究热点, 特别是将二进制代码相似度分析技术应用在漏洞检测领域取得了一系列的研究成果, 主要体现在以下几个方面.

(1) 检测的粒度越来越细: 从早期的文件级, 到函数级, 再到基本块级;

(2) 特征表示越来越有效: 从早期的以文件的模糊哈希为特征表示, 到人工逆向统计出来的静态数据作为特征表示, 再到以控制流图或数据流图为代表的图表示, 到以深度学习模型的嵌入表示为目标的特征自动化学习, 使得特征表示越来越能表达目标代码所蕴含的更深层次的语义信息, 更能表征目标代码;

(3) 适用场景越来越具有普适性: 从初始的单一架构 (x86) 到跨架构 (x86, ARM, MIPS 等), 从单一特征到混合特征, 从仅应对一种编译选项, 到可以应对多种编译器、多种优化级别、混淆技术, 甚至跨版本, 使得二进制分析

工具的普适性越来越强,越来越能充分应对复杂多变的嵌入式设备固件环境;

(4) 方法选择越来越灵活: 静态分析无法衡量语义,对一些特征的变化比较敏感;动态执行复杂度高,可伸缩性差;深度学习的自动化程度高,但存在误报;近年来,逐渐出现了将多种方法混合在一起发挥其各自优势,从单一的静态分析,到静、动结合,到融合深度学习和动态分析,提升漏洞搜索的精确度;

(5) 机器学习新理论新技术的出现促进了漏洞检测的发展:从早期的支持向量机、逻辑回归等典型分类模型的应用,到 LSTM、NLP 等新型深度学习模型的应用,机器学习模型理论和技术不断发展,也在一步步地促进其在漏洞挖掘领域更具针对性且更精确地使用。

6.2 现有二进制相似度分析技术在嵌入式设备固件漏洞搜索领域仍然面临的一些挑战

二进制代码相似度分析技术在嵌入式设备固件漏洞搜索领域获取了一定的效果,从最初的文件相似度比较,到细粒度的函数比较,再到更细粒度的基本块级的比较,编码方案从最初的模糊哈希,到基于神经网络的图嵌入,到基于 NLP 的指令及基本块嵌入,漏洞关联速度和准确度都有了很大的提升。然而,现阶段二进制相似度分析技术在嵌入式设备固件漏洞搜索领域的发展并不十分成熟,还面临着以下挑战。

(1) 嵌入式设备固件逆向。固件逆向的准确度直接决定了二进制相似度匹配的结果,现有工作大多建立在“逆向能够获得精确的结果”这一假设基础之上,然而,事实上,固件逆向是一项耗时且极具挑战性的任务,并且通常需要专家知识。完全精确且完备的二进制固件逆向几乎是不可能实现的^[130]。

(2) 嵌入式设备固件二进制代码的存储及高效查询。嵌入式设备固件种类繁多,每一个固件中提取出来的二进制文件几乎都有数百个,每一个二进制文件可以提取出数百甚至上千个函数,因此固件数据集的存储和查询直接影响到其查询的性能,目前尚没有工作提及对固件数据集的存储和查询的优化。

(3) 针对嵌入式设备固件的漏洞数据集的构建。除了 BinARM^[119]提供了一个 ARM IED 固件的漏洞数据集之外,现有工作使用的多是应用程序级的特定 CVE,对于每一个新的 CVE,都需要做一次重新编译、特征提取、特征表示、相似度计算等工作,甚至还需要去重新训练模型,因此,可扩展性不强。

(4) 无法应对跨函数/跨二进制文件的漏洞检测。目前的解决方案仅限于单个函数之间的相似度比对为依据的漏洞搜索。然而,由于嵌入式设备固件的特有性,其漏洞经常是跨函数,甚至是跨二进制文件的,对于这类漏洞的搜索仍然是嵌入式设备固件漏洞搜索急需解决的难题,像 Karnote^[131]这样的跨二进制的静态分析方法可以提供一些有价值的参考;

(5) 误报或漏报问题是固有问题。虽然 Patchcko^[39]和 IoTSeeker^[49]等引入了动态特征对第一阶段产生的候选做进一步的筛选,一定程度上减少了误报的概率,但由于其本质上并没有实际执行目标代码,因此,误报或漏报的问题仍然存在。

(6) 缺乏统一的评估标准。从表 7 和表 8 中可以看出,在二进制相似度分析领域,目前还不存在任何两个工作在所使用的数据集、作为搜索样本的 CVE 漏洞、评估标准等完全相同的。而大多数工作其数据集或源码是不开放的,因此,很难执行横向评估。

6.3 未来的一些开放性的研究方向

正如第 6.2 节所述,现阶段基于二进制相似度分析技术的嵌入式设备固件漏洞搜索尚处于初步发展阶段,仍然存在很多挑战,但是新的挑战也会产生新的机遇。本文提出了一些开放性的研究方向。

(1) 提升二进制相似度分析技术应对不同配置的能力

虽然现有的二进制相似度分析技术在跨体系、跨编译器和跨优化选项方面都有所考虑且取得了显著的成果(表 7),然而,本文认为二进制代码相似度分析技术的研究仍然需要进一步应对函数内联、混淆、补丁存在性、跨项目等不同配置的挑战。同时,随着桌面应用程序、嵌入式设备以及它们之间的互连性急剧增长,部署的软件数量也在呈指数级的增长,大规模的漏洞检测是绝对必要的。因此,所提方法应该能够在代价最小的情况下扩展应用到新加入的软件。引入预先过滤机制(首先过滤掉那些极其不相似的函数)来最小化搜索空间,以更有效地静态识别漏洞,并且在动态分析的情况下提供更好的代码覆盖率。

(2) 提升二进制逆向的准确度

如第 4.2 节所述, 大多数的二进制代码相似度分析工具严重依赖于二进制逆向工具 (比如 IDA Pro/Angr). 然而, 即使是 IDA Pro 这种相对成熟的反汇编工具也并不完美^[130], 它在应对二进制逆向特别是二进制固件逆向时依然极具挑战, 主要体现在以下几个方面.

- 入口点和函数边界识别. 反汇编器通常使用符号表来识别函数边界以及构建控制流图. 然而, 当符号表不准确或者不可用时, 寻找函数边界就变得很有挑战^[132]. 特别是在二进制固件^[133]中, 入口点和基地址是未知的. 此外, 有些函数可能有多个入口点^[134], 需要进一步识别.

- 代码混淆/变换. 合法/良性程序的作者可能会出于不同的原因保护他们的程序, 比如知识产权保护或者防止其程序被重新打包或作为恶意软件发布. 同样, 恶意软件作者会在其恶意代码上应用某些技术以逃避分析. 用于这些目的的混淆^[135], 加密和打包^[136]技术使得二进制分析更具挑战性且更困难. 现有的分析方法只有 20% 考虑了代码混淆, 且仅考虑了 obfuscator-llvm 形式的混淆.

- 应对编译选项的特殊配置. 比如, 在处理 clang 加上 PIC 选项编译而来的 MIPS 二进制文件时, IDA Pro 是无法分析间接分支的. 特别地, 编译器使用寄存器-间接分支指令 (比如 jalr) 以一种独立于位置的方式调用函数. 再比如, 当调用一个函数时, gcc 会在 gp 寄存器中存储 GOT 的基地址, 并且在运行时使用它来计算函数地址, 而 clang 则使用 s0 或 v0 寄存器来存储这样的基地址. 这种微小的差异会混淆 IDA Pro, 使其不能获取 GOT 的基地址, 因此不能计算间接分支的目标地址. 系统发现这样的错误是一项困难的任务, 因为很多逆向工具的内部细节并没有完全公开, 而且它们之间的差异很大.

(3) 构建更符合嵌入式设备固件漏洞场景的搜索模型

现有研究工作主要集中在根据第三方库中暴出来的已知漏洞, 提取出其语义特征, 与存储在固件数据集中的语义特征进行一对一的比对, 选择出最接近于已知漏洞语义特征的候选对象作为可疑对象, 然而本文认为还可以在以下几个方面对这个流程进行改进.

- 深度学习模型的选择. 现有研究工作大多使用有监督的学习方法, 数据集的标记需要很大的工作量, 且需要考虑训练样本的平衡问题. 文献 [137] 指出: 半监督学习利用少量的已标记样本和大量的未标记样本, 非常符合漏洞检测的应用场景. 未来的研究工作, 在深度学习模型的选择方面, 可以考虑使用半监督学习和强化学习等模型进行更多的尝试.

- 漏洞的形成机理分析. 现有研究工作大多仅考虑单个函数内发生漏洞的情况, 并没有对漏洞的形成机理进行深入分析, 提出类似于漏洞模板一样的查询函数, 因此, 其在实际应用于, 只能做一对一的搜索, 可扩展性不强. 且无法应对跨函数或跨二进制文件的漏洞. 未来的研究工作中, 对漏洞的形成机理进行深入分析, 构建能够精确表达漏洞代码业务流程的代码特征也必将是一个重要的研究方向.

- 泛化漏洞签名. 正如上面所述, 现有的大多数分析方法都为每一个漏洞函数以不同的表示和特定级别提供了一种特定模式, 并且利用匹配技术或相似性度量来识别它. 为每一种类型的漏洞 (比如, 缓冲区溢出) 提供一个通用签名而不是与已经具有特定漏洞的函数相匹配是未来的研究方向之一.

(4) 构建统一且规范的数据集集合

为了更好地推进二进制相似度分析技术及其在嵌入式设备固件漏洞搜索领域的应用, 本文认为需要构建统一且规范的数据集集合, 包括: 用于训练的二进制数据集, 用于评估的真实固件二进制数据集, 以及用于搜索查询的漏洞数据库 (含固件漏洞) 以及相应的补丁数据库, 以解决:

- 代码库的多样性问题. 代码库决定了分析工具的搜索空间, 代码库的特征决定了分析工具可以在搜索空间中发现什么. 现有大多数工作以不同的方式构建其代码基, 这可能会影响到其工具的性能和可用性. 实际环境中的代码库规模 (单个固件镜像) 通常有数百万行代码, 然而, 许多现有的工具只在小规模的代码库中测试他们的工具, 因此, 其结果可能不能直接推广到大规模固件镜像组成的数据集.

- 搜索查询的数量问题. 表 7 列出了近 50% 的工具仅使用不超过 3 个的 CVE 漏洞作为其查询样本来评估其工具在漏洞查询上的表现, 这样的查询规模只能涵盖软件开发人员实际使用的有限数量的真实查询. 分析发现, 阻

碍查询规模的主要原因在于工具评估方法, 现有研究表明, 目前约 100% 的工具需要通过手工评估来进行验证, 增加查询规模将会大大增加在样本标记期间的负担. 因此, 构建一个统一的、规范的、涵盖各种类型漏洞 (包括固件漏洞) 的开源漏洞数据集, 也将成为未来的一个研究方向.

- 工具的有效性和可扩展性评估问题. 从表 7 和表 8 中可以看出, 在二进制相似度分析及其在嵌入式设备固件漏洞中的搜索领域, 目前尚不存在任何两个工作在其训练所使用的数据集、作为搜索样本的 CVE 漏洞、评估标准等方面是完全相同的, 而且现有的大多数工作其数据集或源码都不是公开可访问的. 因此, 无法对工具的有效性和可扩展性进行评估.

(5) 构建集静态分析、动态验证和深度学习于一体的嵌入式设备固件综合分析模型

如第 1.1 节所示, 针对嵌入式设备及其固件的测试, 近年来, 学术界和工业界提出了很多不同的技术, 然而, 这些技术大多数都仅适用于特定的设置和环境, 比如 FirmAFL^[12]、FirmFuzz^[13]仅适用于 Firmadyne^[11]可以仿真起来的设备固件. 为了应对大规模嵌入式设备固件漏洞搜索的需求, 本文认为: 融合不同的研究方法, 构建一个集静态分析、动态验证和深度学习于一体的综合分析模型, 共同测试嵌入式设备固件仍然是一个开放性的研究课题.

- 自动化的重新托管. 目前在嵌入式设备固件领域可用的重新托管技术^[21,24]有助于集成表现良好的二进制分析方法 (比如灰盒测试或符号执行) 到嵌入式设备固件测试环境中. 然而, 针对嵌入式设备固件提供的仿真解决方案仍处于起步阶段, 为嵌入式设备固件构建部分或完全仿真环境是一项艰巨而复杂的任务. 此外, 重新托管仍然需要手动调整, 以适用于不同的 CPU 架构. Pretender^[24]是自动化的重新托管以进行可扩展的二进制固件分析的初步尝试. 自动化地捕获/学习原始设备的输入/输出, 并且通过机器学习算法聚类/复制这些输入/输出, 也将是未来满足对仿真组件和外设的有效输入/输出需求的一个有效途径.

- 面向嵌入式设备固件的 (增强的) 模糊测试方法. 尽管重新托管技术确实可以帮助模糊测试技术应用于嵌入式设备固件的测试, 但目前为止, 并没有更好的针对嵌入式设备固件漏洞的模糊测试方法提出 (比如测试例生成、动态插桩和错误监测等). 这是因为大多数的嵌入式设备固件并没有实现内存管理, 即便发生了内存崩溃, 其错误也不会报告出来. 因此, 构建一个仿真器, 能够报告内存破坏, 并且可以在实现的模糊测试方法上提供反馈, 将会为分析不同的模糊测试方法在嵌入式设备上的效果提供机遇.

- 融合不同的方法到嵌入式设备固件的测试. 尽管目前提出了很多不同的方法用于嵌入式设备固件的测试, 但融合不同的方法仍然极具挑战但可以进一步研究. 比如以二进制相似度分析报告出来的脆弱点作为模糊测试的“兴趣点”, 将模糊测试导向到脆弱点做进一步的执行测试, 以确定漏洞以及触发漏洞的输入. 比如, 使用符号执行来求解需要什么样的外设值才可以使模糊测试继续执行. 比如, 使用深度学习为测试生成针对性的输入样本等.

References:

- [1] Cui A, Costello M, Stolfo SJ. When firmware modifications attack: A case study of embedded exploitation. In: Proc. of the 20th Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2013.
- [2] Antonakakis M, April T, Bailey M, Bernhard M, Bursztein E, Cochran J, Durumeric Z, Halderman JA, Invernizzi L, Kallitsis M, Kumar D, Lever C, Ma Z, Mason J, Menscher D, Seaman C, Sullivan N, Thomas K, Zhou Y. Understanding the mirai botnet. In: Proc. of the 26th USENIX Security Symp., USENIX Security 2017. Vancouver: USENIX Association, 2017. 1093–1110.
- [3] Pewny J, Garmany B, Gawlik R, Rossow C, Holz T. Cross-architecture bug search in binary executables. In: Proc. of the 2015 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2015. 709–724. [doi: 10.1109/SP.2015.49]
- [4] Eschweiler S, Yakdan K, Gerhards-Padilla E. discovRE: Efficient cross-architecture identification of bugs in binary code. In: Proc. of the 23rd Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2016. 58–79.
- [5] Feng Q, Zhou RD, Xu CC, Cheng Y, Testa B, Yin H. Scalable graph-based bug search for firmware images. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: Association for Computing Machinery, 2016. 480–491. [doi: 10.1145/2976749.2978370]
- [6] Xu XJ, Liu C, Feng Q, Yin H, Song L, Song D. Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: Association for Computing Machinery, 2017. 363–376. [doi: 10.1145/3133956.3134018]

- [7] Ul Haq I, Caballero J. A survey of binary code similarity. arXiv:1909.11424, 2019.
- [8] Muench M, Stijohann J, Kargl F, Francillon A, Balzarotti D. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In: Proc. of the 25th Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2018. 30–43.
- [9] Costin A, Zaddach J, Francillon A, Balzarotti D. A large-scale analysis of the security of embedded firmwares. In: Proc. of the 23rd USENIX Conf. on Security Symp. San Diego: USENIX Association, 2014. 95–110.
- [10] Costin A, Zarras A, Francillon A. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In: Proc. of the 11th ACM on Asia Conf. on Computer and Communications Security. Xi'an: Association for Computing Machinery, 2016. 437–448. [doi: 10.1145/2897845.2897900]
- [11] Chen DD, Egele M, Woo M, Brumley D. Towards automated dynamic analysis for linux-based embedded firmware. In: Proc. of the 23rd Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2016. 150–167. [doi: 10.14722/ndss.2016.23415]
- [12] Zheng YW, Davanian A, Yin H, Song CY, Zhu HS, Sun LM. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In: Proc. of the 28th USENIX Conf. on Security Symp. Santa Clara: USENIX Association, 2019. 1099–1114.
- [13] Srivastava P, Peng H, Li JH, Okhravi HR, Shrobe HE, Payer M. FirmFuzz: Automated IoT Firmware Introspection and Analysis. In: Proc. of the 2nd Int'l ACM Workshop on Security and Privacy for the Internet-of-Things. London: Association for Computing Machinery, 2019. 15–21. [doi: 10.1145/3338507.3358616]
- [14] Zaddach J, Bruno L, Francillon A, Balzarotti D. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: Proc. of the 21st Network and Distributed System Security Symp. San Diego: The Internet Society, 2014. 1–16. [doi: 10.14722/ndss.2014.23229]
- [15] Muench M, Nisi D, Francillon A, Balzarotti D. Avatar²: A multi-target orchestration platform. In: Proc. of the Workshop on Binary Analysis Research. Cham: Springer, 2018. 1–11.
- [16] Kammerstetter M, Platzer C, Kastner W. Prospect: Peripheral proxying supported embedded code testing. In: Proc. of the 9th ACM Symp. on Information, Computer and Communications Security. Kyoto: Association for Computing Machinery, 2014. 329–340. [doi: 10.1145/2590296.2590301]
- [17] Koscher K, Kohn T, Molnar D. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In: Proc. of the 9th USENIX Workshop on Offensive Technologies. Washington: USENIX Association, 2015. 67–80.
- [18] Kammerstetter M, Burian D, Kastner W. Embedded security testing with peripheral device caching and runtime program state approximation. In: Proc. of the 10th Int'l Conf. on Emerging Security Information, Systems and Technologies (SECUREWARE). Venice, 2016. 339–345.
- [19] Talebi SMS, Tavakoli H, Zhang H, Zhang Z, Sani AA, Qian ZY. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In: Proc. of the 27th USENIX Conf. on Security Symp. Baltimore: USENIX Association, 2018. 291–307.
- [20] Corteggiani N, Camurati G, Francillon A. Inception: System-wide security testing of real-world embedded systems software. In: Proc. of the 27th USENIX Conf. on Security Symp. Baltimore: USENIX Association, 2018. 309–326.
- [21] Feng B, Mera A, Lu L. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: Proc. of the 29th USENIX Conf. on Security Symp. USENIX Association, 2020. 70.
- [22] Clements A A, Gustafson E, Scharnowski T, Grosen P, Fritz D, Kruegel C, Vigna G, Bagchi S, Payer M. HALucinator: Firmware re-hosting through abstraction layer emulation. In: Proc. of the 29th USENIX Security Symp. USENIX Association, 2020. 1–18.
- [23] Harrison L, Vijayakumar H, Padhye R, Sen K, Grace MC. Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In: Proc. of the 29th USENIX Conf. on Security Symp. USENIX Association, 2020. 789–806.
- [24] Gustafson E, Muench M, Spensky C, Redini N, Machiry A, Fratantonio Y, Balzarotti D, Francillon A, Choe YR, Kruegel C, Vigna G. Toward the analysis of embedded firmware through automated Re-hosting. In: Proc. of the 22nd Int'l Symp. on Research in Attacks, Intrusions and Defenses. Beijing: USENIX Association, 2019. 135–150.
- [25] Cao C, Guan L, Ming J, Liu P. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In: Proc. of the Annual Computer Security Applications Conf. Austin: Association for Computing Machinery, 2020. 746–759. [doi: 10.1145/3427228.3427280]
- [26] Mera A, Feng B, Lu L, *et al.* DICE: Automatic emulation of DMA input channels for dynamic firmware analysis. In: Proc. of the 2021 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2021. 1938–1954. [doi: 10.1109/SP40001.2021.00018]
- [27] Li Z, Zou DQ, Xu SH, Ou XY, Jin H, Wang SJ, Deng ZJ, Zhong YY. Vuldeepecker: A deep learning-based system for vulnerability detection. In: Proc. of the 25th Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2018. 1–15.

- [28] Li Z, Zou DQ, Xu SH, Jin H, Zhu YW, Chen ZX. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. on Dependable and Secure Computing*, 2021; 1. [doi: 10.1109/TDSC.2021.3051525]
- [29] Wang HT, Ye GX, Tang ZY, Tan SH, Haung SF, Fang DY, Feng YS, Bian L Z, Wang Z. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Trans. on Information Forensics and Security*, 2021, 16: 1943–1958. [doi: 10.1109/TIFS.2020.3044773]
- [30] Zou DQ, Wang SJ, Xu SH, Li Z, Jin H. μ VulDeePecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Trans. on Dependable and Secure Computing*, 2021, 18(5): 2224–2236. [doi: 10.1109/TDSC.2019.2942930]
- [31] Ghafoor I, Jattala I, Durrani S, Tahir CM. Analysis of OpenSSL heartbleed vulnerability for embedded systems. In: *Proc. of the 17th IEEE Int'l Multi Topic Conf. 2014. Karachi: IEEE, 2014. 314–319. [doi: 10.1109/INMIC.2014.7097358]*
- [32] David Y, Partush N, Yahav E. Statistical similarity of binaries. *ACM Sigplan Notices*, 2016, 51(6): 266–280. [doi: 10.1145/2980983.2908126]
- [33] David Y, Partush N, Yahav E. Similarity of binaries through re-optimization. In: *Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Barcelona: Association for Computing Machinery, 2017. 79–94. [doi: 10.1145/3062341.3062387]*
- [34] Shalev N, Partush N. Binary similarity detection using machine learning. In: *Proc. of the 13th Workshop on Programming Languages and Analysis for Security. Toronto: Association for Computing Machinery, 2018. 42–47. [doi: 10.1145/3264820.3264821]*
- [35] Feng Q, Wang MH, Zhang M, Zhou RD, Henderson A, Yin H. Extracting conditional formulas for cross-platform bug search. In: *Proc. of the 2017 ACM on Asia Conf. on Computer and Communications Security. 2017. 346–359. [doi: 10.1145/3052973.3052995]*
- [36] Chandramohan M, Xue YX, Xu ZZ, Liu Y, Cho CY, Tan HKB. Bingo: Cross-architecture cross-OS binary search. In: *Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: Association for Computing Machinery, 2016. 678–689. [doi: 10.1145/2950290.2950350]*
- [37] David Y, Partush N, Yahav E. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices*, 2018, 53(2): 392–404. [doi: 10.1145/3296957.3177157]
- [38] Wang S, Wu DH. In-memory fuzzing for binary code similarity analysis. In: *Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Urbana: IEEE, 2017. 319–330. [doi: 10.1109/ASE.2017.8115645]*
- [39] Sun PF, Garcia L, Salles-Loustau G, Zonouz S. Hybrid firmware analysis for known mobile and IoT security vulnerabilities. In: *Proc. of the 50th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). Valencia: IEEE, 2020. 373–384. [doi: 10.1109/DSN48063.2020.00053]*
- [40] Hu YK, Zhang YY, Li JR, Gu DW. Cross-architecture binary semantics understanding via similar code comparison. In: *Proc. of the 23rd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). Osaka: IEEE, 2016. 57–67. [doi: 10.1109/SANER.2016.50]*
- [41] Luo LN, Ming J, Wu DH, Liu P, Zhu SC. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Trans. on Software Engineering*, 2017, 43(12): 1157–1177. [doi: 10.1109/TSE.2017.2655046]
- [42] Ming J, Xu DP, Jiang YF, Wu DH. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In: *Proc. of the 26th USENIX Conf. on Security Symp. Vancouver: USENIX Association, 2017. 253–270.*
- [43] Xue YX, Xu ZZ, Chandramohan M, Liu Y. Accurate and scalable cross-architecture cross-OS binary code search with emulation. *IEEE Trans. on Software Engineering*, 2018, 45(11): 1125–1149. [doi: 10.1109/TSE.2018.2827379]
- [44] Hu YK, Zhang YY, Li JR, Gu DW. Binary code clone detection across architectures and compiling configurations. In: *Proc. of the 25th IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). Buenos Aires: IEEE, 2017. 88–98. [doi: 10.1109/ICPC.2017.22]*
- [45] Hu YK, Zhang YY, Li JR, Wang H, Li BD, Gu DW. Binmatch: A semantics-based hybrid approach on binary code clone analysis. In: *Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Madrid: IEEE, 2018. 104–114. [doi: 10.1109/ICSME.2018.00019]*
- [46] Gao J, Yang X, Fu Y, Jiang Y, Shi HY, Sun JG. Vulseeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation. In: *Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena Vista: Association for Computing Machinery, 2018. 803–808. [doi: 10.1145/3236024.3275524]*
- [47] Liu BC, Huo W, Zhang C, Li WC, Li F, Piao AH, Zou W. α Diff: Cross-version binary code similarity detection with DNN. In: *Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. Montpellier: Association for Computing Machinery, 2018. 667–678. [doi: 10.1145/3238147.3238199]*
- [48] Gao J, Yang X, Fu Y, Jiang Y, Sun JG. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In: *Proc. of*

- the 33rd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Montpellier: IEEE, 2018. 896–899. [doi: 10.1145/3238147.3240480]
- [49] Gao J, Yang X, Jiang Y, Song HB, Choo KKR, Sun JG. Semantic learning based cross-platform binary vulnerability search for IoT devices. *IEEE Trans. on Industrial Informatics*, 2019, 17(2): 971–979. [doi: 10.1109/TII.2019.2947432]
 - [50] Wu H, Shu H, Kang F, Xiong XB, BiN: A two-level learning-based bug search for cross-architecture binary. *IEEE Access*, 2019, 7: 169548–169564. [doi: 10.1109/ACCESS.2019.2953173]
 - [51] Liang HL, Xie ZS, Chen YX, Ning H, Wang JL. FIT: Inspect vulnerabilities in cross-architecture firmware by deep learning and bipartite matching. *Computers & Security*, 2020, 99: 102032. [doi: 10.1016/j.cose.2020.102032]
 - [52] Kim D, Kim E, Cha SK, Son S, Kim Y. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *arXiv:2011.10749*, 2020.
 - [53] Ding SHH, Fung BCM, Charland P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: *Proc. of the 2019 IEEE Symp. on Security and Privacy (SP)*. San Francisco: IEEE, 2019. 472–489. [doi: 10.1109/SP.2019.00003]
 - [54] Zuo F, Li XP, Young P, Luo LN, Zeng Q, Zhang ZX. Neural machine translation inspired binary code similarity comparison beyond function Pairs. *arXiv:1808.04706*, 2018. [doi: 10.14722/ndss.2019.23492]
 - [55] Redmond K, Luo LN, Zeng Q. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. In: *Proc. of the 2019 Workshop on Binary Analysis Research (BAR)*. San Diego: NDSS, 2019. 1–8. [doi: 10.14722/bar.2019.23057]
 - [56] Massarelli L, Di Luna GA, Petroni F, *et al.* Safe: Self-attentive function embeddings for binary similarity. In: *Proc. of the 16th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*. Gothenburg: Springer, 2019. 309–329. [doi: 10.1007/978-3-030-22038-9_15]
 - [57] Zhang XC, Sun WJ, Pang JM, Liu FD, Ma Z. Similarity metric method for binary basic blocks of cross-instruction set architecture. In: *Proc. of the 2020 Workshop on Binary Analysis Research*. San Diego: NDSS, 2020. 23–26. [doi: 10.14722/bar.2020.23002]
 - [58] Yu ZP, Cao R, Tang QY, Nie S, Huang JZ, Wu S. Order matters: Semantic-aware neural networks for binary code similarity detection. *Proc. of the AAAI Conf. on Artificial Intelligence*, 2020, 34(1): 1145–1152. [doi: 10.1609/aaai.v34i01.5466]
 - [59] Pei KX, Xuan Z, Yang JF, Jana S, Ray B. TREX: Learning execution semantics from micro-traces for Binary similarity. *arXiv:2012.08680*, 2020.
 - [60] Duan Y, Li X, Wang JH, Yin H. DeepBinDiff: Learning program-wide code representations for binary diffing. In: *Proc. of the 2020 Network and Distributed Systems Security (NDSS) Symp.* San Diego: NDSS, 2020. 1–16. [doi: 10.14722/ndss.2020.24311]
 - [61] Bourquin M, King A, Robbins E. Binslayer: Accurate comparison of binary executables. In: *Proc. of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. Rome: Association for Computing Machinery, 2013. 4. [doi: 10.1145/2430553.2430557]
 - [62] Kuhn HW. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 1955, 2(1–2): 83–97. [doi: 10.1002/nav.3800020109]
 - [63] David Y, Yahav E. Tracelet-based code search in executables. *ACM Sigplan Notices*, 2014, 49(6): 349–360. [doi: 10.1145/2666356.2594343]
 - [64] Egele M, Woo M, Chapman P, Brumley D. Blanket execution: Dynamic similarity testing for program binaries and components. In: *Proc. of the 23rd USENIX Conf. on Security Symp.* San Diego: USENIX Association, 2014. 303–317.
 - [65] Gao DB, Reiter MK, Song D. BinHunt: Automatically finding semantic differences in binary programs. In: *Proc. of the 10th Int'l Conf. on Information and Communications Security*. Birmingham: Springer, 2008. 238–255. [doi: 10.1007/978-3-540-88625-9_16]
 - [66] Ming J, Pan M, Gao DB. iBinHunt: Binary hunting with inter-procedural control flow. In: *Proc. of the 15th Int'l Conf. on Information Security and Cryptology*. Seoul: Springer, 2012. 92–109. [doi: 10.1007/978-3-642-37682-5_8]
 - [67] Song L. Structure2Vec: Deep Learning for Security Analytics Over Graphs. Atlanta, GA: USENIX Association, 2018.
 - [68] Dai H, Dai B, Song L. Discriminative embeddings of latent variable models for structured data. In: *Proc. of the 33rd Int'l Conf. on Machine Learning*. New York: JMLR.org, 2016. 2702–2711.
 - [69] He AF, Luo C, Tian XM, Zeng WJ. A twofold Siamese network for real-time object tracking. In: *Proc. of the 2018 IEEE/CVF Conf. on Computer Vision and Pattern Recognition*. Salt Lake City: IEEE, 2018. 4834–4843. [doi: 10.1109/CVPR.2018.00508]
 - [70] Brumley D, Jager I, Avgerinos T, Schwartz E J. BAP: A binary analysis platform. In: *Proc. of the 23rd Int'l Conf. on Computer Aided Verification*. Snowbird: Springer, 2011. 463–469. [doi: 10.1007/978-3-642-22110-1_37]
 - [71] Wang F, Shoshitaishvili Y. Angr—The next generation of binary analysis. In: *Proc. of the 2017 IEEE Cybersecurity Development*

- (SecDev). Cambridge: IEEE, 2017. 8–9. [doi: 10.1109/SecDev.2017.14]
- [72] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 2007, 42(6): 89–100. [doi: 10.1145/1273442.1250746]
 - [73] Kosub S. A note on the triangle inequality for the Jaccard distance. *Pattern Recognition Letters*, 2019, 120: 36–38. [doi: 10.1016/j.patrec.2018.12.007]
 - [74] Gao XB, Xiao B, Tao DC, Li XL. A survey of graph edit distance. *Pattern Analysis and Applications*, 2010, 13(1): 113–129. [doi: 10.1007/s10044-008-0141-y]
 - [75] Rakamarić Z, Emmi M. SMACK: Decoupling source language details from verifier implementations. In: *Proc. of the 26th Int'l Conf. on Computer Aided Verification*. Vienna: Springer, 2014. 106–113. [doi: 10.1007/978-3-319-08867-9_7]
 - [76] Jiang JG, Li GW, Yu M, Li G, Liu C, Lv ZQ, Lv B, Huang WQ. Similarity of binaries across optimization levels and obfuscation. In: *Proc. of the 25th European Symp. on Research in Computer Security*. Guildford: Springer, 2020. 295–315. [doi: 10.1007/978-3-030-58951-6_15]
 - [77] Dinaburg A, Ruef A. Mcsema: Static translation of x86 instructions to LLVM. In: *Proc. of the ReCon 2014 Conf.* Montreal, 2014.
 - [78] Song D, Brumley D, Yin H, Caballero J, Jager I, Kang GY, Liang ZK, Newsome J, Poosankam P, Saxena P. BitBlaze: A new approach to computer security via binary analysis. In: *Proc. of the 4th Int'l Conf. on Information Systems Security*. Hyderabad: Springer, 2008. 1–25. [doi: 10.1007/978-3-540-89862-7_1]
 - [79] Appel AW. SSA is functional programming. *ACM SIGPLAN Notices*, 1998, 33(4): 17–20. [doi: 10.1145/278283.278285]
 - [80] Lin H, Zhao DD, Ran LJ, Han MS, Tian J, Xiang JW, Ma X, Zhong YS. Cvssa: Cross-architecture vulnerability search in firmware based on support vector machine and attributed control flow graph. In: *Proc. of the 2017 Int'l Conf. on Dependable Systems and Their Applications (DSA)*. Beijing: IEEE, 2017. 35–41. [doi: 10.1109/DSA.2017.15]
 - [81] Devlin J, Chang MW, Lee K, Toutanova K. Bert: Pre-training of deep bidirectional transformers for language understanding. In: *Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis: Association for Computational Linguistics, 2018. 4171–4186. [doi: 10.18653/v1/N19-1423]
 - [82] De Moura L, Bjørner N. Z3: An efficient SMT solver. In: *Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Budapest: Springer, 2008. 337–340. [doi: 10.1007/978-3-540-78800-3_24]
 - [83] Lawler GF, Limic V. *Random Walk: A Modern Introduction*. Cambridge: Cambridge University Press, 2010.
 - [84] Xu YF, Xu ZZ, Chen BH, Song F, Liu Y, Liu T. Patch based vulnerability matching for binary programs. In: *Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*. Association for Computing Machinery, 2020. 376–387. [doi: 10.1145/3395363.3397361]
 - [85] Godefroid P. Micro execution. In: *Proc. of the 36th Int'l Conf. on Software Engineering*. Hyderabad: Association for Computing Machinery, 2014. 539–549. [doi: 10.1145/2568225.2568273]
 - [86] Hagberg AA, Schult DA, Swart PJ. Exploring network structure, dynamics, and function using NetworkX. In: *Proc. of the 7th Python in Science Conf.* 2008. 11–16.
 - [87] Quynh NA, Vu DH. Unicorn: Next generation CPU emulator framework. *BlackHat USA*, 2015: 476.
 - [88] Church KW. Word2Vec. *Natural Language Engineering*, 2017, 23(1): 155–162. [doi: 10.1017/S1351324916000334]
 - [89] Li YD, Hao ZB, Lei H. Survey of convolutional neural network. *Journal of Computer Applications*, 2016, 36(9): 2515–2516.
 - [90] Young T, Hazarika D, Poria S, Cambria E. Recent trends in deep learning based natural language processing. *IEEE Computational Intelligence Magazine*, 2018, 13(3): 55–75. [doi: 10.1109/MCI.2018.2840738]
 - [91] Zhu XD, Sobihani P, Guo HY. Long short-term memory over recursive structures. In: *Proc. of the 32nd Int'l Conf. on Machine Learning*. Lille: JMLR.org, 2015. 1604–1612.
 - [92] Le Q, Mikolov T. Distributed representations of sentences and documents. In: *Proc. of the 31st Int'l Conf. on Int'l Conf. on Machine Learning*. Beijing: JMLR.org, 2014. 1188–1196.
 - [93] Lin ZH, Feng MW, Dos Santos CN, Yu M, Xiang B, Zhou BW, Bengio Y. A structured self-attentive sentence embedding. *arXiv:1703.03130*, 2017.
 - [94] Director, Research. *Practical graph isomorphism*. *Computation Theory & Mathematics*, 2014. 110–117.
 - [95] Marimont RB, Shapiro MB. Nearest neighbour searches and the curse of dimensionality. *IMA Journal of Applied Mathematics*, 1979, 24(1): 59–70. [doi: 10.1093/imamat/24.1.59]
 - [96] Dai B, Shen XT, Wang JH. Embedding learning. *Journal of the American Statistical Association*, 2020: 1–13. [doi: 10.1080/01621459.2020.1775614]

- [97] Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. In: Proc. of the 2013 Int'l Conf. on Learning Representations. Scottsdale: ICLR, 2013. 150–169.
- [98] Kenter T, Borisov A, De Rijke M. Siamese CBOW: Optimizing word embeddings for sentence representations. arXiv:1606.04640, 2016.
- [99] Luong MT, Pham H, Manning CD. Bilingual word representations with monolingual quality in mind. In: Proc. of the Workshop on Vector Space Modeling for Natural Language Processing. Denver: Association for Computational Linguistics, 2015. 151–159. [doi: 10.3115/v1/W15-1521]
- [100] Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J. Distributed representations of words and phrases and their compositionality. In: Proc. of the 26th Int'l Conf. on Neural Information Processing Systems. Lake Tahoe: Association for Computing Machinery, 2013. 3111–3119.
- [101] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin L. Attention is all you need. arXiv:1706.0362, 2017.
- [102] Cho K, Van Merriënboer B, Bahdanau D, Bengio Y. On the properties of neural machine translation: Encoder-decoder approaches. In: Proc. of the 8th Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST-8). Doha: Association for Computational Linguistics, 2014. 103–111.
- [103] Gilmer J, Schoenholz SS, Riley PF, Vinyals O, Dahl GE. Neural message passing for quantum chemistry. In: Proc. of the 34th Int'l Conf. on Machine Learning. Sydney: JMLR.org, 2017. 1263–1272.
- [104] He KM, Zhang XY, Ren SQ, Sun J. Deep residual learning for image recognition. In: Proc. of the 2016 IEEE Conf. on Computer Vision and Pattern Recognition. Las Vegas: IEEE, 2016. 770–778. [doi: 10.1109/CVPR.2016.90]
- [105] Ye ZL, Zhao HX, Zhang K, Zhu Y, Xiao YZ. Text-associated max-margin DeepWalk. In: Proc. of the 6th CCF Conf. on Big Data. Xi'an: Springer, 2018. 301–321. [doi: 10.1007/978-981-13-2922-7_21]
- [106] Fernandez RC, Min J, Nava D, Madden S. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In: Proc. of the 35th IEEE Int'l Conf. on Data Engineering (ICDE). Macao: IEEE, 2019. 1190–1201. [doi: 10.1109/ICDE.2019.00109]
- [107] Bouchard M, Jusselme AL, Doré PE. A proof for the positive definiteness of the Jaccard index matrix. Int'l Journal of Approximate Reasoning, 2013, 54(5): 615–626. [doi: 10.1016/j.ijar.2013.01.006]
- [108] Sadowski C, Levin G. Simhash: Hash-based similarity detection. Technical Report, 2007.
- [109] Norouzi M, Fleet DJ, Salakhutdinov RR. Hamming distance metric learning. In: Proc. of the 25th Int'l Conf. on Neural Information Processing Systems. Red Hook: Curran Associates Inc., 2012. 1061–1069.
- [110] Raymond JW, Willett P. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. Journal of Computer-aided Molecular Design, 2002, 16(7): 521–533. [doi: 10.1023/A:1021271615909]
- [111] Riesen K, Bunke H. Approximate graph edit distance computation by means of bipartite graph matching. Image and Vision Computing, 2009, 27(7): 950–959. [doi: 10.1016/j.imavis.2008.04.004]
- [112] Broumi S, Smarandache F. Neutrosophic Refined Similarity Measure Based on Cosine Function. Infinite Study, 2014
- [113] Malkauthekar MD. Analysis of Euclidean distance and Manhattan distance measure in Face recognition. In: Proc. of the 3rd Int'l Conf. on Computational Intelligence and Information Technology (CIIT 2013). Mumbai: IET, 2013. 503–507. [doi: 10.1049/cp.2013.2636]
- [114] Broder AZ. On the resemblance and containment of documents. In: Proc. of the Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171). Salerno: IEEE, 1997. 21–29. [doi: 10.1109/SEQUEN.1997.666900]
- [115] Zhao YX, Zhen G. MD5 research. In: Proc. of the 2nd Int'l Conf. on Multimedia and Information Technology. Kaifeng: IEEE, 2010. 271–273. [doi: 10.1109/MMIT.2010.186]
- [116] Massarelli L, Di Luna GA, Petroni F, Querzoni L, Baldoni R. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In: Proc. of the 2nd Workshop on Binary Analysis Research (BAR). San Diego, 2019. 1–11. [doi: 10.14722/bar.2019.23020]
- [117] Bottou L. Stochastic gradient learning in neural networks. Proc. of Neuro-Nimes, 1991, 91(8): 12.
- [118] Xiao Q, Luo H, Zhang C. Margin sample mining loss: A deep learning based method for person re-identification. arXiv:1710.00478, 2017.
- [119] Shirani P, Collard L, Agba BL, Lebel B, Debbabi M, Wang LY, Hanna A. Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In: Proc. of the 15th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment. Saclay: Springer, 2018. 114–138. [doi: 10.1007/978-3-319-93411-2_6]
- [120] Junod P, Rinaldini J, Wehrli J, Michielin J. Obfuscator-LLVM—Software protection for the masses. In: Proc. of the 2015 IEEE/ACM

- Int'l Workshop on Software Protection. Florence: IEEE, 2015. 3–9. [doi: 10.1109/SPRO.2015.10]
- [121] Luk CK, Cohn RS, Muth RM, Patil HG, Klauser AW, Lowney GN, Wallace S, Reddi VJ, Hazelwood KM. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*, 2005, 40(6): 190–200. [doi: 10.1145/1064978.1065034]
- [122] Stallman R, Pesch R, Shebs S. *Debugging with GDB*. Free Software Foundation, 1988: 675.
- [123] Kim JH, Sim HC, Kang YH, Ik Eom Y. Design and implementation of a remote debugger for concurrent debugging of multiple processes in embedded Linux systems. In: *Proc. of the IFIP Int'l Conf. on Network and Parallel Computing*. Wuhan: Springer, 2004. 280–283. [doi: 10.1007/978-3-540-30141-7_41]
- [124] Lee EH, Chang H, Jun SI, Cho JH. A cross debugging architecture for switching software. In: *Proc. of the 1996 Int'l Conf. on Communication Technology (ICCT1996)*. Beijing: IEEE, 1996. 214–217. [doi: 10.1109/ICCT.1996.545164]
- [125] Roussev V. Data fingerprinting with similarity digests. In: *Proc. of the 6th IFIP Int'l Conf. on Digital Forensics*. Hong Kong: Springer, 2010. 207–226. [doi: 10.1007/978-3-642-15506-2_15]
- [126] Kornblum J. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 2006, 3: 91–97. [doi: 10.1016/j.diin.2006.06.015]
- [127] Li D, Yin Q, LIN J, Lv XF. Firmware vulnerability detection in embedded device based on homology analysis. *Computer Engineering*, 2017, 43(1): 72–78 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-3428.2017.01.013]
- [128] Chen Y, Li H, Zhao WW, Zhang L, Liu ZJ, Shi ZQ. IHB: A scalable and efficient scheme to identify homologous binaries in IoT firmwares. In: *Proc. of the 36th IEEE Int'l Performance Computing and Communications Conf. (IPCCC)*. San Diego: IEEE, 2017. 1–8. [doi: 10.1109/PCCC.2017.8280478]
- [129] Ma C, Yang CS, Yang F, Zhuang YQ, Zhang ZW, Jia HZ, Xie XD. Trajectory factory: Tracklet cleaving and re-connection by deep Siamese Bi-GRU for multiple object tracking. In: *Proc. of the 2018 IEEE Int'l Conf. on Multimedia and Expo (ICME)*. San Diego: IEEE, 2018. 1–6. [doi: 10.1109/ICME.2018.8486454]
- [130] Andriesse D, Chen X, van der Veen V, Slowinska A, Bos H. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: *Proc. of the 25th USENIX Conf. on Security Symp.* Austin: USENIX Association, 2016. 583–600.
- [131] Redini N, Machiry A, Wang RY, Spensky C, Continella A, Shoshitaishvili Y, Kruegel C, Vigna G. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In: *Proc. of the 2020 IEEE Symp. on Security and Privacy (SP)*. San Francisco: IEEE, 2020. 1544–1561. [doi: 10.1109/SP40000.2020.00036]
- [132] Zhang MW, Sekar R. Control flow integrity for COTS binaries. In: *Proc. of the 22nd USENIX Conf. on Security*. Washington, DC: USENIX Association, 2013. 337–352.
- [133] Shoshitaishvili Y, Wang RY, Hauser C, Kruegel C, Vigna G. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: *Proc. of the 22nd Annual Network and Distributed System Security Symp.* San Diego: The Internet Society, 2015. 1–15.
- [134] Bao T, Burket J, Woo M, Turner R, Brumley D. BYTEWEIGHT: Learning to recognize functions in binary code. In: *Proc. of the 23rd USENIX Conf. on Security Symp.* San Diego: USENIX Association, 2014. 845–860.
- [135] Zeng JY, Fu YC, Miller KA, Lin ZQ, Zhang XY, Xu DY. Obfuscation resilient binary code reuse through trace-oriented programming. In: *Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security*. Berlin: Association for Computing Machinery, 2013. 487–498. [doi: 10.1145/2508859.2516664]
- [136] Kang MG, Poosankam P, Yin H. Renovo: A hidden code extractor for packed executables. In: *Proc. of the 2007 ACM Workshop on Recurring Malcode*. Alexandria: Association for Computing Machinery, 2007. 46–53. [doi: 10.1145/1314389.1314399]
- [137] Meng QK, Wen SM, Feng C, Tang CJ. Predicting buffer overflow using semi-supervised learning. In: *Proc. of the 9th Int'l Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*. Datong: IEEE, 2016. 1959–1963. [doi: 10.1109/CISP-BMEI.2016.7853039]

附中文参考文献:

- [127] 李登, 尹青, 林键, 吕雪峰. 基于同源性分析的嵌入式设备固件漏洞检测. *计算机工程*, 2017, 43(1): 72–78. [doi: 10.3969/j.issn.1000-3428.2017.01.013]



于颖超(1983—), 女, 工程师, 主要研究领域为网络安全, 嵌入式设备固件安全分析.



秦晓军(1975—), 男, 博士, 工程师, 主要研究领域为系统安全, 软件脆弱性分析, 人工智能.



甘水滔(1986—), 男, 博士, 副研究员, 主要研究领域为系统安全, 软件脆弱性分析.



陈左宁(1957—), 女, 博士生导师, CCF 会士, 主要研究领域为操作系统, 高性能计算, 网络和信息安全, 人工智能.



邱俊洋(1989—), 男, 博士, 工程师, 主要研究领域为系统安全, 软件脆弱性分析, 机器学习.