软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(7):3291−3313 [doi: 10.13328/j.cnki.jos.006919]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于静态和动态混合分析的内存拷贝类函数识别*

尹小康, 蔡瑞杰, 杨启超, 刘胜利

(数学工程与先进计算国家重点实验室 (信息工程大学), 河南 郑州 450001)

通信作者: 刘胜利, E-mail: mr_shengliliu@163.com

摘　要: 缓冲区溢出等内存错误漏洞的产生往往来自对内存拷贝类函数的不当使用. 对二进制程序中的内存拷贝类函数进行识别有利于发现内存错误漏洞. 目前针对二进制程序中内存拷贝类函数的识别方法主要借助静态分析来提取函数的特征、控制流、数据流等信息进行识别, 具有较高的误报率和漏报率. 为了提高对内存拷贝类函数识别的效果, 提出一种基于静态和动态混合分析的技术 CPSeeker. 所提方法结合静态分析和动态分析各自的优势, 分阶段对函数的全局静态信息和局部执行信息进行搜集, 对提取到的信息进行融合分析, 进而识别二进制程序中的内存拷贝类函数. 实验结果表明, 尽管 CPSeeker 在运行时间上有所增加, 但在内存拷贝类函数识别的效果上, 其 *F*1 值达到了 0.96, 远优于最新的工作 BootStomp、SaTC、CPYFinder 以及 Gemini, 并且不受编译环境 (编译器版本、编译器种类、编译器优化等级) 的影响. 此外, CPSeeker 在真实的固件测试中也有更好的表现.

关键词: 静态分析; 动态分析; 仿真执行; 内存拷贝类函数; 函数识别

中图法分类号: TP311

中文引用格式: 尹小康, 蔡瑞杰, 杨启超, 刘胜利. 基于静态和动态混合分析的内存拷贝类函数识别. 软件学报, 2024, 35(7): 3291–3313. http://www.jos.org.cn/1000-9825/6919.htm

英文引用格式: Yin XK, Cai RJ, Yang QC, Liu SL. Identification of Memory Copy Function via Hybrid Static and Dynamic Analysis. Ruan Jian Xue Bao/Journal of Software, 2024, 35(7): 3291–3313 (in Chinese). http://www.jos.org.cn/1000-9825/6919.htm

Identification of Memory Copy Function via Hybrid Static and Dynamic Analysis

YIN Xiao-Kang, CAI Rui-Jie, YANG Qi-Chao, LIU Sheng-Li

(State Key Laboratory of Mathematical Engineering and Advanced Computing (Information Engineering University), Zhengzhou 450001, China)

Abstract: Memory error vulnerabilities (e.g., buffer overflow) are often caused by improper use of memory copy functions. The identification of memory copy functions in binary programs is beneficial for finding memory error vulnerabilities. However, current methods for identifying memory copy functions in binary programs mainly rely on static analysis to extract functions' features, control flow, data flow, and other information, with a high false positive and false negative. This study proposes a technique, namely CPSeeker, based on hybrid static and dynamic analysis to improve the effectiveness of identifying memory copy functions. CPSeeker combines the advantages of static analysis and dynamic analysis, collects the global static information and local execution information of functions in stages, and fuses the extracted information to identify memory copy functions in binary programs. The experimental results show that CPSeeker outperforms the state-of-the-art BootStomp, SaTC, CPYFinder, and Gemini in identifying memory copy functions, despite its increased runtime consumption, and its *F*1 value reaches 0.96. Furthermore, CPSeeker is not affected by the compilation environment (compiler version, compiler type, and compiler optimization level). In addition, CPSeeker has a better performance in actual firmware tests.

Key words: static analysis; dynamic analysis; simulation execution; memory copy function; function identification

软件漏洞已成为影响网络空间安全的重要威胁. 对软件系统中的漏洞进行检测和修补成为一个研究热点[1]. 在无法获取软件系统源码的情况下, 研究者只能对二进制代码进行分析, 利用逆向技术对二进制程序中的信息 (函

* 基金项目: 科技委基础加强重点项目 (2019-JCJQ-ZD-113)
收稿时间: 2022-03-27; 修改时间: 2022-09-19, 2022-11-17; 采用时间: 2023-02-15; jos 在线出版时间: 2023-07-26
CNKI 网络首发时间: 2023-07-27

© 中国科学院软件研究所 http://www.jos.org.cn

数边界、函数名、参数等) 进行恢复, 结合污点分析[2,3]、符号执行[4,5]、二进制代码重写[6]以及模糊测试[7]等技术进行漏洞的检测和发现. 在软件漏洞中, 危害最严重的漏洞当属内存错误漏洞, 它能造成服务中断、系统崩溃、重要信息泄露等, 使得攻击者可以窃取机密信息、远程执行命令以及获得系统的控制权限.

对内存错误漏洞的发现和检测能够有效地避免网络攻击的发生, 成为安全研究的一个重要分支. 在进行内存数据的处理和转移时, 未对用户的输入或者网络数据包进行严格的检查导致了内存错误漏洞的产生. 内存数据的处理和转移常常借助于内存拷贝类函数 (memory copy function) 进行实现, 对内存拷贝类函数的识别和定位在内存错误漏洞的检测和发现中起着至关重要的作用. 例如 Mouzarani 等人[8]在对二进制程序进行模糊测试时需要首先分析对 *memcpy*, *strcpy* 等函数的调用情况; 工具 Arbiter[9]在挖掘二进制程序中的数据流敏感 (data-flow sensitive vulnerabilities) 的漏洞时, 依赖函数名来识别关键函数, 例如 *system*, *malloc*, *setuid* 等函数. 随着物联网技术的发展, 研究者开始重视物联网设备的安全, 对物联网设备的固件进行脆弱性分析, 例如 DTaint[10] 和 SaTC[11]等借助 *memcpy*, *strncpy* 等函数, 来定位污点数据的 sink 点, 然后对设备的固件程序进行污点分析. 然而, 这些方法在应用到剥离符号表的二进制程序 (stripped binary, 后文简称剥离二进制) 时, 会因无法定位到这些关键函数而导致效果急剧下降.

为了定位存在风险的函数调用位置, 即对内存拷贝类函数的调用点, 研究者提出了一些对内存拷贝类函数 (后文简称拷贝类函数) 识别的技术和方法[2,11−13], 来识别二进制程序中的拷贝类函数. 然而, 现有的方法依赖于函数名, 使用静态的分析方法进行特征的提取, 存在较高的误报率和漏报率. 对拷贝类函数进行识别的难点一方面来源于函数本身的复杂性, 即在实现自定义的拷贝类函数时, 由于功能需求, 需要对数据进行转换处理, 这个过程中可能涉及复杂的操作运算; 另一方面来源于编译器的优化, 即使是相同的源码, 经过编译器的优化后, 函数的控制流和数据流也发生了较大的变化. 因此需要确定拷贝类函数本质的特征, 才能提高拷贝类函数识别的准确率.

针对上述问题, 本文提出了一种基于静态和动态混合分析的拷贝类函数识别方法 CPSeeker, 该方法的流程如图 1 所示. CPSeeker 通过融合静态和动态分析提取到的信息, 对二进制程序中的拷贝类函数进行识别, 并且不依赖于函数名等信息, 使得该方法适用于剥离二进制. 第 1 阶段, CPSeeker 对函数进行静态分析, 分析函数的控制流图 (control flow graph, CFG) 和内存访问模式, 过滤掉在 CFG 结构上不具备拷贝类函数特征的函数 (非拷贝类函数), 只保留疑似拷贝类函数, 并提取循环路径、循环指令、内存依赖等关键信息; 第 2 阶段, 结合第 1 阶段提取到的信息, 构造参数输入, 对获取到的疑似拷贝类函数进行动态模拟仿真, 获取函数执行的状态信息, 包括执行轨迹和内存访问轨迹等. 最终 CPSeeker 融合静态和动态分析获取的信息, 对内存访问序列进行识别和函数执行前后的内存状态进行分析, 判断函数是否为拷贝类函数 (严格拷贝类函数和非严格拷贝类函数, 相关定义见第 2.2 节). 基于混合分析的拷贝类函数识别需要解决两个关键问题: (1) 在 x86 执行环境下, 如何捕获多指令集架构下的二进制函数的动态执行信息和 (2) 如何利用静态和动态分析提取的信息进行拷贝类函数的判定. 为解决问题 (1) 设计了单个二进制函数执行框架 autoEmu, 用于提取单个函数执行时的状态信息, 并能够在 x86 环境下执行多种指令集架构的二进制程序; 为解决问题 (2) 基于拷贝类函数的行为特征, 设计了内存连续访问序列判定方法, 并结合静态分析的结果对拷贝类函数进行识别. 该方法结合了动态分析和静态分析各自的优势, 弥补了静态分析方法准确率低的弊端, 同时避免了动态分析适用性差的问题, 提高了识别的准确率. 实验表明 CPSeeker 在 C 库中拷贝类函数和自定义实现的拷贝类函数识别中效果都优于现有的方法, 同时避免了受编译环境的影响.

本文的主要贡献包括以下 4 个方面.

1) 设计了单个二进制函数的模拟执行框架 autoEmu, 用于对函数的运行状态捕获, 支持多指令集架构, 可用于固件程序中函数级别的动态分析任务.

2) 提出了基于静态和动态混合分析的拷贝类函数识别模型, 并实现了二进制程序中拷贝类函数识别技术 CPSeeker, 该方法通过静态和动态的混合分析来提高对拷贝类函数识别的效果.

3) 与最新的工作 BootStomp, SaTC, CPYFinder 以及 Gemini 相比, CPSeeker 在对拷贝类函数的识别上具有更好的效果, 其 $F1$ 值达到了 0.96, 在真实的固件中的拷贝类函数识别也具有更好的效果.

© 中国科学院软件研究所 http://www.jos.org.cn

4) 为方便进一步的研究, 对所实现代码以及使用的数据集进行了开源, 存放在 GitHub 的仓库中 (https://github.com/CPSeek/CPSeeker).

本文第 1 节介绍对拷贝类函数识别的相关方法和研究现状. 第 2 节介绍本文所需的基础知识, 包括相关符号定义、对拷贝类函数识别问题的描述以及对仿真模拟器的介绍. 第 3 节介绍本文构建的基于静态和动态混合分析的拷贝类函数识别模型. 第 4 节介绍该方法的具体实现和特定问题的解决方法. 第 5 节通过对比实验验证了所提方法的有效性. 第 6 节对所提方法进行讨论分析. 最后总结全文.

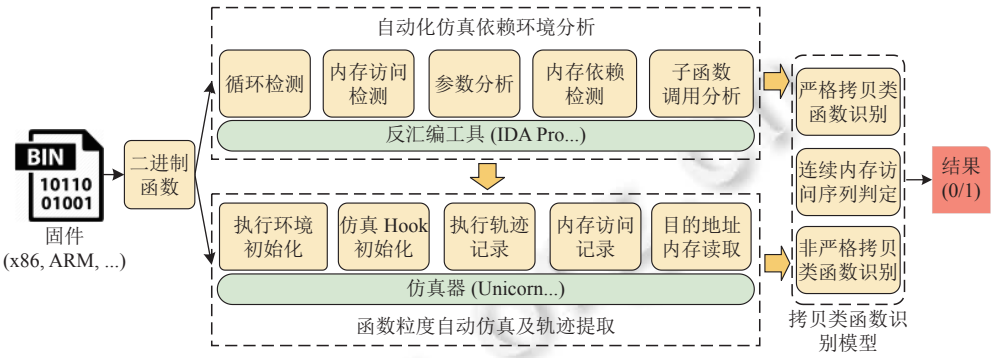


图 1 拷贝类函数识别方法 CPSeeker 的流程

1 拷贝类函数识别相关工作

内存拷贝类函数在缓冲区溢出和信息泄露等漏洞的检测上存在诸多的应用. 对内存拷贝类函数的不正确调用导致了内存错误漏洞的产生, 因此, 对内存拷贝类函数的识别和定位对漏洞的检测具有重要的价值. 研究人员关注到拷贝类函数对漏洞检测的重要性, 尝试提出了不同的方法来识别二进制程序中的拷贝类函数. 本文对现有的拷贝类函数识别方法总结, 将其分为 3 种类型: 基于符号表信息的方法、基于函数签名的方法和基于特征匹配的静态方法.

基于符号表信息的方法主要是通过借助二进制程序中的函数名进行识别, 当二进制程序未剥除函数名时, 通过逆向工具 (例如 IDA Pro[14]、Ghidra[15]、angr[16]等) 可以恢复出函数名, 直接通过函数名进行识别, 例如 SaTC[11] 和 Karonte[3], 但是当二进制程序剥除函数名后, 逆向工具目前仍无法完整的恢复出函数名, 解析出来的函数名称全部以地址的方式命名, 例如 IDA Pro 以“sub_”+函数起始地址的方式命名. 因此, 基于符号表信息的方法不适用于剥离二进制.

基于签名的方法是通过选取一些特征: 函数的参数个数和参数类型、函数的起始字节码、函数的返回值、控制流程图等信息, 为已知的函数构建函数签名, 利用函数签名来识别二进制程序中的函数, 例如 IDA Pro 中的库函数快速识别技术 (fast library identification and recognition technology, FLIRT)[17], 工具 Radare2 的 Zsignature[18] 等. 此外, 随着人工智能 (artificial intelligence, AI) 在程序分析领域的兴起, 研究者尝试使用 AI 来解决代码相似性的问题, 利用训练后的模型可以为函数生成嵌入向量 (作为函数的签名) 用于函数的匹配. 例如, Gemini[19]、VulSeeker[20]、SAFE[21]、PalmTree[22]、jTran[23]等. 该类型的方法能够识别剥离二进制中的函数, 但是无法识别未知的函数, 因为必须先为函数构建签名才能识别. 此外由于编译器会对函数的一些特征造成改变, 函数的签名也会随之改变, 这会造成识别的准确率下降.

基于特征匹配的静态分析方法是通过静态分析技术对拷贝类函数的特征进行分析和提取, 例如函数 CFG 中通常存在循环、内存访问地址存在增加固定的偏移等, 依据静态特征对二进制程序中的拷贝类函数进行识别, 例如 BootStomp[2]、Karonte[3]和 SaTC[11]等 (使用了 2 种方法: 基于符号表和基于特征匹配的方法). 此方法不仅能够识别剥离二进制中的拷贝类函数, 还一定程度上避免了受编译器的影响, 但是识别的准确率仍较低. 2020 年, Luo

© 中国科学院软件研究所 http://www.jos.org.cn

等人[24]提出静态审计源码中由循环路径导致的漏洞, 但由于二进制程序中丢失了较多的语义、变量类型、结构体等信息, 该方法不适用于对二进制程序的分析. 2022 年, 尹小康等人[13]提出了基于控制流和数据流分析的拷贝类函数识别方法 CPYFinder, 该方法通过分析函数中的循环路径以及循环路径中的数据流进行拷贝类函数的识别, 然而虽然该方法增加了对数据流的分析[25], 提高了拷贝类函数识别的准确率和召回率, 但是由于静态分析的局限性, 仍存在较高的误报率.

上述 3 种方法其本质上均是通过静态分析获取函数的特征, 然后进行拷贝类函数的识别. 由于程序在编译过程中存在信息的丢失, 静态分析很难完全恢复这些信息, 造成提取到的信息并非十分准确. 此外, 编译器对程序会进行优化, 加剧这种差异, 导致函数的静态特征发生改变, 同样造成了基于静态分析的方法识别效果的下降. 虽然编译器会改变函数的静态特征, 但是函数的功能是不受影响的. 因此, 本文并不直接以静态分析提取到的信息对函数是否为拷贝类函数进行判定, 也不在静态模式下分析寄存器间的数据流, 而是利用静态分析提取拷贝类函数识别的必要不充分因素 (例如存在循环结构、循环结构内同时存在内存的读取和写入等) 以及函数仿真执行时的依赖条件, 在静态分析的辅助下对二进制函数进行模拟执行, 提取函数执行时的状态和行为, 通过分析函数执行前后的内存状态、内存读取和访问信息, 结合静态分析获取到的信息 (包含内存读写的循环中的基本块信息、指令信息), 识别二进制程序中的拷贝类函数. 本文所提方法能够支持多种指令集架构, 避免了受编译器的影响.

2 基础知识

本文所提方法主要基于拷贝类函数的行为特征以及利用动态仿真执行捕获执行过程中的信息对其进行识别, 下面就相关概念和基本知识予以介绍.

2.1 符号定义

设存在二进制函数 f, 若 f 为拷贝类函数, 则记 $\varphi(f)=1$, 否则记 $\varphi(f)=0$.

函数指令集: 对整个函数遍历获得的指令集集合, 记为 F_i, 记录了每条指令的起始地址.

函数基本块集: 对函数的基本块遍历获得的基本块的集合, 记为 F_b, 记录了每个基本块的起始地址.

循环指令集: 对循环进行遍历获得的指令集合, 记为 L_i, 记录了循环路径中每条指令的起始地址.

循环基本块集: 对循环进行遍历获得的基本块的集合, 记为 L_b, 记录了循环路径中每个基本块的起始地址.

函数参数源地址数据: 构造的用于传递给函数的待处理数据, 记为 D_s.

函数目的地址数据: 函数执行完毕后, 从目的地址内存中读取到的数据, 记为 D_d.

执行指令轨迹: 在特定输入下, 函数执行指令的轨迹, 记为 E_i, 按照执行的先后顺序记录了函数执行的每条指令的起始地址.

执行基本块轨迹: 在特定输入下, 函数执行的基本块的轨迹, 记为 E_b, 按照执行的先后顺序记录了函数执行的每个基本块的起始地址.

内存读取集: 函数在执行时读取内存的集合, 记为 M_r, 记录了函数读取的内存地址、长度以及内容, 分别用 a_r, s_r, v_r 表示.

内存写入集: 函数在执行时写入内存的集合记为 M_w, 记录了函数写入的内存地址、长度以及内容, 分别用 a_w, s_w, v_w 表示.

2.2 拷贝类函数的识别问题

BootStomp[2]、Karonte[3]、SaTC[11]、CPYFinder[13]已经对拷贝类函数在代码结构上的特征进行了分析, 这里不再赘述. 本文在拷贝类函数的代码结构基础上, 对其执行时对内存的操作进行分析. 为便于阐述, 本文将拷贝类函数分为: 严格拷贝类函数和非严格拷贝类函数. 严格拷贝类函数是将内存单元中的数据不加修改地复制到另一段内存单元中的函数, 例如常见的 C 语言库中的拷贝类函数 *strcpy*, *strncpy*, *memcpy* 等函数. 非严格拷贝类函数是将内存单元中的数据 (不一定全部读取) 经过处理 (过滤、转换、重新编码等), 然后存储到另一段内存单元中的函

© 中国科学院软件研究所 http://www.jos.org.cn

数, 例如漏洞编号为 CVE-2020-8423 中的函数 *stringModify*() 对字符串进行了修改. 如图 2 所示, 展示了拷贝类函数对内存中数据的操作. 其中, 图 2 中从 (d) 到 (e) 为非严格的内存拷贝, 将数据存储到新分配的内存区域时, 对数据进行了修改, 修改后的数据在内容和大小上会存在差异, 例如在处理字符串时, 将特殊的符号删除; 图 2 中从 (b) 到 (a) 为严格的内存拷贝, 将数据存储到新分配的内存区域时, 保持数据的一致性. 内存拷贝类函数在访问内存上的一个重要特征是在内存的读取和内存的写入时具有地址空间上的连续性, 即从一段连续的内存区域读取数据, 然后经过处理存储到另一段连续的内存区域中. 因此, 拷贝类函数识别的关键和难点是如何获取二进制函数执行时的状态信息及内存访问情况以及对内存访问情况的分析.

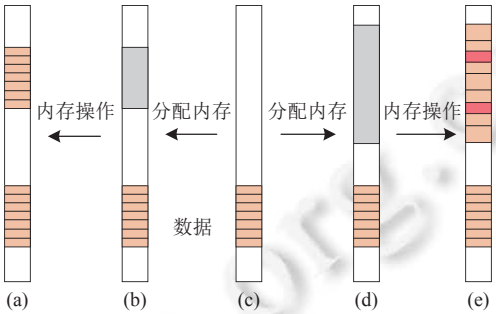


图 2 拷贝类函数对内存中数据的操作

2.3 Unicorn 仿真模拟器

一方面由于嵌入式设备中程序之间的相互依赖, 通常无法直接在桌面操作系统中执行并监控其状态; 另一方面嵌入式设备的指令集架构多样, 包括 ARM、MIPS、PowerPC (PPC) 等, 因此需要支持多指令集的 CPU 仿真器进行仿真. 常用的仿真器是 QEMU[26], 常用于对恶意代码分析[27]、漏洞分析[28]、嵌入式设备的安全分析[29]等, 支持对全系统的仿真. 然而由于 QEMU 除了对 CPU 进行仿真, 还包括 ROM、BIOS 及一些外围设备, 不适用于对单个函数的模拟执行. Unicorn 在 QEMU 的基础上进行了裁剪和开发以删除过多的功能, 只集中在对 CPU 的仿真上, 使其成为轻量级的模拟器. 此外, Unicorn 提供了更多的 API 用于研究人员的二次开发, 例如, 指令的插桩、Hook 等. 由于其良好的可拓展性, Unicorn 被应用到众多的产品上, 包括沙箱、模糊测试工具、文件分析器等. 据官方统计数据, 目前 Unicorn 被应用到 123 个工具中[30], 其中著名的工具有 Qiling 沙箱[31]、Radare2 逆向工具[18]、Pwntools 工具箱[32]、AFLplusplus 模糊测试工具[33]等. 利用 Unicorn 可以实现多种多样的分析任务.

因此, 为在 x86 平台下支持对固件中的程序进行分析, 本文采用 Unicorn 提供的 API 构建单个函数的模拟执行框架, 对单个函数进行执行, 记录函数执行时的状态信息, 同时结合静态分析获取的全局信息, 来对二进制程序中的拷贝类函数进行识别. 该方法结合了静态分析和动态分析各自的优势, 具有较好的拓展性以及较高的准确率.

3 基于混合分析的拷贝类函数识别方法 CPSeeker

本节对基于静态和动态混合分析的拷贝类函数识别方法的原理进行详细介绍. 二进制程序中拷贝类函数识别技术——CPSeeker 的流程如图 1 所示. CPSeeker 总共分为 3 个部分: 基于静态分析的信息提取、基于动态分析的信息提取以及对所提取信息的混合分析. 其中, 基于静态分析的信息提取主要对函数进行预处理, 提取循环参与指令等全局信息, 同时过滤掉不具备拷贝类函数特征的函数以及提取函数级仿真执行依赖的信息; 基于动态分析的信息提取负责对二进制函数仿真执行的环境进行初始化, 并记录函数执行的状态和对内存的修改等情况; 信息的混合分析处理静态分析和动态分析提取到的信息, 判断函数是否为拷贝类函数.

具体的拷贝类函数识别算法如算法 1 所示.

© 中国科学院软件研究所 http://www.jos.org.cn

算法 1. 基于混合分析的拷贝类函数识别算法.

输入: 函数起始地址 *ea*;
输出: 函数是否为拷贝类函数 (值为 0 或 1).

① *func* = *get_func*(*ea*);
② *blocks* = [*v* for *v* in *FlowChart*(*func*)];
③ *cfg* = *get_cfg* (*blocks*); /*生成控制流图*/
④ *loops* = *simple_cycles*(*cfg*); /*遍历 CFG 获取循环路径*/
⑤ if *loops* == []: return 0; /*无循环路径*/
⑥ *vex_irs* = *generate_vex_ir*(*blocks, loops*); /*将循环路径上的指令转换为 VEX IR 指令*/
⑦ *memacess* = *judge_mem*(*vex_irs*); /*在 IR 指令的基础上判断是否存在内存的读取和写入*/
⑧ if *memaccess* == 0: return 0; /*无内存的读取和写入*/
⑨ *envir* = *envir_analysis*(*ea*); /*对函数的执行环境依赖进行分析, 包括内存依赖, 子函数调用, 参数*/
⑩ *segm* = *get_segm*(*ea*); /*获取程序的段地址*/
⑪ *arch* = *get_arch*(); /*获取程序的架构*/
⑫ *aemu* = *autoEmu*(*arch, segm, envir*); /*初始化函数执行对象*/
⑬ *aemu.init_reg_mem*(); /*对函数的参数和内存进行初始化*/
⑭ *aemu.init_hooks*(); /*对函数分析使用的 Hooks 进行初始化, 包括库函数调用、系统调用/中断、执行轨迹记录、内存访问记录、错误处理等*/
⑮ *aemu.auto_emu_start*(); /*开始执行函数*/
⑯ *is_copy* = *aemu.strict_copy_mem*(); /*判断源和目的数据是否相同*/
⑰ if *is_copy* == 1: return 1; /*严格拷贝类函数*/
⑱ *is_copy* == *aemu.non_strict_copy_mem*(); /*判断是否存在内存连续的读取和写入, 以及循环路径是否执行等*/
⑲ if *is_copy* == 1: return 1; /*非严格拷贝类函数*/
⑳ return 0.

3.1 基于静态分析的信息提取

拷贝类函数在代码结构上的基本特征是依赖循环进行内存数据的复制、处理和转移 (x86 指令集下部分循环借助 rep 指令进行实现, 其本身也可以看作是一种循环实现方式). 因此, 对拷贝类函数的识别第 1 阶段的工作即是判断函数中是否存在循环、循环中是否存在内存的访问 (本文中内存访问包括内存的读取和写入) 以及为动态仿真执行提取依赖信息. 相比于动态分析, 静态分析能够捕获函数的全局信息, 更方便、适用性更强. 因此, 首先通过静态分析提取函数的信息: CFG 以及 $\{F_i, F_b, L_i, L_b\}$, 并对待分析的函数进行过滤, 加快识别的速度.

如图 3 所示, 基于静态分析的信息提取主要分为 8 个步骤: 循环检测、路径提取、IR (intermediate representation) 转换、内存访问检测、参数分析、内存依赖分析、函数调用分析和信息整合. 静态分析阶段首先对函数的 CFG 进行提取和分析, 识别 CFG 中是否存在循环路径, 过滤掉 CFG 中不包含循环的函数, 以及保存循环路径中包含的基本块和指令信息; 然后将循环路径中的代码转换为 IR 代码进行内存访问检测; 在满足存在内存读取和加载指令后, 对函数的参数情况、内存依赖和子函数调用进行分析, 包括参数的类型和参数的个数, 用于对函数模拟仿真的输入, 并对函数执行中访问到的内存进行预分析, 降低模拟执行失败的概率; 对函数在执行过程中子函数调用情况进行分析, 用于对函数执行环境的初始化; 最后对收集到的信息进行整合, 以一定格式存储用于函数粒度的模拟仿真和拷贝类函数的判定.

内存拷贝功能的实现往往借助循环结构进行实现 (x86 指令集下可借助“rep”指令实现, 这一特殊情况根据关键字匹配处理, 这里不再详细讨论), 在非 x86 指令集下, 对循环的检测是拷贝类函数识别的关键. 通过对 CFG 进

© 中国科学院软件研究所 http://www.jos.org.cn

行遍历来判断是否存在循环结构. 当函数中存在循环结构时, 提取循环路径上指令集合 L_i 和基本块的集合 L_b, 同时提取二进制函数的指令集合 F_i 和基本块集合 F_b. 此外, 如果函数中存在多个满足条件的循环路径, 则对每个循环路径上的指令和基本块集合进行提取并单独存储.

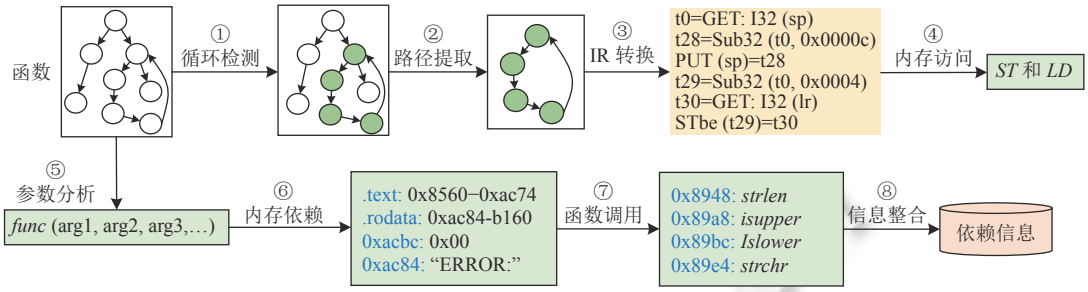


图 3 静态分析预处理与信息提取

循环路径中存在内存的读取和写入是具有内存拷贝功能的前提条件. 因此, 循环路径中内存访问的检测可以减少待分析函数的数量. 如图 4 中所示, 以 MIPS 指令集下函数 *strcmp* 为例, 从图中可以看出该 CFG 存在循环路径, 并且循环路径中存在对内存的读取 (指令"lbu v0,0(v0,0(a0)"和指令"lbu v1,0(v1,0(a1)"), 但是却未将数据写入到内存中, 因此为非拷贝类函数. 对于内存访问检测模块, 为方便支持不同的指令集架构, 将二进制代码转换成 IR 代码进行分析, 例如 VEX IR[34], LLVM IR[35]等. 将二进制代码转换成中间语言代码能够减少分析的工作量以及降低对不同指令集汇编代码知识的依赖程度.

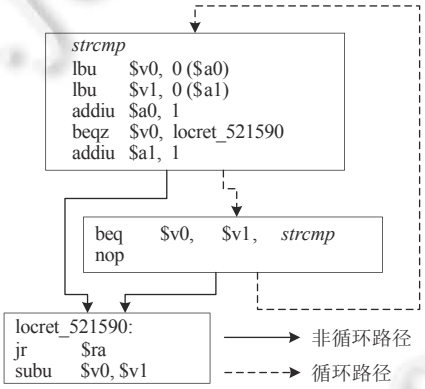


图 4 非拷贝类函数 (函数 *strcmp*)

参数分析模块的结果用于对动态执行环境的初始化, 即对函数的参数进行赋值. 拷贝类函数通常使用函数参数来接收输入, 包括拷贝的源地址、目的地址、长度以及其他参数等. 该模块主要分析函数参数的个数以及各个参数的类型. 目前已经有很多函数参数类型的恢复技术, 其中函数参数类型的恢复可以分为基于传统技术的方法和基于神经网络的方法. 基于传统技术的方法通过动态或静态分析对参数的访问类型, 参与的运算类型以及进行的内存操作情况进行提取, 结合专家知识推导出参数类型的可能值, 例如 OSPREY[36]; 基于神经网络的方法是通过将函数指令输入到神经网络模型中, 利用神经网络进行参数类型的推理, 降低了对专家知识的依赖, 成为一个研究的热点, 例如 EKLAVYA[37], NFRE[38], DIRTY[39]. 由于本文的核心是对拷贝类函数的识别, 而非函数参数类型的恢复, 并且只需要获取参数的类型是指针类型 (源地址, 目的地址) 还是整型 (长度) 即可, 因此采用常规的推理方法或者根据专家知识预设参数的类型和个数, 对函数的参数进行赋值, 用于函数的仿真执行.

内存依赖分析是为了确保对函数的模拟仿真能够成功执行. 部分函数在使用参数接收输入的同时还可能访问其他的内存单元, 例如存储在内存中的全局变量, 常量字符串等. 通过对函数的指令进行遍历分析, 获取指令对内

© 中国科学院软件研究所 http://www.jos.org.cn

存地址以及数据的引用关系, 确定函数执行中可能访问的内存单元. 在仿真环境初始化时, 对其初始化, 防止在执行过程产生非法访问造成执行失败.

子函数调用分析对函数在执行过程中的函数调用情况进行分析, 静态分析情况下只能获取函数的直接调用情况, 对于间接调用情况由于分析的准确性有限, 暂时不做考虑. 子函数调用分析一方面为了防止函数在执行过程中会迭代的调用, 造成执行效率较低; 另一方面是由于非静态编译的程序存在导入函数, 例如常见的 C 语言库函数 *strlen*(), *strcmp*() 等函数. 子函数调用分析用于限制函数仿真执行时调用的层级, 以及对常见的库函数进行 Hook, 使得函数尽可能地完整执行. 使用递归算法对函数的子函数调用进行分析: 首先查找函数调用点, 并解析出函数的地址, 判断其是否为库函数或者导入函数; 对于非导入函数则解析其子函数调用情况, 直至解析完所有的非导入函数或者达到限定的层级. 二进制函数的子函数调用解析结果以字典的形式保存: $\{call\_ea:(func\_ea, func\_name, func\_type)\}$, 记录函数的调用地址、函数地址、函数名以及函数类型. 当分析动态链接二进制程序时, 则对导入函数进行 Hook, 对其功能进行仿真执行; 当分析静态编译的二进制程序时, 则直接将子函数加载到内存空间进行执行.

3.2 基于动态分析的信息提取

相比于静态分析, 动态分析能够根据函数的输入准确提取函数的执行状态. 为在 x86 平台下支持对固件程序中的函数执行 (固件程序通常包含多种指令集架构, 例如 x86、MIPS、ARM), 使用仿真执行对函数的执行状态进行提取, 用于对拷贝类函数的识别. 动态分析的流程如图 5 所示, 该阶段首先为函数的执行构建内存空间映射, 将函数执行依赖的内存、寄存器以及参数等进行初始化; 然后进行仿真 Hook 的初始化, 对函数执行过程中的函数调用、内存访问、系统调用等设置 Hook 操作, 以及对执行错误进行 Hook 处理; 最后是对函数的执行状态信息进行捕获, 包括对内存读取的记录, 内存写入的记录, 执行轨迹的记录, 以及从目的地址分别读取数据用于内存中数据的比对.

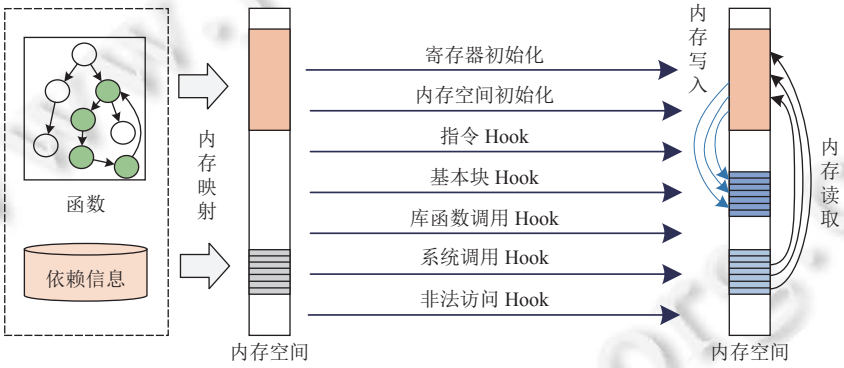


图 5 二进制函数仿真执行与信息提取

为仿真执行二进制函数, 则需要对函数执行所依赖的执行环境进行初始化, 包括可执行代码空间、数据空间、栈空间、函数参数、寄存器等的初始化. 为了执行不同指令集架构的函数, 需要在 x86 平台上为其执行环境构建内存映射, 主要分为 4 个内存映射: 可执行代码段、执行依赖数据段、I/O 数据段以及栈空间. 由于嵌入式固件中的程序一般只有几 MB, 因此可以直接为整个二进制程序构建内存映射. 当二进制程序较大时也可以根据函数的地址进行内存空间的映射. 构建的 I/O 数据段内存映射用于模拟从 I/O 接收到的数据, 存放拷贝前后的内存数据, 使用 $D(a_s,a_e)$ 表示, 其中 a_s 表示数据空间的起始地址, a_e 表示数据空间的结束地址. 构建的栈空间内存映射用于存放函数执行过程中的临时变量以及函数的部分参数值. 根据二进制程序的指令集架构和参数传递约定[40], 对参数和寄存器进行初始化, 将构造好的函数输入 D_s 写入到内存和寄存器中. 对于拷贝类函数依赖的输入为源地址 src 、目的地址 dst 以及可能存在表示长度的参数, 拷贝前的长度用 l_s 表示, 拷贝后的长度用 l_d 表示. 源地址和目的地址设定到映射的数据段之间, 即满足: $a_s < dst + l_d < src + l_s < a_e$, 确保源地址内存中的数据和目的地址内存中的数据互不干扰 ($dst + l_d$ 和 $src + l_s$ 可以互换位置), 即源地址和目的地址应尽量远离彼此.

© 中国科学院软件研究所 http://www.jos.org.cn

对函数执行状态的捕获首先需要对子函数调用 Hook、指令执行状态捕获、内存状态捕获等功能进行初始化. 在执行之前, 为了捕获函数的执行状态分别添加代码 Hook、基本块 Hook、系统调用/中断 Hook、非法内存访问 Hook 以及内存访问记录 Hook. 代码 Hook 和基本块 Hook 用于记录执行的指令和基本块. 此外, 使用代码 Hook 来处理函数调用, 当调用的函数是动态加载的函数 (导入函数) 时, 由于此类函数的可执行代码未加载到内存中, 对此类函数进行特殊处理, 以最大限度的执行. 例如对于 C 语言的库函数或者已知函数, 对这些函数的功能进行模拟实现, 将执行的结果写入到寄存器或者内存中. 以函数 *strlen*() 为例对执行过程进行说明, 在执行到函数 *strlen*() 时, 由于库函数是动态链接的, 无可执行代码, 因此不去执行真正的函数 *strlen*(), 而是直接读取来自栈或寄存器的参数, 从内存中读取数据并统计字符串的长度, 然后将结果写入到返回值寄存器中, 继续执行函数. 由于无法处理系统调用/中断, 因此使用 Hook 对系统调用或者中断跳过执行, 直接返回结果 0, 避免函数执行出现错误. 此外, 函数的参数存在多样性, 并不能保证传递给函数的参数完全正确, 因此在出现非法访问未映射的内存单元时, 对其进行 Hook, 输出内存访问错误情况, 提供异常反馈. 内存访问 Hook 用于在函数执行中发生对内存的读取和写入时, 对内存的读写情况进行记录.

代码 Hook 和基本块 Hook 在二进制函数执行时对函数执行的指令和基本块进行记录, 获取函数执行指令集合 E_i 和函数执行基本块集合 E_b . 记录的内容主要为: 指令的地址、指令执行的次数、指令的大小、基本块的地址、基本块的大小. 同时在调试的模式下, 可以记录每条指令执行后各个寄存器的值, 用于对异常的分析.

内存访问 Hook 对函数执行过程中读取和写入的内存情况进行记录, 获取函数执行过程中内存读取的信息 $M_r = \{(a_r,\ s_r,\ v_r)\}$ 和函数执行过程中内存写入的信息 $M_w = \{(a_w,\ s_w,\ v_w)\}$. 记录的内容为: 读取内存的地址、读取的长度和内容、写入内存的地址、写入的长度和内容. 该信息作为判定函数是否为拷贝类函数的信息之一.

目的内存读取负责对函数执行完成后目的地址 *dst* 指向内存空间的数据进行读取, 获取内存中的数据 D_d , 用于与源地址 *src* 指向的内存中存储的数据 D_s 进行比较.

3.3 对捕获数据的混合分析

混合分析将静态分析获取的函数的全局数据与动态仿真执行捕获的程序的状态信息进行融合分析, 进而判断函数是否为拷贝类函数.

经过静态分析和动态仿真执行后, 获取了函数的信息 $T = \{F_i, F_b, L_i, L_b, E_i, E_b, M_r, M_w, D_s, D_d\}$. 其中信息组合 $T_1 = \{F_i, L_i, E_i, M_r, M_w, D_s, D_d\}$ 和信息组合 $T_2 = \{F_b, L_b, E_b, M_r, M_w, D_s, D_d\}$ 均可用于判断函数是否为拷贝类函数, 不同的是 T_1 以指令粒度进行分析, 而 T_2 以基本块粒度进行分析. 由于静态分析和动态分析在基本块的划分时会存在差异 (例如, IDA 分析出来的基本块和 Unicorn 执行时获取的基本块会存在部分差异, 详见第 4.2 节), 所示以指令粒度进行分析在识别的效果上会好于以基本块粒度进行的分析, 但在时耗上会略有增加. 这两种只是处理上的差异, 在方法上是相同的, 因此只以其中的一种来阐述如何利用这些利用静态和仿真执行获取的信息来判定函数是否为拷贝类函数, 即对 $T_1 = \{F_i, L_i, E_i, M_r, M_w, D_s, D_d\}$ 进行分析.

3.3.1 连续内存访问序列判定

拷贝类函数在进行内存数据的复制及处理时, 必定存在对内存的连续读取和连续写入. 因此需要对内存访问轨迹进行处理, 判断其中是否存在连续的内存读取和连续的内存写入. 对于判定函数中是否存在对内存区域的连续读取和连续写入, 判定方法是一致的, 只是需要对读取和写入内存序列分别处理, 因此这里以 $M_r(1,n) = \{(a_1, s_1, v_1), (a_2, s_2, v_2), \ldots, (a_n, s_n, v_n)\}$ 为例来判定该内存访问序列是否存在连续的内存访问. 设存在一个内存读取序列, 记为 $M_r(i, i+j) = \{(a_i, s_i, v_i), (a_{i+1}, s_{i+1}, v_{i+1}), (a_{i+2}, s_{i+2}, v_{i+2}), \ldots\}$, 若存在 $\forall\, k \in [0, j),\ k \in \mathbb{Z}$, 有 $a_{i+k} + s_{i+k} = a_{i+k+1}$, 则 $\delta(M_r(i, i+j)) = 1$, 即 $M_r(i, i+j)$ 为连续序列, 序列的长度 (读取的内存长度) 为 j . 同理, 若 $\delta(M_w(m, m+n)) = 1$, 则 $M_w(m, m+n)$ 为连续序列, 序列的长度为 n .

3.3.2 拷贝类函数判定

由于严格拷贝类函数在拷贝数据前后保持了数据的一致性, 能够通过源地址内存数据和目的地址内存数据的对比, 对其进行判定, 更加的快速高效, 而对于非严格拷贝类函数, 需要融合处理多种信息, 因此首先依据严格拷贝

© 中国科学院软件研究所 http://www.jos.org.cn

类函数的判定标准进行识别, 再用非严格拷贝类函数的判定标准进行识别.

(1) 严格拷贝类函数识别

对于严格的拷贝类函数, 由于在数据拷贝前后, 数据未发生改变, 因此最直接的判定方式为判断源地址和目的地址内存中的数据是否相等, 当不相等时则使用非严格拷贝类函数的判断方式. 严格拷贝类函数判定方式为:

$$D_r = D_w \Rightarrow \varphi(f) = 1 \tag{1}$$

(2) 非严格拷贝类函数识别

除了 C 语言库中的拷贝类函数, 开发者会根据需求自定义实现拷贝类函数, 这些函数在内存拷贝时会对数据进行修改, 因此, 公式 (1) 中的判定方式不再适用, 这里考虑使用拷贝类函数对内存的访问特点进行判定. 由于写入内存中的数据会发生变化, 因此不再考虑内存访问时的数据. 非严格拷贝类函数的判定方式为:

$$\begin{cases} \exists\,(M_r(i,i+j)) \text{ s.t. } \delta(M_r(i,i+j)) = 1 \wedge j \geqslant \theta_r \\ \exists\,(M_w(m,m+n)) \text{ s.t. } \delta(M_w(m,m+n)) = 1 \wedge n \geqslant \theta_w \\ \exists\, L_i \text{ s.t. } L_i \subset E_i \end{cases} \Rightarrow \varphi(f) = 1 \tag{2}$$

其中, θ_r 为连续读取序列长度的阈值, θ_w 为连续写入序列长度的阈值. 设置阈值的原因是由于非严格拷贝类函数在处理内存数据时, 会因为某个关键字或者特殊字节的存在停止读取全部的字符串, 导致内存连续读取的长度会与存储到源地址指向内存区域中数据的长度不一致, 即 $j < l_s$. 同理, 由于对读取到的数据会存在丢弃或者重新进行编码, 会导致 $n \neq l_d$. 而 $L_i \subset E_i$ 则是保证了循环路径得以执行.

此外, 为方便对函数的执行情况进行分析和调试, 提供了函数的执行覆盖率: 代码覆盖率和基本块覆盖率, 如公式 (3) 和公式 (4) 所示, 即函数中已执行指令的数量 $N(E_i \cap F_i)$ (基本块的数量 $N(E_b \cap F_b)$) 比函数中总的指令数 $N(F_i)$ (基本块数 $N(F_b)$). 正如前文所述, 由于函数的多样性, 传递给函数的参数值并不能够完全保证函数的正常执行结束或者只执行了错误处理的分支, 对于执行中途中断或者代码覆盖率低的函数, 则利用静态分析方法 (即控制流和数据流分析的方法) 进行判断.

$$\text{代码覆盖率}: P_i = \frac{N(E_i \cap F_i)}{N(F_i)} \tag{3}$$

$$\text{基本块覆盖率}: P_b = \frac{N(E_b \cap F_b)}{N(F_b)} \tag{4}$$

4 系统实现

本文对所提方法 CPSeeker 进行实现, 实现环境为 Intel Core 6-core 3.7 GHz i7-8700K CPU and 64 GB RAM, 实现所使用的语言为 Python, 使用的软件包括: IDA Pro (版本 7.5)、Python (版本为 3.7.3)、pyvex (版本为 9.0.5376)、networkx (版本为 2.4)、Unicorn (版本为 1.0.2rc4), 以及基于 buildroot 构建的交叉编译环境用于生成不同架构、不同优化等级的二进制程序.

4.1 基于静态分析的信息提取

本文和 CPYFinder[13]采用相同的方法, 即借助 IDA Pro 和 networkx 库[41]进行实现. 利用 IDA Pro 的函数 *FlowChart*() 来获取函数基本块间的关系, 用于构建 CFG, 利用 networkx 的函数 *simple_cycles*() 判断 CFG 是否包含循环路径以及生成函数的所有循环路径.

对于内存访问情况的判定, 本文借助于中间语言转换工具 pyvex 进行实现. pyvex 将不同指令集的代码转换成相同的格式, 即 VEX IR, 方便对指令的分析. pyvex 会将所有的内存访问指令转换成自身的格式进行表示, 用“LDb(l)e”表示从内存中加载数据以及用“STb(l)e”表示将数据存储到内存中 (此处“LDb(l)e”分为“LDbe”和“LDle”, 指二进制文件的大端和小端, “STb(l)e”同理). 因此, 首先将循环路径中的所有指令 L_i 转换成 VEX IR, 然后对所有的 VEX IR 指令进行遍历检测是否同时包含“LDb(l)e”和“STb(l)e”指令. 对于同时存在内存读取和写入的函数, 则利用函数 *Heads*(ea, end) 获取整个函数指令的起始地址 F_i, 利用函数 *FlowChart*() 获取整个函数基本块的地址 F_b.

© 中国科学院软件研究所 http://www.jos.org.cn

同样从循环路径中提取, 参与循环的基本块的起始地址 L_b. 利用 IDA 的解析结果, 调用函数 *DataRefsFrom*() 来获取数据的引用关系, 并使用函数 *XrefsFrom*() 对函数的调用情况进行分析, 提取动态仿真和拷贝类函数判定依赖的数据, 使用函数 *is_call_insn*() 来获取函数的调用点, 并通过函数的 flags 标志位来判定是否为库函数, 通过递归来识别所有的函数调用信息.

4.2 基于动态仿真的信息提取

为了动态仿真执行单个的二进制函数, 基于 Unicorn 实现了 autoEmu 框架. 利用 Unicorn 提供的函数 *mem_map*() 分别构建虚拟的代码段内存映射、数据段内存映射以及栈空间映射等, 其中: 数据段的基地址设定为 0x6ffff000, 映射的大小为 2 MB; 栈空间的基地址设定为 0x7ffff000, 映射的大小为 2 MB, 并利用函数 *mem_write*() 将二进制程序加载到映射的内存空间, 利用函数 *mem_write*() 将构造的数据 D_s 写入源地址的内存单元, 利用 *mem_write*() 将源地址 *src*, 目的地址 *dst*, 以及其他参数写入到栈中或者利用函数 *reg_write*() 将参数写入到寄存器中. 使用函数 *hook_add*() 分别添加代码 Hook、基本块 Hook、系统调用/中断 Hook, 非法内存访问 Hook 以及内存访问 Hook, 分别用于获取 E_i, E_b, M_r, M_w. 将内存写入和内存读取的信息分别存放到变量 *mem_write_addr* 和变量 *mem_read_addr* 利用函数 *mem_read*() 从目的地址 *dst* 指向的内存中读取数据并保存到 D_d 中.

在进行库函数的 Hook 时, 依据指令集架构的参数传递约定, 从寄存器中或者栈空间中读取参数, 然后对库函数的功能进行模拟. 这里预设的函数参数的个数值最大为 8 个. 以 MIPS 指令集为例, MIPS 指令集提供了 4 个参数寄存器 $\{a_0,a_1,a_2,a_3\}$, 因此首先从参数寄存器中读取值, 再从栈空间中读取 4 个数据, 构成了函数参数集合. 对应的库函数从这些参数集合中依次读取参数, 例如函数 *strlen* 只需读取第 1 个参数, 然后从该参数指向的内存中读取数据直到遇到“\x00”并统计读取到的长度, 然后使用函数 *reg_write* 将字符串长度写入到函数调用返回值寄存器 *v*0 中 (MIPS 指令集下返回值寄存器为 *v*0 和 *v*1, 此处函数 *strlen* 只有一个返回值).

此外由于 MIPS 指令集存在分支延迟槽, 即对其子函数的调用地址直接 Hook 时, 则会导致利用分支延迟槽传递的参数无法执行, 因此需要特殊处理. 对于 MIPS 指令集子函数的 Hook 采用的方式是将分支延迟槽中的指令写入到函数调用处的地址, 将分支延迟槽中的指令提前, 然后将分支延迟槽中的指令进行“nop”化处理, 将“\x00\x00\x00\x00”写入到分支延迟槽的位置, 对分支延迟槽的地址进行 Hook. 这样确保了在执行函数调用时, 函数的参数传递是正确的.

由于 IDA 和 Unicorn 在解析基本块时的差异以及效率问题, 因此设计了指令粒度和基本块粒度的判断循环路径是否执行的标准. 以后文图 6 为例说明 IDA 和 Unicorn 对基本块处理的差异. 该图展示了 IDA 对 ARM 指令集下函数 *memcpy*() 分析出的部分基本块, IDA 将其分割为 3 个基本块{loc_2D930, loc_2D938, loc_2D93C}, 其中循环路径为{loc_2D93C}, 然而当 Unicorn 执行时, 拷贝的内存较少时, 则就执行一次基本块 loc_2D93C, 其在记录基本块时, 由于动态仿真是局部的执行, Unicorn 会将基本块 loc_2D938 与基本块 loc_2D93C 合并, 只会存在基本块 loc_2D938 的起始地址. 当以基本块起始地址判定循环路径是否执行时, 则会出现无法匹配的现象, 而以指令粒度进行判断不会出现这种差异. 因此设计了第 3 节所述的两种粒度的判定: 指令和基本块粒度的判定方法.

5 实验分析

本节对 CPSeeker 的识别效果进行评估, 并与现有方法 BootStomp、Karonte、SaTC、CPYFinder 以及 Gemini 进行对比. 主要包括对库函数识别效果、自定义函数识别效果、对多架构的支持、受编译环境的影响、在实际的剥离二进制程序中 (固件) 的识别效果, 漏洞相关函数的发现, 以及运行效率方面进行评测.

5.1 数据集和编译环境

5.1.1 数据集

本文从数据集 I、数据集 II 和数据集 III 分别测试所提方法 CPSeeker 的效果, 以及与基准方法进行对比 (数据集已发布在 https://github.com/CPSeek/CPSeeker). 各个数据集详情如下.

© 中国科学院软件研究所 http://www.jos.org.cn

数据集 I: 本文首先使用与 CPYFinder 相同的数据集, 即 C 语言库中的拷贝类函数和其从 GitHub (Netgear R7000 路由器代码[42]以及 Mirai 代码[43]) 上搜集的 22 个此类的函数 (详见文献 [13]).

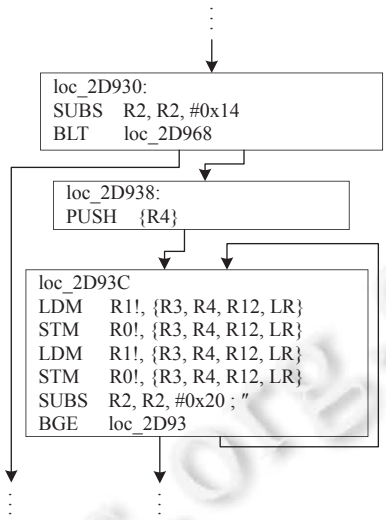


图 6 IDA 对函数基本块的分析结果

数据集 II: 为了测试在实际固件分析中的效果, 收集了路由器中由拷贝类函数引起的内存错误漏洞 (溢出) 漏洞进行测试, 此类漏洞为实现非严格的内存拷贝引起的, 即并非调用 C 语言库中的拷贝类函数 (*strcpy*, *strncpy* 等函数) 引起, 最终共收集到 6 个此类的 CVE 漏洞, 详细信息如表 1 所示.

表 1 数据集 II 的详细信息

厂商	型号 (版本)	二进制程序	CVE
ASUS	RT-N12 B1 (3.0.0.4.3802943)	networkmap	CVE-2017-6548
TP-Link	TL-R600VPNv3 (1.3.0)	httpd	CVE-2018-3950 CVE-2018-3951
	wr841nv10_en (150310)	httpd	CVE-2020-8423
	wr940nv4_us (160617)	httpd	CVE-2017-13772
D-Link	DIR-816 A2 (1.10B05)	goahead	CVE-2018-11013

数据集 III: 为了证明 CPSeeker 在真实的剥离二进制程序中的效果, 本文选取了思科 C2900 系列路由器的固件 (文件名为 c2900-universalk9-mz.SPA.157-3.M2.bin) 进行测试. 该固件是静态编译并且完全剥除了函数名, 符合所提方法使用场景. 为了构建真值标签 (ground truth), 依据关键字符串以及手工分析构建了测试数据集, 最终共有 20 个拷贝类函数.

5.1.2 编译环境

本文使用和 CPYFinder 相同的编译环境. 如表 2 所示为实验评估中使用的编译器, 编译器类型及优化选项, 使用 gcc 和 clang 这 2 个编译器进行测试, 并使用 buildroot 构建了不同指令集的交叉工具链, 用于生成不同目标架构的程序. 由于当前 clang 的 O3 和 O4 等级优化相同, 因此不再评估 O4 优化等级下的效果.

表 2 评估实验中所使用的编译器

编译器类型	编译器版本	优化等级
gcc	4.5.4, 4.6.4, 4.7.4, 4.8.5, 5.4.0, 5.5.0, 6.5.0, 7.5.0	-O0-O3
clang	3.9.1, 4.0.1, 5.0.1, 6.0.1	-O0-O3

© 中国科学院软件研究所 http://www.jos.org.cn

5.2 评价指标及基准模型

本文使用如下指标来评估本文所提方法的效果: 精准率 (precision) 记为 P, 其中 $P \in [0,1]$, 如公式 (5), 召回率 (recall) 记为 R, 其中 $R \in [0,1]$, 如公式 (6), 以及 $F1$ 分数 ($F1$-score) 记为 $F1$, 其中 $F1 \in [0,1]$, 如公式 (7). 精准率和召回率越高效果越好, $F1$ 是对精准率和召回率的调和, 更能直接展示方法的效果, 因此最终使用 $F1$ 来评估各个方法的效果. 此外为评估方法的效率, 使用时耗比值 (ratio) 记为 r, 如公式 (8), 用于比较各个方法的时耗, 其中 $t(a)$ 为方法 a 的时耗, $t(b)$ 为方法 b 的时耗.

$$P = \frac{TP}{TP+FP} \tag{5}$$

$$R = \frac{TP}{TP+FN} \tag{6}$$

$$F1 = 2 \times \frac{P \times R}{P+R} \tag{7}$$

$$r = \frac{t(b)}{t(a)} \tag{8}$$

其中, TP 为将拷贝类函数识别为拷贝类函数的数量; FP 为将非拷贝类函数识别为拷贝类函数的数量; FN 为将拷贝类函数识别为非拷贝类函数的数量.

由于本文的分析对象是多指令集多优化等级的剥离二进制, 基于符号表的方法无法在缺少符号表时工作, 而基于签名的方法取决于签名库的数量和质量, 要取得较好效果则需要为每一种指令集和每个优化等级下的函数构建签名, 因此, 不再与这两类传统的方法进行对比, 只与基于特征匹配的方法和基于 AI 的代码相似性检测方法进行对比. 本文首先选取了的基于特征的方法 BootStomp (USENIX 2017), Karonte (S&P 2020), SaTC (USENIX 2021) 以及最新基于控制流和数据流分析的 CPYFinder (JCRD 2022). BootStomp 提出了对拷贝类函数的识别问题并将其应用到对 bootloader 的安全分析中, Karonte 和 SaTC 对拷贝类函数的识别方法进行了进一步改进并应用到污点分析中. 此外, 由于机器学习被广泛应用到程序分析领域, 并且取得了不错的效果, 因此, 本文选择使用基于 AI 的代码相似性检测方法进行拷贝类函数的识别 (基于代码相似性的方法可以看作是基于签名方法的一种特殊形式), 选取的方法为 Gemini, 一个基于图神经网络的代码相似性分析方法, 支持跨架构的函数粒度的相似性分析 (由于 Karonte 与 SaTC 对拷贝类函数实现完全相同, 本文中只与 SaTC 对比). CPYFinder 指出 SaTC 对参数的个数限制会影响其识别的效果 (angr 对参数个数错误的识别), 因此采用相同的方法, 即取消 SaTC 对参数个数的限制. 表 3 展示各个方法实现主要依赖的工具和支持的指令集架构 (注: 在实现本文所提方法时, Unicorn 还未支持 PPC 指令集, 因此目前 CPSeeker 还未支持 PPC, 后续将对 Unicorn 进行升级, 并支持 PPC).

表 3 现有方法与 CPSeeker 对比

方法	实现工具	支持的指令集
BootStomp	IDA Pro	ARM
Karonte	angr, pyvex	ARM, MIPS
SaTC	angr, pyvex	ARM, MIPS
CPYFinder	IDA Pro, pyvex	x86, ARM, MIPS, PPC
CPSeeker	IDA Pro, pyvex, Unicorn	x86, MIPS, ARM

5.3 实验方法

本文使用来自 C 语言库中的拷贝类函数和从 GitHub 上收集到的自定义实现的拷贝类函数, 根据不同的编译环境, 交叉编译成二进制程序作为数据集 (数据集中包含了一些不具备内存复制功能的函数, 用于干扰拷贝类函数的识别, 例如 *find_p*、*util_stristr* 等函数), 以及使用真实的固件程序作为数据集. 为了判断识别的效果, 本文使用函数名来标记函数是否为拷贝类函数; 同时, 为了方便测试, 保留了二进制程序中的函数名, 但是所提方法并不依赖

© 中国科学院软件研究所 http://www.jos.org.cn

于函数名. 为了保证公平所有的方法都不去读取逆向工具识别出的函数名. 测试时, 在 IDA Pro 运行所提方法 CPSeeker 以及 3 种基准对比方法 BootStomp、SaTC、CPYFinder. 每个方法最后输出当前二进制程序下的测试效果, 即精准率 P, 召回率 R 和 $F1$ 值, 然后记录并对比各个值. 在真实的固件中, 测试各个方法是否能够成功识别出标记的拷贝类函数和漏洞函数. 此外, 为了测试基于 AI 的代码相似性方法 (Gemini) 对拷贝类函数识别的效果, 首先按照 Gemini 论文中所使用的参数和数据集训练模型, 然后从自定义实现的拷贝类函数中选取了使用 ARM 架构 O0 优化等级编译成的 8 个函数作为基准. 测试的方法为: 以这 8 个函数作为基准, 在另外的 11 个二进制程序中进行搜索 (在基准二进制程序中进行搜索, 排名总是 1. 因此, 只测试另外的 11 个二进制程序: 3 个 ARM 架构、4 个 MIPS 架构和 4 个 x86 架构), 查看当前函数所处的排名 (利用函数名来构建 ground-truth).

5.4 实验结果与分析

为了评估基于静态和动态混合分析的拷贝类函数识别方法 CPSeeker 的有效性, 本文研究了以下 4 个问题.

- RQ1: CPSeeker 是否在 C 语言库中的拷贝类函数识别上具有更好的效果 (第 5.4.1 节)?
- RQ2: CPSeeker 是否在自定义实现的拷贝类函数识别上具有更好的效果 (第 5.4.2 节)?
- RQ3: CPSeeker 的识别效果是否受编译环境的影响 (第 5.4.3–5.4.5 节)?
- RQ4: CPSeeker 在实际的固件程序中测试效果如何 (第 5.4.6 和 5.4.7 节)?

5.4.1 对 C 语言库中的拷贝类函数识别

与 CPYFinder 相同, 测试不同方法对 C 库中拷贝类函数的识别效果. 测试样本通过集成库函数中的拷贝类函数, 由不同的编译器静态编译成 ARM 架构获得. 测试结果如表 4 所示, 从表中数据可以看出 CPSeeker 的效果最好, 对库函数识别的 $F1$ 值达到了 0.96, 远好于 BootStomp、SaTC、CPYFinder. 其中: BootStomp 最好效果的 $F1$ 值为 0.67; SaTC 最好效果的 $F1$ 值为 0.42; CPYFinder 最好效果的 $F1$ 值为 0.90. 此外, 相比于其他 3 种方法, CPSeeker 不受编译器版本的影响, 在 4 种编译器版本下, $F1$ 值均为 0.96, 而其他 3 种方法均受编译器不同程度的影响. 对 CPSeeker 的结果进一步分析发现, 其漏报的原因来自于对函数参数传递错误, 例如函数 *wcsnrtombs*(char **dest*, const wchar_t ***src*, …), 其***src* 参数应传递源内存数据的地址的存放位置, 而仿真执行时传递的是源数据所在的地址, 这导致函数执行过程中将其中的数据解析为内存地址, 导致内存访问失败.

表 4 各方法对 C 语言库中拷贝类函数识别效果对比

二进制程序 (ARM)	BootStomp			SaTC			CPYFinder			CPSeeker		
	P	*R*	*F*1	*P*	*R*	*F*1	*P*	*R*	*F*1	*P*	*R*	*F*1
gcc-4.5.4-O0	1.00	0.50	0.67	0.28	0.36	0.32	0.56	1.00	0.72	1.00	0.92	0.96
gcc-4.6.4-O0	1.00	0.50	0.67	0.28	0.36	0.32	0.56	1.00	0.72	1.00	0.92	0.96
gcc-4.7.4-O0	1.00	0.43	0.60	0.44	0.29	0.35	0.81	1.00	0.90	1.00	0.92	0.96
gcc-4.8.5-O0	1.00	0.43	0.60	0.50	0.36	0.42	0.79	0.85	0.82	1.00	0.92	0.96

因此, 基于静态和动态混合分析的拷贝类函数识别方法 CPSeeker 对库函数中拷贝类的效果优于基于静态分析的方法 (BootStomp、SaTC、CPYFinder).

5.4.2 自定义拷贝类函数的识别效果

对自定义拷贝类函数的识别效果如表 5 所示. 从表中可以看出: BootStomp 对自定义实现的拷贝类函数的识别效果极差, 其 $F1$ 值出现了 0; SaTC 对自定义拷贝类函数识别的效果好于 BootStomp 的效果, 但 $F1$ 值仍不足 0.5; CPYFinder 的识别效果优于前两种方法, $F1$ 值达到了 0.78; CPSeeker 的识别效果仍是最好的, $F1$ 值为 0.92. 对于自定义的拷贝类函数的识别, CPSeeker 仍不受编译环境的影响, 其他 3 种方法仍然受编译器版本的影响. 对 CPSeeker 的识别结果进一步分析发现, 漏报的原因是由于部分函数, 例如函数 *alps_lib_toupper*(), 在内存拷贝前首先对长度进行了限制, 而预设的数据长度超过了这个限制, 导致函数无法执行内存的拷贝.

尽管 CPSeeker 对自定义实现的拷贝类函数识别的效果略低于对库中拷贝类函数识别的效果, 但是其识别效果仍远好于基于静态分析的方法 (BootStomp、SaTC、CPYFinder).

© 中国科学院软件研究所 http://www.jos.org.cn

表 5 各方法对自定义拷贝类函数识别的效果对比

二进制程序 (ARM)	BootStomp			SaTC			CPYFinder			CPSeeker		
	P	*R*	*F*1	*P*	*R*	*F*1	*P*	*R*	*F*1	*P*	*R*	*F*1
gcc-4.5.4-O0	0.00	0.00	0.00	0.64	0.29	0.40	0.62	0.45	0.52	1.00	0.86	0.92
gcc-4.6.4-O0	0.00	0.00	0.00	0.64	0.29	0.40	0.62	0.45	0.52	1.00	0.86	0.92
gcc-4.7.4-O0	0.00	0.00	0.00	0.64	0.29	0.40	0.62	0.45	0.52	1.00	0.86	0.92
gcc-4.8.5-O0	1.00	0.02	0.04	0.64	0.29	0.40	0.81	0.68	0.74	1.00	0.86	0.92
gcc-5.4.0-O0	1.00	0.02	0.04	0.61	0.29	0.39	0.81	0.68	0.74	1.00	0.86	0.92
gcc-5.5.0-O0	1.00	0.02	0.04	0.61	0.29	0.39	0.81	0.68	0.74	1.00	0.86	0.92
gcc-6.5.0-O0	1.00	0.02	0.04	0.61	0.29	0.39	0.81	0.68	0.74	1.00	0.86	0.92
gcc-7.5.0-O0	1.00	0.02	0.04	0.61	0.29	0.39	0.87	0.70	0.78	1.00	0.86	0.92

5.4.3 不同指令集及优化等级下的识别效果

尽管从第 5.4.1 节和第 5.4.2 节可以看出 CPSeeker 不受编译器版本的影响, 但是为了进一步验证 CPSeeker 是否受编译环境的影响, 对其在不同编译优化等级下的效果进行评估 (使用 4 种不同的优化等级 O0–O3). 由于当前 CPSeeker 还不支持 PPC 架构的二进制程序, 因此只评估其在 3 种架构下的效果 (x86、ARM、MIPS). 由于 BootStomp 和 SaTC 不完全支持上述指令集架构, 因此只对比 CPSeeker 与 CPYFinder 对不同架构下的拷贝类函数识别效果, 实验结果如表 6 所示. 从表 6 中可以看出, CPSeeker 支持不同指令集架构的二进制程序, 并且 CPSeeker 的识别效果优于 CPYFinder, 其中 CPSeeker 的 *F*1 值为 0.89–0.92, 而 CPYFinder 的 *F*1 值为 0.62–0.74. 对实验结果的进一步分析发现, x86 下 CPSeeker 的效果在 O0 和 O1 优化等级下效果略差的主要原因是部分具备拷贝功能的函数被函数 *main* 展开, 使得 CPSeeker 认为函数 *main*, 因此为标签标记错误导致, 如图 7 所示. 而 ARM 和 MIPS 下同样是由于编译器原因导致函数名称发生更改, 例如函数 *alps_lib_toupper* 被编译成两个函数, 其中源码中原始函数的主体在函数 *alps_lib_toupper.part.*1 中, 如图 8 所示. 此外, 为了对比另外 2 种方法受编译优化等级的影响, 制作了表 7. 从表 7 中可以看出, 另外 2 种方法均受编译优化等级的影响较大, 而且识别的效果都较差, 远低于 CPYFinder 和 CPSeeker.

表 6 不同指令集及优化等级对识别的影响 (*F*1)

二进制程序	CPYFinder			CPSeeker		
	x86	ARM	MIPS	x86	ARM	MIPS
gcc-5.4.0-O0	0.68	0.74	0.72	0.90	0.92	0.92
gcc-5.4.0-O1	0.70	0.70	0.70	0.89	0.92	0.92
gcc-5.4.0-O2	0.66	0.62	0.66	0.92	0.89	0.89
gcc-5.4.0-O3	0.68	0.64	0.68	0.92	0.92	0.92

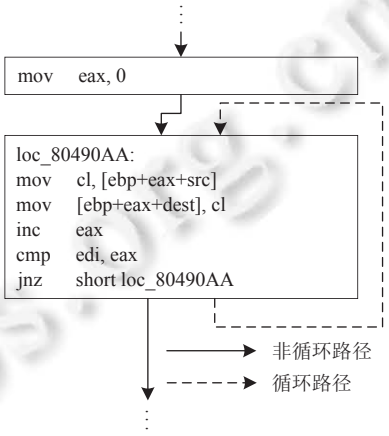


图 7 函数 *main* 由于编译器原因导致标签错误

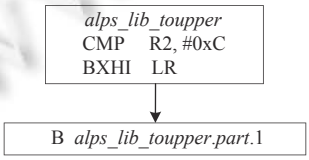


图 8 函数 *alps_lib_toupper* 被优化导致标签错误

© 中国科学院软件研究所 http://www.jos.org.cn

表 7 优化等级对各个方法识别效果的影响

二进制程序 (ARM)	BootStomp			SaTC			CPYFinder			CPSeeker		
	P	*R*	*F*1	*P*	*R*	*F*1	*P*	*R*	*F*1	*P*	*R*	*F*1
gcc-5.4.0-O0	1.00	0.03	0.06	0.61	0.30	0.40	0.68	0.81	0.74	1.00	0.86	0.92
gcc-5.4.0-O1	1.00	0.22	0.36	0.44	0.19	0.27	0.64	0.78	0.70	1.00	0.85	0.92
gcc-5.4.0-O2	0.89	0.22	0.35	0.60	0.08	0.14	0.60	0.64	0.62	0.94	0.85	0.89
gcc-5.4.0-O3	1.00	0.19	0.32	0.75	0.08	0.14	0.64	0.64	0.64	1.00	0.85	0.92

因此, 相比于 CPYFinder, CPSeeker 几乎不受指令集架构和编译优化等级的影响, CPYFinder 受指令集架构和编译优化等级的影响较大. BootStomp 和 SaTC 同样受编译器等级的影响较大.

5.4.4 不同编译器下拷贝类函数识别效果

表 8 展示了不同编译器种类和版本下, 不同方法对拷贝类函数识别的效果. 从表中可以看出 CPSeeker 不受编译器种类的影响, *F*1 值为 0.91–0.92, 而另外 3 种方法 (BootStomp, SaTC, CPYFinder) 均受编译器种类和编译版本的影响, BootStomp 的 *F*1 值为 0.23–0.45, SaTC 的 *F*1 值为 0.23–0.37, CPYFinder 的 *F*1 值为 0.68–0.79.

表 8 不同编译器下各方法的识别效果对比

二进制程序	BootStomp			SaTC			CPYFinder			CPSeeker		
	P	*R*	*F1*	*P*	*R*	*F*1	*P*	*R*	*F*1	*P*	*R*	*F*1
gcc-4.5.4-O1	1.00	0.13	0.23	0.56	0.24	0.34	0.64	0.78	0.70	1.00	0.85	0.92
gcc-4.6.4-O1	1.00	0.16	0.28	0.43	0.18	0.25	0.64	0.78	0.70	1.00	0.85	0.92
gcc-4.7.4-O1	1.00	0.18	0.31	0.42	0.16	0.23	0.64	0.78	0.70	1.00	0.85	0.92
gcc-4.8.5-O1	1.00	0.18	0.31	0.42	0.16	0.23	0.62	0.76	0.68	1.00	0.84	0.91
clang-3.9.1-O1	1.00	0.29	0.45	0.58	0.27	0.37	0.72	0.81	0.76	1.00	0.86	0.92
clang-4.0.1-O1	1.00	0.29	0.45	0.58	0.27	0.37	0.72	0.81	0.76	1.00	0.86	0.92
clang-5.0.1-O1	1.00	0.29	0.45	0.58	0.27	0.37	0.73	0.87	0.79	1.00	0.86	0.92
clang-6.0.1-O1	1.00	0.29	0.45	0.58	0.27	0.37	0.73	0.87	0.79	1.00	0.86	0.92

表 9 展示了不同方法在 clang编译器下, 对不同优化等级下的拷贝类函数识别效果. 从表中可以看出, 在 clang 编译器下, CPSeeker 仍然几乎不受编译器优化等级的影响, 而 CPYFinder 受编译器优化等级影响较小, BootStomp 和 SaTC 受编译器优化等级影响较大.

表 9 clang 编译器对拷贝类函数识别的影响

二进制程序	BootStomp			SaTC			CPYFinder			CPSeeker		
	P	*R*	*F*1	*P*	*R*	*F*1	*P*	*R*	*F*1	*P*	*R*	*F*1
clang-3.9.1-O0	0.00	0.00	0.00	0.66	0.16	0.26	0.63	0.75	0.68	0.95	0.86	0.90
clang-3.9.1-O1	1.00	0.29	0.45	0.58	0.27	0.37	0.72	0.81	0.76	1.00	0.86	0.92
clang-3.9.1-O2	1.00	0.24	0.39	0.00	0.00	0.00	0.68	0.78	0.73	1.00	0.85	0.92
clang-3.9.1-O3	1.00	0.24	0.39	0.50	0.02	0.04	0.68	0.78	0.73	1.00	0.85	0.92
clang-6.0.1-O0	1.00	0.24	0.39	0.50	0.02	0.04	0.68	0.78	0.73	0.95	0.86	0.90
clang-6.0.1-O1	0.00	0.00	0.00	0.66	0.16	0.26	0.65	0.81	0.72	1.00	0.86	0.92
clang-6.0.1-O2	1.00	0.29	0.45	0.58	0.27	0.37	0.73	0.87	0.79	1.00	0.85	0.92
clang-6.0.1-O3	1.00	0.24	0.39	0.50	0.02	0.04	0.68	0.78	0.73	1.00	0.85	0.92

因此, 基于静态和动态混合分析的拷贝类函数识别方法 CPSeeker 在不同的编译环境下 (编译器版本、编译器种类、编译优化等级) 均优于基于静态分析的方法 (BootStomp、SaTC、CPYFinder).

优势分析: 从上述实验结果可知, CPSeeker 的精准率几乎为 1.0, 召回率略低于精准率. BootStomp 与此类似, 精准率远高于召回率. SaTC 和 CPYFinder 相似, 精准率和召回率相差不大. 由 RQ1–RQ3 的结果可知, 本文所提方法在不同的场景下 (即, 不同来源、不同指令集、不同优化等级以及不同编译器) 都取得了较好的识别效果, 以上结果表明 CPSeeker 具有更好的健壮性. 基于静态分析的方法 (BootStomp、SaTC、CPYFinder) 由于采用静态的代码特征和数据流特征, 当函数复杂时, 静态分析无法获取准确的结果, 编译器的优化也会改变内存寻址的方式, 代

© 中国科学院软件研究所 http://www.jos.org.cn

码特征和数据流特征也会随之发生变化, 导致无法准确地识别拷贝类函数. 而 CPSeeker 使用模拟仿真对函数执行状态的监视, 捕获了函数执行时对内存的访问情况和指令的执行情况, 通过对内存访问轨迹和指令执行轨迹进行分析, 进而判断函数是否为拷贝类函数. 无论拷贝类函数的代码特征和数据流如何变化, 其最终本质上还是要从一段连续的内存中读取数据, 然后经过处理存放到另一段连续的内存区域中. 在函数得到完整执行的情况下, CPSeeker 能够准确地捕获到函数的执行状态信息, 进而判断函数是否为拷贝类函数. 目前, 由于在对函数的参数识别时所得的结果并不能保证每一个函数都能得到正确的执行, 因此会存在漏报的情况, 即召回率略低于精准率.

5.4.5 基于 AI 的代码相似性检测方法的效果

表 10 给出了基于 AI 的代码相似性方法的搜索的排名结果. 从排名结果来看, 基于 AI 的代码相似性检测方法 Gemini 的效果并不理想, 在所有测试中, 只有 6 个结果处于排名为 1 的位置, 其他的结果相对较差. 基于 AI 的代码相似性检测方法受编译器、优化等级的影响较大. 此外, 基于 AI 的代码相似性检测方法对于检测拷贝类函数还存在两方面的弊端: 一是必须有基准函数, 只能检测已知的拷贝类函数, 无法识别未知的函数; 二是即使函数已知, 也需要构建基准样本, 当前的方法并不能完美解决跨优化等级以及跨指令集的搜索, 为每个优化等级、每个指令集的函数均构建基准样本的工作量是巨大的.

表 10 基于代码相似性的拷贝类函数识别效果

函数名	ARM			MIPS				x86			
	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3
stpcpy	6	7	7	1	21	22	20	2	9	11	5
my_copy	44	23	22	1	8	11	8	2	45	31	24
zmemcpy	25	29	15	1	8	4	12	23	17	20	23
wxStrncpy	5	5	32	8	3	6	26	26	8	15	15
w_copy	27	28	25	2	9	9	13	10	10	12	20
strlcpy	6	6	6	4	7	6	6	8	4	2	2
StrnCpy_fn	19	24	26	5	24	24	21	13	26	30	29
alps_lib_toupper	20	23	36	1	1	4	33	6	1	18	24

5.4.6 剥离的固件中拷贝类函数识别

为了测试真实的固件中, CPSeeker 的识别效果, 选取了数据集 II 进行测试, 这里只与 CPYFinder 进行对比, 判断其是否能够对拷贝类函数进行识别. 其中在标记的 20 个拷贝类函数中, CPSeeker 能够正确地识别出 19 个, 而 CPYFinder 只能正确地识别出 14 个. 相比于 CPYFinder, CPSeeker 识别的召回率提高了 35.7%. 对 CPSeeker 的识别结果分析发现, 其中函数 *bcopy* 识别错误的原因是在执行 Hook 的时候出现了错误, 导致函数未能成功执行, 而 CPYFinder 使用静态分析的方法能够正确地识别出了该函数.

5.4.7 固件中已知漏洞函数检测

为了测试 CPSeeker 在实际固件中拷贝类函数识别的效果, 本文对数据集 III 的固件中的程序进行测试, 测试结果如表 11 所示. 由于 BootStomp 依赖于 IDA Pro 的反编译引擎, 而当前使用的 IDA Pro 无 MIPS 指令集的反编译引擎, 因此 BootStomp 无法识别, 此外由于 BootStomp 基于静态特征分析进行识别, 即使存在 MIPS 反编译引擎, 可以预见其结果并不会优于 CPYFinder. SaTC 未识别出这 6 个 CVE 漏洞函数, 由于其依赖于 angr, 在两个程序分析中执行崩溃, 未给出结果. 而 CPYFinder 识别出 4 个 CVE 漏洞函数, 其中未识别出 CVE-2017-6548 和 CVE-2018-11013 的漏洞函数, 正如其文中所述, 其在关闭特征的限制后能够识别出 CVE-2018-11013. 尽管此操作能够识别此函数, 但是同样会降低其识别的准确率. CPSeeker 识别出了其中的 5 个 CVE 漏洞函数, 而同样未识别出 CVE-2018-11013.

对其进一步分析发现: 因为该函数从入口到循环的路径上存在动态链接的函数 *nvram_bufget*(), 由于此函数为非常见的函数, 因此未对其进行功能实现, 不支持对其进行 Hook, 导致函数无法执行到循环路径, 如图 9 所示中的基本块 loc_41E980, 该基本块中多次调用函数 *nvram_bufget*() 来解析参数. 此外, 由于 CPSeeker 的动态执行依赖函数参数的赋值, 不同的参数会导致不同的执行路径, 而执行到循环所需的参数值相对严格, 目前的参数构造方法不足以执行到循环拷贝块. 因此, 后续会从两个方面着手解决: 构造函数执行相关的输入和执行更细粒度的代码片段.

© 中国科学院软件研究所 http://www.jos.org.cn

表 11 对由循环拷贝导致的溢出漏洞的识别

CVE	检测结果			
	BootStomp	SaTC	CPYFinder	CPSeeker
CVE-2017-6548	*	×	×	√
CVE-2017-13772	*	×	√	√
CVE-2018-3950	*	*	√	√
CVE-2018-3951	*	*	√	√
CVE-2018-11013	*	×	×	×
CVE-2020-8423	*	×	√	√

注: *表示不支持或者运行崩溃; ×表示未识别出; √表示识别出

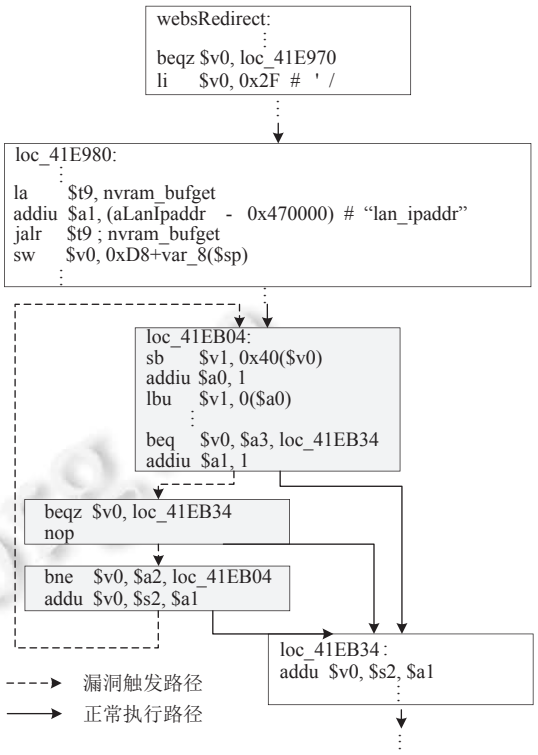


图 9 CPSeeker 和 CPYFinder 均未识别出的漏洞函数

综上所述, 基于静态和动态混合分析的拷贝类函数识别方法 CPYFinder 在实际的漏洞函数发现上, 效果同样远远好于基于静态分析的方法 (BootStomp、SaTC、CPYFinder).

5.4.8 案例分析

本节选取特定的实例来讨论分析 CPSeeker 相对于现有方法的优势以及存在的不足.

(1) SaTC 和 CPYFinder 对函数 *find_p* 的误报

函数 *find_p* 的功能是从给定的字符串中检查是否存在字符"p". 其对应的源码及汇编代码见图 10(a) 和图 11(a). 在对函数 *find_p* 进行分析时, SaTC 和 CPYFinder 均将其识别为拷贝类函数, 从源码明显可以看出其不具有内存拷贝的功能, 然后经过编译器编译后, 其变得相对复杂, 在循环路径中的特征满足了 SaTC 和 CPYFinder 的判定条件, 因此, SaTC 和 CPYFinder 均将其识别为拷贝类函数. CPSeeker 正确地识别出其非拷贝类函数, 原因是: 在静态分析时, 其满足了拷贝类函数依赖的代码特征, CPSeeker 对其进行了模拟执行, 捕获了其执行轨迹和内存访问轨迹, 经过对执行轨迹和内存访问轨迹的分析, 发现该函数对循环路径进行了多次执行, 但是只存在对内存的连续读取, 未发现对内存的连续写入. 因此, CPSeeker 将其识别为非拷贝类函数.

(2) SaTC 对函数 *wxStrcpy* 的漏报

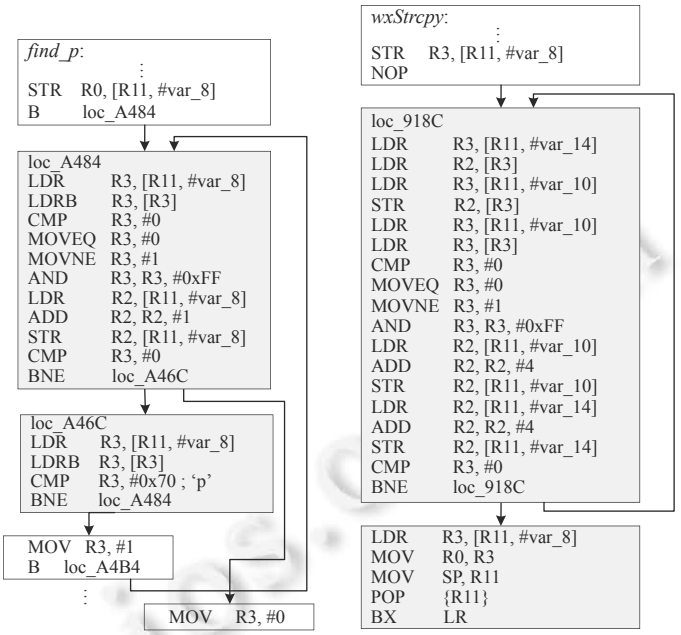
函数 *wxStrcpy* 的功能是将宽字符串复制到另一段内存区域中, 其对应的源码以及汇编代码见图 10(b) 和图 11(b). SaTC 将其识别为非拷贝类函数, 然而, 从其源码看, *wxStrcpy* 明显是拷贝类函数. CPSeeker 将其正确地识别为拷贝类函数, 原因如下: 在函数参数正常地传递下 (源地址为 0x70005000, 目的地址为 0x70003000), 该函数得到了正常地执行, CPSeeker 捕获到了执行的轨迹和所有的内存访问轨迹, 检测到该函数从 0x70005000 连续读取内存直到 0x70005044, 并将数据连续写入到 0x70003000 直到 0x70003044. 并且根据静态分析的结果识别到循环中的指令得到了全部执行, 函数执行代码覆盖率为 1.0. 因此, CPSeeker 认为此函数为拷贝类函数.

(3) CPSeeker 对函数 *StrnCpy_fn* 的漏报

函数原型为 char **StrnCpy_fn*(const char *fn, int line, char *dest, const char *src, size_t n), 从该原型可以看出该

© 中国科学院软件研究所 http://www.jos.org.cn

函数的参数传递情况比较复杂, 而 CPSeeker 未能正确地传递参数导致在执行的时候出现了非法的内存访问 (Invalid memory read: addr @0x48, size: 0x1, val: 0x0), 最终函数未能够完全执行. 因此, CPSeeker 未能够识别出该函数.



(a) 函数 *find_p* 的 CFG 及汇编代码 (b) 函数 *wxStrcpy* 的 CFG 及汇编代码

图 10 函数 *find_p* 和函数 *wxStrcpy* 的 CFG 以及汇编代码

```
BOOL find_p (char *str) {
    while (*str++ != '\0')
    {
        if (*str == 'p')
        return TRUE;
    }
    return FALSE;
}
```

(a) 函数 *find_p* 的 C 源码

```
WXDLLEXPORT wxChar * wxStrcpy (wxChar *dest,
    const wxChar *src){
  wxChar *ret = dest;
  while ((*dest++ = *src++));
  return ret;
}
```

(b) 函数 *wxStrcpy* 的 C 源码

图 11 函数 *find_p* 和函数 *wxStrcpy* 的 C 语言源码

5.5 效率分析

本节对 BootStomp、SaTC、CPYFinder 和 CPSeeker 方法识别的效率进行对比, 选取的程序为第 5.2.1 节中测试的程序. 各个方法时耗的对比结果如表 12 所示 (表中时耗比值以 CPSeeker 总时耗为基准, 其他方法与其相除所得), 从表中数据来看, CPSeeker 的耗时明显增加, 其耗时与 BootStomp 的耗时相当, 但是远多于 SaTC 和 CPYFinder 的时耗, 约为其 5 倍. 由于 CPSeeker 采用的是静态和动态混合分析的方法, 其需要对函数进行静态分析和动态的仿真执行, 并且当函数存在子函数调用时, 还需要对子函数进行执行分析. 此外, CPSeeker 在函数执行错误或者未给出结果时, 需要对函数的参数进行调整, 存在多次执行的情况, 因此耗时会明显增加. 虽然 CPSeeker 在识别拷贝类函数时的运行时间增加较大, 但是考虑其识别的效果极好, 其识别的准确性能够减少后续分析任务的效率, 因此仍具有较高的价值.

6 讨 论

源码经过编译器编译和优化, 丢失了部分语义信息, 导致对二进制代码进行分析无法完整的获取原始的信息,

© 中国科学院软件研究所 http://www.jos.org.cn

因此无论是使用代码结构特征还是数据流信息等基于静态分析的方法, 例如 Karonte、SaTC、CPYFinder 等, 都存在较多的信息损失, 识别的效果较差. 利用动态仿真执行捕获函数执行过程中的状态, 弥补了静态分析的准确率低、误报率高的不足. 本文提出的基于静态和动态混合分析的拷贝类函数识别技术 CPSeeker, 通过结合静态分析和动态仿真执行获取的信息, 对拷贝类函数进行识别, 支持多指令集架构, 并且几乎不受编译器和编译优化的影响. 尽管在运行的时间上有所增加, 但是所提方法支持对 C 语言库中的拷贝类函数和用户自定义的拷贝类函数的识别, 识别的效果优于现有的方法. 此外, 提供的函数执行框架能够应用于其他任务中, 例如结合 Fuzz 前端, 对函数进行测试, 或者利用捕获的函数执行状态信息进行其他类型函数的识别.

表 12　对拷贝类函数识别时耗对比

对比项	二进制程序	BootStomp	SaTC	CPYFinder	CPSeeker
时耗 (s)	gcc-4.5.4-O0	12.54	3.95	3.8	25.16
	gcc-4.6.4-O0	14.47	4.57	3.60	25.98
	gcc-4.7.4-O0	9.75	0.84	2.56	25.26
	gcc-4.8.5-O0	9.68	0.86	2.56	24.90
	gcc-5.5.0-O0	71.84	13.89	10.47	25.19
总时耗 (s)		118.28	24.11	22.99	126.49
时耗比值r		1.07	5.25	5.50	1

虽然基于静态和动态混合分析的方法提高了拷贝类函数识别的准确率, 但是由于动态仿真执行不能完全保证函数能够正常执行, 或者执行到特定的路径, 并且函数执行的结果极大的依赖函数的输入, 即参数的赋值, 因此, 在已知函数原型 (参数类型、参数个数) 的情况下或者函数原型为简单的模式 (目的地址、源地址、长度) 时, 识别效果更好, 正如实验评估中表现的对 C 语言库中的拷贝类函数识别效果更好. 当函数参数传递模式复杂时, 会由于静态分析对参数分析的不准确, 导致传递的参数错误, 使得函数无法正常执行, 因此对于自定义实现的拷贝类函数识别效果略差. 此外, 对于静态链接的二进制程序中的拷贝类函数识别效果较好; 对于动态链接的二进制程序中拷贝类函数识别效果较差, 因为在执行过程中会因缺失导入函数导致执行失败. 尽管已经对一些常见的库函数进行了 Hook, 实现了这些函数的功能, 但是存在一些其他的函数, 需要了解其功能后对其进行实现, 以解决执行调用失败的问题.

本文借助于 IDA Pro 实现对二进制程序的逆向分析, 利用 IDAPython 提供的 API 获取函数的信息. 由于当前的逆向分析工具并不能保证给出的结果是完全准确的, 因此, 会不可避免地对识别的结果产生影响. 例如, 在 IDA Pro 对嵌入式设备固件进行解析的时候, 会存在一部分函数无法被 IDA Pro 识别的情况, 当拷贝类函数存在于这些函数中时, 则会遗漏这些函数. 为了能够使得固件中的程序被完全解析, 本文使用了一些启发式的方法来辅助 IDA Pro 对函数的解析. 此外, 本文使用 Networkx 来对 CFG 中的循环进行识别, 当函数相对复杂的时候, 存在 Networkx 无法正常结束的情况, 因此, 本文对循环的识别设置了超时时间, 超时则自动结束.

在后续的研究中, 将尝试借助模糊测试等技术, 构造参数的输入值, 来提高函数执行的成功率以及执行代码的覆盖率. 此外, 对于动态链接的二进制程序, 则考虑更小粒度的执行, 即从函数级别 (function) 降为代码片段级别 (code snippet), 以避免从函数入口无法执行到循环块.

正如前文所述, 拷贝类函数的识别, 尤其是对剥离符号表 (缺失函数名) 的固件中拷贝类函数的识别对于下游的分析任务具有重要的价值. 基于污点分析的漏洞发现方法多以已知的拷贝类函数, 例如 *strcpy*、*memcpy* 等作为污点汇聚点 (sink), 通过函数名来识别此类的函数, 当固件中缺失函数名时, 此类方法的效果将受到限制. 本文所提方法能够识别剥离二进制中的拷贝类函数, 提高了拷贝类函数识别的效果, 因此, 本文所提方法能够提高以拷贝类函数作为 sink 的方法的漏洞发现的效果. 然而, 由于本文所提方法并不是完整的漏洞发现工具, 其依赖于其他的具体的漏洞挖掘的方法只能辅助其他工具进行漏洞挖掘, 需要与下游的工具进行进一步的结合, 为下游工具提供拷贝类函数的调用点 (污点汇聚点). 本文对当前的漏洞发现工具进行了分析和尝试, 发现它们不能很好地适用于剥离的二进制固件. 因此, 当前 CPSeeker 仍处于辅助人工进行漏洞发现和分析的阶段 (目前 CPSeeker 帮助我们

© 中国科学院软件研究所 http://www.jos.org.cn

在思科 IOS 固件中定位到了一些未公开的漏洞, 例如, C1900 15.1.4M2 版本中的漏洞函数 sub_2166C87C, sub_2166D100, sub_2166C1DC, sub_2166B3FC, sub_21669C84, sub_21669274), 在后续的研究中我们将尝试 CPSeeker 与现有的漏洞挖掘方法进行结合实现针对剥离二进制固件的自动化漏洞发现.

7 总 结

对拷贝类函数的调用错误常常引发缓冲区溢出漏洞, 对二进制程序中的拷贝类函数进行检测能够提高发现溢出漏洞的可能性, 提高下游分析任务的效率. 本文提出了一种基于静态和动态混合分析的拷贝类函数识别技术 CPSeeker, 用于对剥离二进制中的拷贝类函数识别. CPSeeker 融合静态和动态分析的优势, 支持对单个二进制函数的仿真执行来捕获运行时信息, 提高了对拷贝类函数识别的效果, 其识别效果远优于现有的方法 BootStomp、SaTC、CPYFinder 以及基于 AI 的代码相似性检测方法 Gemini. 由于 CPSeeker 需要动态执行函数, 因此其运行耗时有所增加, 但是避免了受编译环境 (编译器种类、版本、优化等级) 的影响, 对发现多种架构下路由器固件中的溢出漏洞具有重要作用.

References:

[1] Li Z, Zou DQ, Wang ZL, Jin H. Survey on static software vulnerability detection for source code. Chinese Journal of Network and Information Security, 2019, 5(1): 1–14 (in Chinese with English abstract). [doi: 10.11959/j.issn.2096-109x.2019001]

[2] Redini N, Machiry A, Das D, Fratantonio Y, Bianchi A, Gustafson E, Shoshitaishvili Y, Kruegel C, Vigna G. BootStomp: On the security of bootloaders in mobile devices. In: Proc. of the 26th USENIX Conf. on Security Symp. Vancouver: USENIX Association, 2017. 781–798. [doi: 10.5555/3241189.3241251]

[3] Wang L, He DJ, Li L, Feng XB. Sparse framework based static taint analysis optimization. Journal of Computer Research and Development, 2019, 56(3): 480–495 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2019.20180071]

[4] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, 2011, 46(3): 265–278. [doi: 10.1145/1961296.1950396]

[5] Cha SK, Avgerinos T, Rebert A, Brumley D. Unleashing mayhem on binary code. In: Proc. of the 33rd IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2012. 380–394. [doi: 10.1109/SP.2012.31]

[6] Dinesh S, Burow N, Xu DY, Payer, M. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In: Proc. of the 41st IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1497–1511. [doi: 10.1109/SP40000.2020.00009]

[7] Van Tonder R, Kotheimer J, Le Goues C. Semantic crash bucketing. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. Montpellier: ACM, 2018. 612–622. [doi: 10.1145/3238147.3238200]

[8] Mouzarani M, Sadeghiyan B, Zolfaghari M. A smart fuzzing method for detecting heap-based buffer overflow in executable codes. In: Proc. of the 21st IEEE Pacific Rim Int'l Symp. on Dependable Computing. Zhangjiajie: IEEE, 2015. 42–49. [doi: 10.1109/PRDC.2015.10]

[9] Vadayath J, Eckert M, Zeng K, Weideman N, Menon GP, Fratantonio Y, Balzarotti D, Doupé A, Bao T, Wang R, Hauser C, Shoshitaishvili Y. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 413–430.

[10] Cheng K, Li Q, Wang L, Chen Q, Zheng YW, Sun LM, Liang ZK. DTaint: Detecting the taint-style vulnerability in embedded device firmware. In: Proc. of the 48th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. Luxembourg: IEEE, 2018. 430–441. [doi: 10.1109/DSN.2018.00052]

[11] Chen LB, Wang YH, Cai QP, Zhan YF, Hu H, Linghu JQ, Hou QS, Zhang C, Duan HX, Xue Z. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In: Proc. of the 30th USENIX Security Symp. Berkeley: USENIX Association, 2021. 303–319.

[12] 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 41st IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2020. 1544–1561. [doi: 10.1109/SP40000.2020.00036]

[13] Yin XK, Lu B, Cai RJ, Zhu XY, Yang QC, Liu SL. Memory copy function identification technique with control flow and data flow analysis. Journal of Computer Research and Development, 2023, 60(2): 326–340 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.202110990]

© 中国科学院软件研究所 http://www.jos.org.cn

[14] Hex-Rays Corporation. A powerful disassembler and a versatile debugger. 2021. https://hex-rays.com/ida-pro/

[15] National Security Agency. Ghidra is a software reverse engineering (SRE) framework. 2021. https://github.com/NationalSecurityAgency/ghidra

[16] Shoshitaishvili Y, Wang RY, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng SJ, Hauser C, Kruegel C, Vigna G. Sok: (State of) the art of war: Offensive techniques in binary analysis. In: Proc. of the 37th IEEE Symp. on Security and Privacy (SP). San Jose: IEEE, 2016. 138–157. [doi: 10.1109/SP.2016.17]

[17] Hex-ray Corporation. Fast library identification and recognition technology. 2021. https://hex-rays.com/products/ida/tech/flirt/

[18] Radareorg. Radare2 official book. 2021. https://github.com/radareorg/radare2-book

[19] 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 24th ACM Conf. on Computer and Communications Security. Dallas: ACM, 2017. 363–376. [doi: 10.1145/3133956.3134018]

[20] 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 ACM/IEEE Int'l Conf. on Automated Software Engineering. Montpellier: ACM, 2018. 896–899. [doi: 10.1145/3238147.3240480]

[21] Massarelli L, Di Luna GA, Petroni F, Baldoni R, Querzoni L. 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]

[22] Li XZX, Qu Y, Yin H. PalmTree: Learning an assembly language model for instruction embedding. In: Proc. of the 28th ACM Conf. on Computer and Communications Security. New York: ACM, 2021. 3236–3251. [doi: 10.1145/3460120.3484587]

[23] Wang H, Qu WJ, Katz G, Zhu WY, Gao ZY, Qiu H, Zhuge JW, Zhang C. jTrans: Jump-aware transformer for binary code similarity detection. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: ACM, 2022. 1–13. [doi: 10.1145/3533767.3534367]

[24] Luo P, Zou DQ, Du YJ, Jin H, Liu CM, Shen JN. Static detection of real-world buffer overflow induced by loop. Computers & Security, 2020, 89: 101616. [doi: 10.1016/j.cose.2019.101616]

[25] Chen Q, Cheng K, Zheng YW, Zhu HS, Sun LM. Function-level data dependence graph and its application in static vulnerability analysis. Ruan Jian Xue Bao/Journal of Software, 2020, 31(11): 3421−3435 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5833.htm [doi: 10.13328/j.cnki.jos.005833]

[26] Software Freedom Conservancy. A generic and open source machine emulator and virtualizer. 2021. https://www.qemu.org/

[27] Wang LL, Wang BQ, Liu JG, Zhang JH, Miao QG. Study on malicious program detection based on recurrent neural network. Computer Science, 2019, 46(7): 86–90 (in Chinese with English abstract). [doi: 10.11896/j.issn.1002-137X.2019.07.013]

[28] Sha LT, Xiao F, Yang HK, Yu H, Wang RC. Vulnerability discovery method for virtualization in IaaS based on self-adapting fuzzing test. Ruan Jian Xue Bao/Journal of Software, 2018, 29(5): 1303−1317 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5498.htm [doi: 10.13328/j.cnki.jos.005498]

[29] Yu YC, Chen ZN, Gan ST, Qin XJ. Research on the technologies of security analysis technologies on the embedded device Firmware. Chinese Journal of Computers, 2021, 44(5): 859–881 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2021.00859]

[30] Unicorn. Showcase. 2022. https://www.unicorn-engine.org/showcase/

[31] Qiling Framework Team. A true instrumentable binary emulation framework. 2022. https://qiling.io/

[32] Gallopsled. CTF framework and exploit development library. 2022. https://github.com/Gallopsled/pwntools

[33] Michał Z. AFL++ overview. 2022. https://aflplus.plus/

[34] Song D, Brumley D, Yin H, Caballero J, Jager I, Kang MG, Liang ZK, Newsome J, Poosankam1 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]

[35] LLVM-admin Team. The LLVM Compiler Infrastructure. 2021. https://llvm.org/

[36] Zhang Z, Ye YP, You W, Tao GH, Lee WC, Kwon, Y, Aafer Y, Zhang XY. OSPREY: Recovery of variable and data structure via probabilistic analysis for stripped binary. In: Proc. of the 2021 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2021. 813–832. [doi: 10.1109/SP40001.2021.00051]

[37] Chua ZL, Shen SQ, Saxena P, Liang ZK. Neural nets can learn function type signatures from binaries. In: Proc. of the 26th USENIX Conf. on Security Symp. Vancouver: USENIX Association, 2017. 99–116. [doi: 10.5555/3241189.3241199]

[38] Gao H, Cheng SY, Xue YX, Zhang WM. A lightweight framework for function name reassignment based on large-scale stripped binaries. In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: ACM, 2021. 607–619. [doi: 10.1145/

© 中国科学院软件研究所 http://www.jos.org.cn

3460319.3464804]
[39] Chen QB, Lacomis J, Schwartz EJ, Le Goues C, Neubig G, Vasilescu B. Augmenting decompiler output with learned variable names and types. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 4327–4343.
[40] Yin XK, Liu L, Liu L, Liu SL. Function argument number identification in stripped binary under PPC and MIPS instruction set. Chinese Journal of Network and Information Security, 2020, 6(4): 95–103 (in Chinese with English abstract). [doi: 10.11959/j.issn.2096-109x.2020047]
[41] Hagberg AA, Schult DA, Swart PJ. Exploring network structure, dynamics, and function using networkx. In: Proc. of the 7th Python in Science Conf. Los Alamos: Los Alamos National Lab., 2008. 11–15.
[42] Netgear. Netgear GPL code. 2021. https://github.com/jameshilliard/R7000
[43] Senpai A. Leaked mirai source code for research/IoC development purposes. 2021. https://github.com/jgamblin/Mirai-Source-Code

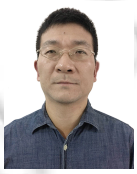
附中文参考文献:

[1] 李珍, 邹德清, 王泽丽, 金海. 面向源代码的软件漏洞静态检测综述. 网络与信息安全学报, 2019, 5(1): 1–14. [doi: 10.11959/j.issn.2096-109x.2019001]
[3] 王蕾, 何冬杰, 李炼, 冯晓兵. 基于稀疏框架的静态污点分析优化技术. 计算机研究与发展, 2019, 56(3): 480–495. [doi: 10.7544/issn1000-1239.2019.20180071]
[13] 尹小康, 芦斌, 蔡瑞杰, 朱肖雅, 杨启超, 刘胜利. 基于控制流和数据流分析的内存拷贝类函数识别技术. 计算机研究与发展, 2023, 60(2): 326–340. [doi: 10.7544/issn1000-1239.202110990]
[25] 陈千, 程凯, 郑尧文, 朱红松, 孙利民. 函数级数据依赖图及其在静态脆弱性分析中的应用. 软件学报, 2020, 31(11): 3421–3435. http://www.jos.org.cn/1000-9825/5833.htm [doi: 10.13328/j.cnki.jos.005833]
[27] 王乐乐, 汪斌强, 刘建港, 张建辉, 苗启广. 基于递归神经网络的恶意程序检测研究. 计算机科学, 2019, 46(7): 86–90. [doi: 10.11896/j.issn.1002-137X.2019.07.013]
[28] 沙乐天, 肖甫, 杨红柯, 喻辉, 王汝传. 基于自适应模糊测试的IaaS层漏洞挖掘方法. 软件学报, 2018, 29(5): 1303–1317. http://www.jos.org.cn/1000-9825/5498.htm [doi: 10.13328/j.cnki.jos.005498]
[29] 于颖超, 陈左宁, 甘水滔, 秦晓军. 嵌入式设备固件安全分析技术研究. 计算机学报, 2021, 44(5): 859–881. [doi: 10.11897/SP.J.1016.2021.00859]
[40] 尹小康, 刘鎏, 刘龙, 刘胜利. PPC和MIPS指令集下二进制代码中函数参数个数的识别方法. 网络与信息安全学报, 2020, 6(4): 95–103. [doi: 10.11959/j.issn.2096-109x.2020047]

尹小康(1993－), 男, 博士, 讲师, 主要研究领域为网络安全, 二进制代码分析, 机器学习.

杨启超(1992－), 男, 讲师, 主要研究领域为网络对抗, 网络安全.

蔡瑞杰(1990－), 男, 讲师, 主要研究领域为网络安全, 二进制代码分析, 漏洞挖掘.



刘胜利(1973－), 男, 博士, 教授, 博士生导师, 主要研究领域为网络攻击检测, 网络基础设施安全.

© 中国科学院软件研究所 http://www.jos.org.cn