二进制程序静态分析技术研究综述

程 凯[1,2], 宋站威[1,2], 刘明东[2], 于 楠[2], 朱红松[1,2], 孙利民[1,2]

[1] 中国科学院大学网络空间安全学院 北京 中国 100049

[2] 中国科学院信息工程研究所物联网信息安全技术北京市重点实验室 北京 中国 100093

摘要 随着物联网技术的兴起, 物联网设备固件扮演越来越重要的作用。不幸的是, 物联网设备固件在现实世界中仍然存在许多漏洞。近年来, 物联网设备固件的安全性受到了广泛关注。对物联网设备固件进行安全测试主要包括静态分析和动态分析两种基本方法。最近, 物联网设备固件的自动化动态分析取得了实质性进展。然而, 一方面, 由于固件的环境依赖关系复杂等, 现有的动态分析技术仍然相当有限; 另一方面, 物联网设备数量的快速增长也使得动态分析难以适应大规模估计分析。与动态分析相比, 静态分析在不实际执行固件代码的情况下分析代码, 因此对于测试大规模物联网设备固件来说, 静态分析是一种更实用、更经济的选择。虽然物联网设备的快速增长对二进制静态分析提出了新的需求, 但二进制静态分析本身也面临着诸多挑战。与发展非常成熟的源代码静态分析技术相比, 二进制静态分析发展缓慢。主要是因为二进制程序丢失了变量的符号名、数据类型和数据结构信息, 这使得现有的基于源代码的数据流分析和指向分析技术无法直接复用于二进制分析。例如基于访问路径的按需别名分析技术。此外, 设备固件中指令架构的多样性和大量的间接调用也给二进制静态分析带来了新的挑战。针对这些挑战, 研究者们提出了多种二进制静态分析技术。本文以静态分析的基本原理为基础, 从数据流分析、别名分析、符号执行和静态污点分析 4 个方面介绍目前二进制静态分析技术的研究现状和进展。最后, 本文对今后该领域的研究重点和方向进行讨论和展望。

关键词 二进制程序静态分析; 数据流分析; 别名分析; 符号执行; 静态污点分析; 物联网设备

中图法分类号 TP311 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2026.01.18

A Survey of Static Analysis Techniques of Binary Code

CHENG Kai[1,2], SONG Zhanwei[1,2], LIU Mingdong[2], YU Nan[2], ZHU Hongsong[1,2], SUN Limin[1,2]

[1] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

[2] Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China

Abstract With the emerging of the Internet of Things (IoT) technologies, IoT device firmware is nowadays playing an increasingly important role. Unfortunately, IoT device firmware still suffers from a number of vulnerabilities in the real world. Recently, the security of IoT device firmware has gained widespread attention. To test the real-world IoT device firmware, static analysis and dynamic analysis are two basic approaches. Recently, substantial progress has been made on automated dynamic analysis of IoT device firmware. However, on the one hand, existing dynamic analysis techniques are still quite limited due to reasons such as complex environment dependencies of the firmware. On the other hand, the rapid growth of the number of IoT devices also makes dynamic analysis difficult to adapt to large-scale analysis. Compared to dynamic analysis, static analysis tests the code without actually executing it, and thus is a more practical and economical option for testing large-scale IoT device firmware. Although the rapid growth of IoT devices brings new demands for binary static analysis, binary static analysis itself has many challenges. Binary static analysis is slow to develop compared to the very mature static analysis techniques for source code. This is mainly because binary programs lose symbol names, data types, and data structure information of variables, which makes the existing source-based data-flow analysis and pointer-to analysis techniques unable to be reused directly in binary analysis. An example is on-demand alias analysis based on access path. In addition, the diversity of instruction architectures in device firmware and thehe large number of indirect calls also bring new challenges to binary static analysis. To address these challenges, researchers have proposed a variety of binary static analysis techniques. Based on the basic principles of static analysis, we will introduce and summarize the binary static analysis techniques from aspects of data-flow analysis, alias analysis, symbolic execution, and static taint analysis. Finally, we will discuss the research focus and direction in the future.

Key words binary static analysis; data-flow analysis; alias analysis; symbolic execution; static taint analysis; Internet of Things devices

通讯作者: 孙利民, 博士, 研究员, Email:sunlimin@iie.ac.cn。

本课题得到广东省重点研发计划(No. 2019B010137004), 国家自然科学基金联合基金项目(No. U1766215)资助。

收稿日期: 2020-12-09; 修改日期: 2021-02-22; 定稿日期: 2023-02-21

1 引言

源代码静态分析技术在过去几十年已近发展得相当成熟, 随着源代码的复杂性和代码量的增加, 源代码静态分析算法也越来越高效, 然而二进制程序静态分析技术发展缓慢。物联网(Internet of Things, IoT)技术的迅速崛起给二进制程序静态分析带来新的挑战和需求。根据 Statista 的研究报告, 连接到互联网世界的 IoT 设备数量在 2025 年将超过 215 亿[1], 包括网络摄像机、可穿戴设备、智能家居、智慧城市、智能汽车等终端设备。在万物互联的浪潮下, 越来越多的物联网设备开始接入网络, 在提供便利的同时, 其安全问题也日益凸显: 针对各种物联网设备的网络攻击事件层出不穷, 部分设备被频繁曝出存在漏洞或后门。2014 年, 中央电视台报道了多起黑客利用家庭监控设备的漏洞将视频泄露至网上的事件。2015 年 2 月, 主营安防产品的海康威视遭遇“黑天鹅”事件, 其生产的监控设备被曝出存在严重的安全隐患, 部分设备已被境外 IP 控制。2016 年 10 月 21 日, 美国域名服务器管理服务商 Dyn 遭遇了大规模的 DDoS 攻击, 导致美国东海岸网络大瘫痪。与传统的 PC 机系统不同, 嵌入式设备的系统通常由一个称作为“固件”的软件组成。固件, 通俗的理解就是“固化的软件”, 是指写入可编程只读存储器如 EEPROM 或 FLASH 中的软件程序。与传统软件类似, 嵌入式设备固件中也存在安全缺陷或漏洞。对嵌入式设备而言, 固件就是其“大脑”, 由于固件代码具备高权限、漏洞利用代码隐蔽性高、漏洞修复周期长等特点, 固件中隐藏的缺陷或漏洞会带来严重的安全隐患。

在针对 PC 系统以及 Android 系统的程序漏洞挖掘技术方面, 数据流分析、污点分析、符号执行、模糊测试、补丁比对等技术都有较快的发展。但是在 IoT 设备固件漏洞挖掘方面, 由于 IoT 设备具有多样的指令架构、不同的操作系统(例如 Linux、RTOS 等)、硬件资源受限、大多数基于 C 语言开发以及封闭的源代码和文档等原因, 导致源代码分析技术、针对 Android App 的分析技术以及针对 PC 上 x86 程序的动态分析技术都不适合于 IoT 设备固件的安全分析。虽然 IoT 设备的诸多特点使得通用平台的漏洞挖掘技术和工具无法直接适用于 IoT 设备, 但是漏洞挖掘技术的核心原理和思路是相通的。

近几年, IoT 设备固件仿真和基于仿真的动态分析技术成为研究热点之一。固件仿真使得传统的动态分析技术, 包括混合符号执行、动态污点分析和模糊测试等技术可以有效地用于 IoT 设备固件漏洞挖掘。但是, 一方面由于 IoT 设备存在外设复杂、与外设 I/O 交互频繁等特点, 导致现有的 IoT 设备固件仿真技术仍然存在诸多限制。另一方面, 基于设备固件仿真的动态分析很难实现对海量的 IoT 设备进行规模化的安全分析。相比动态分析技术, 静态分析技术存在以下优势: ①代码覆盖率高; ②不依赖于具体的输入; ③适合自动化、可扩展性高。这些优势让静态分析技术更加适合于大规模地分析海量的 IoT 设备, 而无需在意 IoT 设备外设复杂等问题。然而, 无论是针对传统 PC 还是 IoT 设备, 二进制程序静态分析技术仍处于起步阶段。但是 IoT 设备的广泛应用正在逐渐改变这一现状, 驱动更多二进制程序静态分析技术的更新。

因此, 本篇论文主要聚焦于二进制程序静态分析技术的研究现状与发展。论文第 2 节简述二进制静态分析的基础; 第 3 节描述通用静态分析技术的相关概念和基本原理; 第 4 节论述二进制程序静态分析技术所面临的挑战与需求; 之后, 第 5~7 节分别介绍二进制程序的数据流分析与别名分析技术、符号执行技术以及静态污点分析技术; 第 8 节介绍静态分析技术在二进制程序漏洞挖掘方面的应用以及对现有二进制程序静态分析工具的总结; 第 9 节总结全文并展望。

2 二进制静态分析基础

在介绍二进制静态分析技术之前, 本节首先介绍反汇编、控制图和函数调用图这 3 个基础的关键任务。在二进制静态分析中, 它们是数据流分析、别名分析、符号执行以及污点分析的核心组成部分。

2.1 二进制反汇编

二进制反汇编是指将机器码转换成汇编语言。对于商用现成软件(COTS)来说源代码通常是不可获取的。为了分析闭源软件的安全性, 在研究人员和开发人员的共同努力下, 二进制反汇编技术在过去十年中取得了突破性的进展, 包括大量开源的框架和工具[2-11]以及商业收费工具[12-13]。IDA Pro 是其中使用最广泛和能力最强的工具之一[14], 它支持 x86/x64、ARM/ARM64、MIPS/MIPS64、PowerPC 等多种架构。

虽然二进制反汇编技术发展非常成熟, 但仍然存在一些难点: (1)如何准确区分代码字节和非代码字节, 例如只读数据和填充字节等; (2)如何正确识别重叠的指令, 即多条指令共享某些字节; (3)如何正确识别由编译器或由开发人员引入的指令。已有相关

工作[15-16]针对这些难点进行总结和分析。

架构的多样性给二进制静态分析技术的实现带来了困难。不同架构的指令集、指令语义、函数调用约定(默认传参方式和返回值寄存器)等存在明显的差异, 导致基于某一架构实现的静态分析技术很难扩展到另一架构。为了克服该困难, 不少工作提出将不同架构的反汇编代码转换成统一的中间表示, 包括 LLVM IR[17]、VEX IR[18]、REIL IR[19]、BAP IR[3]等。基于中间表示, 分析人员能够忽略底层汇编语言的差异从而实现架构无关分析。

2.2 控制流图和函数调用图

控制流图(Control Flow Graph, CFG)是一个有向图, 其中节点表示基本块, 通过节点之间的有向边来描述程序的执行流程。基本块是连续的指令序列, 它只有唯一的入口和唯一的出口, 即中间不存在其他指令会跳转离开这段指令序列。函数调用图(Function Call Graph, FCG)描述函数调用之间的关系, 其每个节点表示一个函数, 节点之间的有向边表示函数调用关系。

准确的控制流图和函数调用图恢复需要解决函数边界识别、直接控制流转移、间接跳转/调用、尾调用、无返回值函数等诸多问题。针对这些问题, Pang 等[16]在 3799 个 x86/x64 二进制上对 9 个开源流行的二进制逆向工具[9-10,20-26]和 2 个商业付费工具[12-13]进行不同维度的评估。总体来说, 商业收费工具在控制流图恢复上表现更好。对于函数调用图恢复, 其难点在于间接调用。由于间接调用的目标函数是动态计算的, 静态分析很难准确计算[15]。大部分工具只解决了间接跳转, 只有少部分工具 ANGR[25]、GHIDRA[9]、IDA Pro[13]和 BINARY NINJA[12]通过常量传播的方式计算间接调用的目标函数。然而, 对于大多数的间接调用点, 他们只能计算出一个目标函数。不同的工具计算间接调用的能力差距较大。例如, ANGR 解决 43 个间接调用点, 并且都只找到一个目标函数; 而 GHIDRA 针对 88078 个间接调用找到了不完整的目标函数集, 但其中 87947 个只找到一个目标函数。

总而言之, 目前无论是在汇编指令发现、函数边界识别、控制流图和函数调用图恢复上都存在一些没有完全解决的问题。特别是在函数调用图恢复中, 间接调用一直都是静态分析的痛点。目前通过二进制静态分析解决间接调用问题的相关工作很少[27-28]。

3 静态分析技术简介

程序静态分析是指在不运行代码的前提下, 通过词法分析、语法分析、数据流分析、别名分析、污点分析、符号执行等技术来实现代码优化、语法检测、安全验证、漏洞发现等众多应用的一种代码分析技术。其中, 词法分析和语法分析则只是用于源代码分析, 在二进制静态分析中, 数据流分析、别名分析、污点分析和符号执行是最常用的静态分析技术。本节首先介绍与上述 4 种静态分析技术相关的一些基本概念, 然后分别介绍这 4 种技术的基本原理与方法。

3.1 基本概念

流敏感: 流敏感分析考虑程序中语句的执行顺序; 而流不敏感分析忽略语句的执行顺序。例如, 在指针分析中流敏感在不同的程序点维护不同的分析结果; 而流不敏感只需要在整个分析过程维护一个单一的指向关系。流敏感分析更加精确, 但是需要更多的时间和内存开销。

上下文敏感: 上下文敏感的过程间分析算法区分一个共同被调用函数的不同的调用上下文, 以便一个调用方携带的信息不会泄露到另一个调用方。上下文敏感分析更加精确, 但通常更慢以及实现更加复杂。

路径敏感: 路径敏感分析根据条件分支语句计算不同路径的信息。路径不敏感分析通常会在分支合并点合并不同路径的分析信息; 而路径敏感分析在分支合并点会保存来自不同路径的分析信息, 需要更多的资源开销以及要考虑路径爆炸问题。

域敏感: 域敏感分析将对象实例的不同的字段当作独立的变量。在高级程序语言中, 可以通过变量的符号名来区分不同的字段变量; 然而, 由于汇编代码中的变量符号名、数据结构等语义信息丢失, 实现域敏感算法更加困难。

对象敏感: 对象敏感分析能区分不同的对象实例。对象敏感分析的难点是如何区分堆对象, 常用的方法是通过到达堆分配程序点的调用链上下文深度(k-object-sensitive)来实现堆对象敏感。

3.2 相关静态分析技术介绍

3.2.1 数据流分析

数据流分析技术通过静态代码分析来获取有关数据如何沿着程序执行路径流动的相关信息。数据流分析技术已经发展几十年, 被广泛用于编译优化、程序验证和程序安全分析等领域。

基本原理: 数据流分析可以简单地概括为一种收集程序中变量的使用、定义及其之间的依赖关系等信息的过程[29]。前向数据流分析的基本原理可以描述为如下的方程形式:

$$in(B) := \bigcup_{B' \in pred(B)} out(B'),$$

$$out(B) := gen(B) \bigcup (in(B) - kill(B)),$$

其中, B 代表基本块, $in(B)$ 代表基本块入口点的输入状态, 包括变量的定义、使用等信息, $out(B)$ 代表基本块结束点的输出状态。每个基本块的输入状态是其所有前继节点 B' 的输出状态的并集; 而输出状态通过转移函数计算基本块中每条语句的作用来获取, 包括在基本块中产生并到达基本块末尾的信息 $gen(B)$, 以及在基本块入口点携带的并且没有被杀死的信息 $in(B)-kill(B)$; 后向数据流分析具有和前向数据流分析相反的过程, 基本块的结束点作为输入状态, 入口点作为输出状态, 转移函数通过遍历基本块内语句的反向序列来获取信息; 后向数据流分析一般通过后序遍历函数的控制流图来传播信息, 而前向数据流分析则是通过后序遍历控制流图的逆序列来传播信息。

数据流问题主要分为(但不限于)两类: 位向量(Bit Vector)问题和 IFDS(Interprocedural, Finite, Distributive, Subset problems)问题。

基于位向量的数据流分析: 位向量问题是指数据流值的集合能表示为位向量, 也被称作 gen-kill 问题或局部可分问题[30]。常见的可达定义分析、实时变量分析、可用表达式分析和定义-使用链分析等都属于位向量问题。

1) 可达定义分析(reaching definitions analysis): 对于在程序点 p 处的定义 d, 如果存在一条从 p 到 q 的路径并且在该路径上定义 d 没有被杀死, 则定义 d 可到达程序点 q。该分析属于前向数据流分析问题, 其中每个变量的定义是一个比特位, 在每个程序点都有一个向量, 表示在该程序点哪些定义是可达的。

2) 实时变量分析(live variable analysis): 对于程序点 p 处定义的变量 v, 在 CFG 中, 确定从程序点 p 开始沿着某些路径上变量 v 的值是否被使用, 如果是, 则称变量 v 在程序点 p 上存活。该分析属于后向数据流分析问题, 其中每个变量的使用对应一个比特位, 在每个程序点都有一个向量, 表示在该程序点哪些变量是活跃的。

3) 可用表达式分析(avaliable expression analysis): 在程序点 p 处表达式可用当且仅当从该表达式出现的位置到程序点 p 的所有路径上不再修改该表达式的操作数。可用表达式分析主要用于编译器优化, 属于前向数据流问题, 其中每个表达式的定义对应一个比特位, 在每个程序点都有一个向量, 表示在该程序点哪些表达式是可用的。

4) 定义-使用链和使用-定义链分析(def-use and use-def chain): 定义-使用链是指将定义 d 链接到该定义可以达到的所有使用 u。而使用-定义链是指将变量 v 的使用链接到所有对变量 v 的定义 d, 并且这些定义 d 都是可以达到变量 v 的。

基于 IFDS 的数据流分析: 上述提到的位向量问题是早期针对某一类问题的描述, 即可通过位向量来表示数据流值的集合。然而还有很多数据流分析问题不属于位向量问题, 例如可能未初始化变量检测、拷贝常量传播分析等, 导致这类问题无法通过位向量算法来实现。因此 Reps 等[30]在 1995 年提出 IFDS 问题。IFDS 问题是指满足过程间分析、分析的域是有限的、数据流函数满足分配率以及是一个子集类型这 4 点的一类问题。每个位向量问题也是一个 IFDS 问题, 而有些 IFDS 问题却不是位向量问题。所有 IFDS 问题都可以通过将其转换为一种特殊的图可达性问题来精确解决。针对 IFDS 问题, Reps 等[30]提出基于上下文无关文法的可实现的路径可达性算法(realizable-path reachability algorithm)。该算法不仅能更加高效地实现各种位向量问题分析, 还能实现其他更加复杂的数据流问题分析, 例如可能未初始化变量检测等。

针对上述两类数据流问题, 在源代码分析中已经有大量工作实现有效的数据流分析算法。在二进制程序数据流分析中, 由于变量的符号名和数据结构等语义信息丢失, 现有研究工作主要是基于已有的数据流分析算法, 解决间接内存访问之间的数据流跟踪。本文将在第 5 节介绍相关工作。

3.2.2 别名分析

别名指的是两个指向同一内存位置的指针。别名分析是编译器理论中的一种技术, 用于确定是否可以通过多个指针访问同一个内存位置。别名信息对于许多应用程序至关重要, 包括动态间接调用目标地址的静态解析、副作用分析、数据流分析、程序切片和编译器优化等。

基本原理: 别名分析通过指针赋值语句来识别别名关系, 其中别名关系的描述包括两种: 别名对和指向集。别名对是指通过一对变量表示两个互为别名的变量, 例如, 赋值语句 p=&x, x=&y, 其中*p 和 x 是别名, *x 和 y 是别名, 其别名关系可以描述为 $\{<*p, x>, <*x, y>, <**p, y>\}$。该表示方式的缺陷是存在冗余信息并且随着程序的复杂别名对集合的大小会快速增长。指向集是指一个指针和其所指向的对象的集合, 例如, 赋值语句 p=&x, x=&y, 其包含的别名关系可以描述为 $p \to \{x, y\}$。别名分析方法通

过判断两个指针的指向集是否存在交集来查询两个指针是否可能互为别名。相比别名对, 指向集消耗更少的内存资源。

通常情况下, 术语别名分析(alias analysis)和指向分析(points-to analysis)具有等价概念。本论文跟随相关工作[31]区分别名分析和指向分析。其中, 别名分析的目标是计算程序中变量之间的可能别名关系, 而指向分析的目标是计算程序中变量之间的指向关系。当通过指向分析查询两个间接内存访问*p 和*q之间的别名关系时, 标准方法是计算指针 p 和 q 的指向集, 通过判断指向集交集来回答*p 和*q 是否存在别名关系。然而, 在一些情况下并不需要计算两个指针的指向集来判断它们是否互为别名。例如, 如果存在赋值语句p=q, 则表达式*p和*q是别名, 而不用考虑它们的指向集。因此, 指向分析属于别名分析中的一类方法。

别名分析技术按照实现粒度可以分为流敏感和上下文敏感。用于别名分析的现有算法主要针对高级程序语言(例如 C/C++、Java 等)。开销最低且最不精确的别名分析算法是流不敏感和上下文不敏感方法[32-36], 其中包括著名的 Andersen 别名分析算法[33]和 Steensgaard 别名分析算法[34]。与此相反, 流敏感和上下文敏感方法[3741]是最精确的, 但同时也是时间和内存开销最大的。而二进制程序的别名分析工作不是太多, 本文将在第 5 节介绍相关工作。

3.2.3 污点分析

污点分析技术是信息流分析技术的一种实践方法[42], 通过跟踪并分析污点信息在程序中流动的技术, 在信息泄露检测、漏洞挖掘、恶意代码检测等方面有广泛的应用。

基本原理: 污点分析通过跟踪程序中变量的数据流来检查污点数据能否在不经过安全检查的前提下沿着程序的执行路径从污点源传播到污点汇聚点。污点分析可以抽象成一个三元组<sources, sinks, sanitizers>, 其中, sources表示污点源, 代表可引入被攻击者控制的输入数据, 例如网络数据接收函数recv; sinks表示污点汇聚点, 代表安全敏感操作, 包括不安全的内存拷贝(例如strcpy函数)、泄漏隐私数据到外界等; sanitizers表示无害处理, 即对可控输入数据进行安全检查, 包括字符串长度检查、数据加密等手段使数据传播不再对软件系统的信息安全产生危害[42]。因此, 通用的污点分析技术包括3个步骤: ①识别程序中的污点源和污点汇聚点; ②基于定制的污染策略传播污点; ③无害处理检查。

当污点分析应用到不同的场景时, 污点源和汇聚点的识别、污染策略以及无害处理检查的方式都各不相同, 其原因主要包括目标程序编程语言的差异(例如PHP、Java、C/C++等)以及针对的安全漏洞类型不同。在C语言编程的二进制代码分析中, 当通过污点分析检测缓冲区溢出漏洞、格式化字符串漏洞、命令注入漏洞时, 表1展示通用的污点源以及污点汇聚点。

表 1 C 二进制分析中常用的污点源和汇聚点

Table 1 the common sources and sinks in C binary

污点源	recv, recvfrom, read, fread, fgets
污点汇聚点	strcpy, strncpy, memcpy, memmove, sprintf, snprintf, sscanf, strcat, strncat, system, popen, execve

污点传播分析是基于定制的污染策略确定污点数据如何传播, 哪些操作类型会引入新的污点或消除污点标记, 以及对污点数据执行哪些检查。污点传播分析包括显示流分析和隐式流分析。其中, 显示流分析是指污点标记如何随着程序中变量之间的数据依赖关系传播, 而隐式流分析是指污点标记如何随着程序中变量之间的控制依赖关系传播[42]。污点传播分析一般存在两个问题, 即过污染(over-taint)和欠污染(under-taint)。其中, 过污染是指在污点分析过程中将不是从污点源派生出来的数据标记为污点, 即产生误报; 欠污染则是在污点分析过程中丢失从污点源到污点汇聚点的污点信息流, 即产生漏报。污点分析技术包括静态污点分析和动态污点分析。本文主要介绍静态污点分析技术。静态污点分析是指在不运行代码的前提下, 通过跟踪程序中变量的数据流来检查污点数据能否沿着程序的执行路径从污点源传播到污点汇聚点。在静态污点分析中, 常见的污点传播方式包括直接赋值传播、过程间传播以及别名传播。然而, 二进制静态污点分析工作相当少, 本文将在第7节介绍相关工作。

3.2.4 符号执行

符号执行作为静态分析中的关键技术, 其核心思想是用符号值替代输入的具体值来模拟执行程序。

基本原理: 在符号执行过程中, 对于程序输入或是无法确定的变量用符号值表示。初始的符号值没有任何约束, 当程序遇到分支跳转时, 会生成对符号值的约束, 并将约束保存到当前路径的约束集中。纯粹的静态符号执行会在每个条件分支点生成新的执行器用于分别探索不同的分支路径。然后, 符号执行通过约束求解器来验证每条路径约束的可解性。如果该路径约束可解, 则说明该路径是可达的,

可通过求解约束生成相应的测试用例。理论上来讲, 符号执行可以覆盖程序的所有路径, 并且针对每条路径都生成满足该路径约束的测试用例[43-44]。

随着程序越来越复杂, 纯粹的静态符号执行应用于真实的大型程序分析遇到两个主要问题: (1)路径爆炸, 即路径随着条件分支个数呈指数增长; (2)存在无法求解的符号路径约束。为了缓解上述两个问题, 后续工作提出混合执行测试(concolic testing)[45-46]、执行生成测试(execution-generated testing)[47]、选择性符号执行(selective symbolic execution)[48]以及欠约束符号执行(under-constraint symbolic execution)[49]。本文简单概述不同的符号执行方法。

1) 混合执行测试: 混合执行测试是一种结合符号执行和具体执行的技术, 其主要思想是: 首先, 系统在选定或随机种子输入下具体地执行一条路径并记录执行的指令流; 其次, 通过将输入符号化, 符号地执行记录的指令流, 并生成路径的符号约束; 最后, 通过对分支条件进行取反生成不同的测试用例, 并使用新的测试用例再次进行具体执行, 反复上述过程直到所有的路径都被探索。该方法的缺陷是每次探索新的路径都需要重复执行大量的指令。

2) 执行生成测试: 执行生成测试也是一种结合符号执行和具体执行的技术, 与混合执行测试最主要的不同点在于混合的方式不一样。执行生成测试以符号值作为程序输入, 在程序执行中监控每条语句, 如果语句涉及符号值, 则通过符号执行; 否则, 通过具体执行来正常运行该语句。如果条件分支包含符号值, 则生成新的执行器, 在符号约束为真的条件下执行真分支, 在符号约束为假的条件下执行假分支。最后, 通过约束求解器验证路径约束并求解生成对应的测试用例。

3) 选择性符号执行: 选择性符号执行通过在符号执行和具体执行之间无缝来回切换, 同时维持系统状态在符号和具体状态之间的转换来实现对真实大型程序的分析。选择性符号执行可以指定任意的代码段或内存区域进行符号分析, 而其他的则采用本地的具体执行来提高效率。选择性符号执行极大地提高了符号执行在实际应用中对大型软件安全测试的可用性, 最具代表性的是 S2E 框架[48]。

4) 欠约束符号执行: 欠约束符号执行通过直接分析单个函数而不是整个程序来提高符号执行的可扩展性。所谓欠约束是指在分析单个函数时, 其输入不受函数上下文约束, 而是通过惰性初始模型(lazy initialization)来生成函数的符号输入[49]。

4 面临的挑战与需求

二进制程序的安全测试技术包括静态分析技术和动态分析技术。相比动态分析, 静态分析的优势包括: ①代码覆盖率高; ②不依赖于具体的输入; ③适合自动化, 可扩展性高。虽然静态分析具有这 3 个明显的优势, 但是二进制程序安全分析工作聚焦于动态分析, 包括动态污点分析、模糊测试等, 其中模糊测试工作占主要部分。静态分析则主要集中于源代码分析, 而二进制程序静态分析技术的研究工作不是太多。

4.1 二进制静态分析的挑战

无论是源代码静态分析技术还是二进制静态分析技术, 包括数据流分析、污点分析、符号执行, 都面临两个关键的挑战: ①指针别名问题; ②间接调用问题。

指针别名问题: 在源代码分析中, 别名分析技术已经非常成熟, 包括流不敏感和上下文不敏感的别名分析方法[32-36]、流敏感和上下文敏感的别名分析方法[37-41], 以及按需别名分析方法[50-54]。然而不幸的是, 二进制程序别名分析工作非常少, 目前只能找到有限的相关文献[55-60]。主要有 2 个原因: ①传统的针对 x86 的二进制程序分析主要以动态分析为主, 而动态分析不需要考虑别名的影响; ②在二进制程序静态分析中, 很难直接从机器码直接反编译成源代码(C/C++)。反汇编工具 IDA Pro 提供从汇编代码反编译成类 C 源代码的功能, 但无法恢复变量类型和复杂的数据结构, 例如链表、嵌套数据结构等。因此, 现有的二进制程序静态分析要么直接分析反汇编代码, 要么通过将反汇编代码转换成中间表示之后再进行程序分析。但是由于二进制程序中变量的符号名和数据结构等语义信息丢失, 很多高级程序语言中的别名分析算法无法直接适用于二进制程序。

在二进制静态分析中, 别名分析工作的缺失也间接导致无法实现高精度和高效率的数据流分析和静态污点分析。例如最为著名的 Android 静态污点分析框架之一 FlowDroid[61], 其通过 IFDS 算法和按需别名分析算法实现流敏感、上下文敏感、域敏感和对象敏感的静态污点分析方法。而其中的 IFDS 算法和按需别名分析算法都是基于访问路径(access path)实现的。访问路径是一个基变量后跟有限的字段访问序列, 例如多层指针变量 $x.y.z$。然而, 基于访问路径的方法不太适合低级程序语言分析, 因为变量在低级程序语言上没有符号名并且不容易区分。

间接调用问题: 完整的过程间控制流图恢复是静态分析的基础。然而, 间接调用的存在导致静态分析很难生成完整的过程间控制流图。因为间接调用的目标函数地址是程序运行时动态决定的。因此, 静态分析通常只能计算出间接调用的一个超近似的目标函数地址集合。目标函数地址集合越小, 代表恢复的间接调用越精确, 从而基于过程间控制流图实现的数据流分析和污点分析的误报率越低。

在二进制静态分析中, 大多数相关工作集中于解决间接跳转[15,62-63](例如 C 语言中 switch 语句造成的多分支跳转)或简单的过程内间接调用恢复[64]。而很少有相关工作能解决复杂情形, 例如涉及过程间的回调函数、通过索引函数表的间接调用以及 C++ 中的虚函数调用。van der Veen 等[27]通过函数参数个数和类型匹配的方法减少间接调用的目标地址集合, 该方法主要目的是用于缓解代码重用攻击, 而不适合于数据流分析和污点分析。因为这种方法找到的间接调用目标地址中存在大量误报。

多样的指令架构: 除了上述两个关键挑战, 物联网的快速兴起使得二进制程序静态分析需要面对更多样的指令架构。传统的二进制程序静态分析主要针对PC中的x86/x64程序, 而物联网设备中大多数是嵌入式设备, 使用ARM、MIPS、PowerPC等架构。因此, 直接基于通用CPU汇编指令的静态分析方法不再适用于物联网设备固件的安全分析。例如, 选择性符号执行框架S2E目前只支持x86架构。

4.2 二进制静态分析的需求

产业驱动技术更新: 从过去几十年内可以看出产业的高速发展驱动程序分析技术的快速更新。随着 PC 的普及, 针对传统 PC 上 x86/x64 二进制程序的安全分析技术成为主流, 伴随着 PC 计算能力和存储能力的增强, 基于符号执行和具体执行的混合执行技术、动态污点分析技术和模糊测试技术都发展迅速。随着 Android 手机的普及, 针对 Android APP 的静/动态分析工作如雨后春笋。

海量异构物联网设备: 根据 Statista 的研究报告, 到 2025 年, 将会有大约 215 亿的 IoT 设备被连接到互联网世界[1]。与传统 PC 和 Android 不同, IoT 设备具有多样的指令架构、不同的操作系统(例如 Linux、RTOS 等)、硬件资源受限、大多基于 C 语言开发以及封闭的源代码和文档等特点。这些特点导致现有的动态分析技术和源代码分析技术都不太适用于 IoT 设备固件安全分析。虽然目前有不少工作通过 IoT 设备固件仿真技术实现混合符号执行、模糊测试等动态分析技术, 但是, 一方面由于 IoT 设备存在外设复杂、与外设 I/O 交互频繁等特点, 导致现有的 IoT 设备固件仿真技术仍然存在诸多限制。例如, Chen 等[65]提出的针对 Linux 内核嵌入式设备固件的全系统仿真平台 Firmadyne, 对收集的 8591 个嵌入式设备固件镜像进行仿真, 只有 23%能仿真成功。另外, 基于设备固件仿真的动态分析无法实现对海量的 IoT 设备进行安全分析。而静态分析可以克服这些限制, 因此物联网的高速发展将驱动二进制程序静态分析技术的进一步更新。

5 二进制数据流分析与别名分析技术

5.1 数据流分析技术

针对二进制程序的数据流分析的核心思想和源代码分析相同, 其主要难点是如何跟踪内存访问之间的数据流信息。在源代码分析中, 可以通过路径访问描述内存访问位置来跟踪它们之间的数据流[66]。而在二进制程序分析中, 由于变量的符号名和数据结构等语义信息丢失, 使得跟踪内存访问之间的数据流变得尤为困难。内存访问跟踪的本质是指针别名识别, 即数据如何在指向同一内存位置的多个别名指针之间进行传播。

Amme 等[67]提出针对汇编代码的过程内的基于值的数据依赖分析方法。早期针对汇编代码的数据流分析方法只确定寄存器访问之间的数据依赖。为了解决内存访问之间的数据依赖, 该论文提出基于符号值传播的方法来建立内存操作间的数据依赖, 而不仅仅是基于地址的数据依赖。然而, 该方法存在以下缺陷: (1) 不是上下文敏感算法, 导致函数间的数据依赖不精确; (2) 该方法把不确定值的寄存器初始化为符号值, 互为别名的两个不确定值的寄存器可能会初始化为不同的符号值, 导致别名变量之间的数据依赖信息丢失。

为了更精确地跟踪内存访问之间的数据流信息, Balakrishnan 和 Reps[68]首次提出值集分析 VSA (Value-Set Analysis)。VSA 的核心思想是确定寄存器或内存位置在每个程序点所保存的数值或地址的超近似值集。首先, VSA 算法识别二进制程序中的类似变量的实体——*a-locs*(abstract locations), 包括栈变量、堆变量和全局变量; 然后, 通过前向数据流分析, VSA 计算每个 *a-locs* 的超近似值集; 最后, VSA 通过聚集结构识别(Aggregation Structure Identification)算法分析 *a-locs*, 恢复变量的结构类型(例如数据结构、数组等), 进一步优化 VSA 的结果。基于 VSA, 该团队实现流敏感、上下文敏感的过程间数据流分析算法。它基于函数调用串[30]来区分不同的上下文, 从

而实现上下文敏感分析。但该方法仍存在以下缺陷: (1) 当 VSA 通过一组规则(例如加法运算)计算指针的地址范围时, 会扩大指针的地址范围, 导致超近似值集, 从而引入错误的数据依赖信息; (2) 当 VSA 无法确定一个指针的内存区域以及地址的约束范围时, 会导致该指针无法被跟踪。

二进制代码中通常根据 call 指令和 ret 指令来识别函数的边界以及函数的调用关系, 而代码混淆技术会混淆 call 指令和 ret 指令, 导致传统的二进制程序的上下文敏感算法(例如函数调用串)不适用。为了解决该问题, Lakhotia 等[69]提出基于栈上下文来构建上下文敏感的过程间数据流。栈上下文是指一组通过压栈或出栈来传递函数调用时的上下文的指令集合。但是, 该方法只适合完全通过压栈方式传递函数参数的程序, 而不适合通过通用寄存器传递函数参数的程序。

已有的针对二进制程序的符号分析[67]没有考虑内存访问变量之间的符号表达式传播, 导致不精确的数据流信息。因此, Anand 等[70]通过静动态结合恢复精确的栈内存模型, 并维护抽象符号和内存位置的映射关系, 在此基础上对二进制程序进行静态符号分析, 实现流敏感、上下文不敏感的数据流分析。该方法的核心是恢复精确的栈内存模型, 以及通过 VSA 确定每个内存访问指令可以访问的内存地址集, 然后生成内存地址和对应的符号值集合之间的映射关系。通过映射关系, 该方法可以在内存访问之间传播符号表达式。

上述方法都是通过精确的栈模型和计算内存访问地址的具体值或值集来生成数据流。但实际程序分析分析中, 有许多内存访问地址无法计算出具体值或值集, 导致这些内存访问之间的数据流信息丢失。因此, Cheng 等[71]提出基于符号表达式的上下文敏感的数据流分析方法, 主要通过静态符号执行对每个函数进行单独分析, 并通过“基地址+偏移”结构的符号表达式描述内存访问, 基于符号表达式构建定义-使用链。在此基础上, 结合指针别名信息对定义-使用链进行更新来构建更加精确的数据依赖。现有数据流分析方法主要使用“自顶向下”遍历函数调用图, 相同函数在不同上下文环境下会多次分析, 导致数据流生成时间开销较大。为了加速数据依赖的生成, 该方法采用“自底向上”遍历函数调用图的方式构建函数间数据依赖, 在维持上下文敏感的同时每个函数最多只需要分析一次。然而, 该方法存在以下缺陷: (1) 对于偏移不是常量的字段访问, 基于符号表达式的方式很难实现准确的数据依赖; (2) 没有评估别名分析对数据依赖的影响, 即别名分析是否提高数据依赖的精度。

Gotovchits 等[72]提出基于模拟执行的路径敏感和上下文敏感的数据依赖分析方法。其核心思想是通过模拟执行指令, 对每个初始读取的寄存器或符号地址均统一地分配随机的具体值, 基于具体值的定义和使用关系生成数据依赖。该工作实现两种模拟执行方式: (1) 通过判断分支条件执行正确的分支路径; (2) 执行所有的分支路径。方式一可实现路径敏感和上下文敏感的数据依赖分析, 并且效率比较高; 方式二则实现流敏感和路径敏感的数据依赖分析。该方法的优势包括 2 点: (1) 可以绕过静态分析中的别名分析、变量恢复和类型恢复等问题从而实现较精确的数据依赖。(2) 可以从任何指令开始分析。然而, 该工作存在以下缺陷: (1) 没有讨论随机初始化值对别名的影响。例如, 互为别名的指针被初始化为不同的随机值。(2) 在执行所有分支路径的方式中, 没有讨论状态爆炸和路径爆炸的问题。虽然实验表明无论是执行一条分支路径还是执行所有分支路径, 该方法的时间开销都很低, 但是这和该工作的应用有很大关系。例如, 用于检测污点类漏洞中, 污点源到污点汇聚点的路径都不是太复杂。(3) 该方法在模拟执行中, 遇到没有建模的动态库函数调用或间接调用时, 会终止执行。这会导致涉及库函数或间接调用的数据依赖信息丢失。

针对二进制程序的数据流分析都在解决同一个关键问题: 如何跟踪内存访问之间的数据流。目前最为著名的方法是 VSA 算法, 通过计算地址的值集来跟踪内存访问。在此基础上, 采用符号值表示不确定的值(例如外部输入), 通过跟踪符号值和符号表达式在寄存器或内存间的传播来提高数据流的准确度。但是现在尚未有效解决的是当符号值用于内存访问地址时如何跟踪该内存中的数据流。

5.2 别名分析技术

Debray 等[55]提出针对可执行代码的流敏感、上下文不敏感的别名分析算法。该算法会在每个程序点将一个寄存器和可能的地址集合关联在一起。通过判断地址集合是否存在交集来确定两个寄存器是否互为别名关系。然而, (1) 该别名分析算法没有跟踪内存访问, 因此从内存读取的寄存器值的信息丢失; (2) 如果一个寄存器的两个不同的定义到达相同的合并点, 该算法会将寄存器的值扩展为符号 *ANY*, 它表示任何可能的地址, 这会导致大量的误报。

为了实现上下文敏感分析, Guo等[56]在2005年第一次提出针对汇编代码的上下文敏感和局部流敏感

的指向分析方法。首先, 对于函数内未初始化值, 该工作通过*UIVs* (Unknown initial values)表示可通过参数或全局变量直接或间接访问的内存位置。*UIVs*可以描述对象的某个字段。例如符号*[A]@4*, 它表示基地址为*A*, 偏移为4的指针所指向的对象。其次, 该工作生成控制流图, 执行过程内和过程间分析, 并通过前向传播具体的函数指针来解决间接调用, 通过迭代分析重构更加完整的过程间控制流图。再次, 为了实现上下文敏感分析, 算法通过“自底向上”遍历函数调用图的方式计算每个函数的转移函数。转移函数总结每个函数对内存的影响, 主要包括对可通过参数或全局变量直接或间接访问的内存位置的修改。该算法借鉴于源代码过程间数据流分析算法[37,41], 即通过调用函数的上下文来更新被调用函数的*UIVs*来实现上下文敏感分析。最后, 算法通过生成的指向集来识别指针别名关系。但是该方法存在以下缺陷: (1)内存访问的别名分析不是流敏感的, 并且采用弱更新来计算指向集; (2)该方法不能识别函数内的*UIVs*之间有没有别名关系。

对于内存访问指针, 之前的二进制别名分析方法无法实现流敏感分析, 并且方法通过单一具体值或符号值来描述访问内存地址, 但是不能描述具有取值范围特征的内存访问地址, 例如数组等结构。针对该问题, Balakrishnan 等[57]提出基于 VSA 来分析 x86 可执行文件中的内存访问, 通过值集来判断两个内存访问指针是否互为别名关系。但是该算法是流敏感和上下文不敏感的。在此基础上, Reps 和 Balakrishnan[58]提出基于 VSA 的改进的内存访问分析方法。该方法通过聚集结构识别算法恢复变量的结构和类型, 以此来提高结构变量(例如数组、链表等)的别名识别。同时, 该方法基于函数调用串[30]来区分不同的上下文, 从而实现上下文敏感分析。然而, 该方法基于 VSA, 继承 VSA 算法本身的缺陷。在别名分析时, 会导致大量的误报。

在二进制别名分析中, VSA 是目前最著名且使用最广泛的方法。除此之外, 一些研究工作尝试利用分析源码的方法来分析二进制别名问题。Brumley 和 Newsome[59]采用基于逻辑的方法, 使用 Datalog 来识别所有可能的别名关系。Datalog 常用于源码的别名分析[73-74]。然而, 该方法存在以下缺陷: (1)不是上下文敏感算法; (2)该方法假设所有内存和寄存器在读取之前先初始化为相同的值, 导致内存访问之间的别名识别中存在误报。

除了上述传统的别名分析方法, 近年来研究人员开始结合深度学习技术来辅助别名分析。基于 VSA 算法的指针别名识别需要依赖从控制流的上下文推测出的内存访问指针的类型(栈指针、堆指针、全局指针)。然而程序的异常崩溃往往导致控制流不完整, 这使得 VSA 无法处理未知类型的指针。针对该问题, Guo 等[60]在 2019 年提出深度学习框架 DEEPVSA, 通过神经网络帮助 VSA 更好地进行别名分析。该论文首先利用指令嵌入网络来捕获每条指令的语义信息; 其次, 采用双向序列到序列的神经结构来学习指令之间的依赖关系; 最后, 通过指令依赖关系来预测每条指令试图访问的内存区域, 从而辅助 VSA 进行别名分析。

相比于源代码中非常丰富的别名分析技术, 二进制程序中的别名分析技术尚处于起步阶段, 例如源代码中的按需别名分析技术, 目前在二进制程序中还没有相关研究工作。并且已有的二进制别名分析工作主要针对 x86 程序, 尚缺少针对 ARM、MIPS 和 PowerPC 等 IoT 中主流的嵌入式架构的别名分析研究工作。

6 二进制符号执行技术

第 3 节介绍过符号执行技术的基本原理, 纯粹的静态符号执行技术缺陷太大。因此, 在二进制程序分析中, 目前符号执行技术大多都是静态符号执行和具体执行的混合执行技术。后续论文将介绍相关技术和工具。

Song 等[2]在 2008 年提出 BitBlaze 二进制分析平台, 其包括 Vine、TEMU 和 Rudder 三个核心组件。其中, Vine 是静态分析组件, 将二进制代码转换为中间表示并提供数据流分析和符号执行等静态分析技术; TEMU 是动态分析组件, 实现细粒度的程序监控和动态二进制检测, 包括语义信息提取和动态污点分析等。Rudder 是基于符号执行和具体执行的执行生成测试组件, 可以自动生成引导程序执行不同路径的输入, 探索程序执行空间的不同部分。其主要方法是: 首先, 初始化符号输入说明, 在程序执行过程中, Rudder 检测指令的操作源, 如果操作源都是具体值, 通过 QEMU[76]具体执行该指令; 反之, 通过符号值标记该指令的操作源并计算目的操作数的符号表达式; 然后, 当一个分支跳转的条件依赖于符号输入时, Rudder 保存当前的执行状态并通过路径选择算法选择一条分支进行探索; 当该分支探索完, Rudder 恢复保存的执行状态并继续探索另外一条分支。对于符号内存地址, Rudder 通过约束求解器 STP[75]计算符号地址的范围。最后, Rudder 通过 STP 求解相应路径的符号约束生成测试用例。Brumley 等[3]在 2011 年对

BitBlaze的静态分析模块Vine进行重新设计实现二进制分析平台BAP, 使其支持x86/ARM等多种架构。

Chipounov等[48]在2009年提出针对二进制程序的选择符号执行框架S2E, 通过在符号执行和具体执行之间无缝来回切换, 同时维持系统状态在符号和具体状态之间的转换来实现对真实大型程序(包括应用程序、库、内核、设备驱动等)的分析。相比于传统的符号执行, S2E的优势是可以选择感兴趣的小部分代码进行符号执行分析(模拟执行), 而大部分代码采用具体执行(本地执行)。这是S2E能将符号执行用于真实系统分析的关键。S2E通过按需转换方法实现具体执行和符号执行之间的高效切换。按需转换指的是只有当数据被读取并作为分支条件的一部分或对被读取的数据进行算术运算时, 才执行具体数据和符号数据之间的转换。并且, S2E使用相同的模型保存符号数据和具体数据, 该方法能让具体执行透明的处理符号数据。S2E的实现是基于QEMU虚拟机[76]和KLEE符号执行引擎[77]。虽然S2E的功能强大, 但是它存在一定的局限性。例如, S2E并不适用于嵌入式设备固件代码的分析。主要原因包括2点: (1) 大多数嵌入式设备采用ARM、MIPS、PowerPC架构, 而S2E目前只支持x86架构; (2) 相比于传统x86程序, 嵌入式设备包含更多样的外设和中断以及更频繁的I/O交互, 目前S2E无法满足不同的外设模拟。因此, 这会引入更多的符号值, 导致S2E的具体执行能力退化, 从而面临传统符号执行的问题, 例如路径爆炸和复杂符号路径约束求解难。

Cha等[78]在2012年提出一种结合线下符号执行和线上符号执行的新型混合符号执行框架Mayhem。其中, 线下符号执行是指系统在种子输入下具体地执行一条路径并记录执行的指令流, 再符号地执行该指令流来探索新的路径, 例如混合执行测试; 线上符号执行是指在条件分支处生成两个执行器来避免重复执行带来的开销, 每个执行器都有当前执行状态的拷贝, 例如执行生成测试。Mayhem系统设计基于4个原则: ①可以永久地执行并不超出给定的资源(例如内存); ②系统不应该做重复的工作; ③系统不应该丢弃任何之前分析的结果; ④系统能解释符号内存地址。而线下和线上符号执行都不能同时满足这4个原则。Mayhem系统包括具体执行客户端(线下)和符号执行服务端(线上)两个部分。其核心思想是线上端在依赖于输入的分支条件处生成两个执行器来探索更多的路径, 当内存资源消耗到达阈值时, 保存所有活动执行器的状态信息(路径约束、统计数据、执行停止点等), 并选择其中一个执行器来求解路径约束, 生成测试用例; 线下端通过生成的测试用例具体地执行目标程序, 在当前选择的符号执行器的停止点处获取具体执行状态(寄存器/内存值、OS状态), 恢复执行器的符号执行状态, 继续切换到线上符号执行探索更多的路径。Mayhem充分利用线下符号执行资源开销低和线上符号执行路径探索能力强的优势。

Shoshitaishvili等[4]在2016年提出二进制程序分析框架Angr, 它是用python开发实现的高度模块化的开源二进制分析框架, 使用Valgrind的VEX中间表示[18]。Angr集成多种二进制程序分析技术, 包括符号执行、欠约束符号执行、静态分析(包括程序切片、数据依赖生成、值集分析等)、崩溃重放、利用生成等。

针对混合执行用于真实程序测试时性能低的问题, Yun等在2018年设计一个快速的混合执行引擎框架QSYM[79], 用于支持混合的模糊测试(hybrid fuzzing)。混合的模糊测试是最近提出的一种模糊测试技术, 它结合模糊测试和混合执行的优势。该论文总结现有的混合执行方法的瓶颈: (1) 缓慢的符号模拟: 现有的混合执行引擎基于中间表示实现(例如LLVM IR、VEX IR), 而中间表示的翻译是导致其慢的关键因素之一; (2) 效率低的快照技术: 由于模糊测试输入并不会共享一个共同的分支, 这导致用于一个测试用例的快照无法在另一个测试用例中被重复利用; (3) 缓慢而不灵活的可靠分析: 复杂约束的求解导致永无止境的分析以及完整的约束会导致路径过约束, 从而限制混合执行找到更多的路径。针对上面的3个瓶颈, 该论文提出3个解决方法: (1) 移除中间表示转换层来降低执行负载以及通过指令级别的选择符号执行来优化符号模拟; (2) 移除快照并通过具体执行来建模外部环境; (3) 在符号执行时收集不完整的约束来提高效率, 并且当存在路径过约束时, 只解决部分的约束。通过这3个方法, 该论文极大地提高了混合执行用于混合模糊测试的效率。

在二进制程序分析中, 相比数据流分析和别名分析, 符号执行技术比较成熟, 在IoT设备固件安全分析中也有相应的应用, 包括基于Qemu仿真的动态固件安全测试[80]和基于符号执行和污点分析的静态固件漏洞挖掘[81-82]。但是目前符号执行技术以混合执行为主。当混合执行用于IoT设备固件分析时, 需要结合固件仿真技术。总体来说, 目前符号执行技术还是有很多方面需要提升, 包括缓解路径爆炸、符号内存访问、提高约束求解能力等。

7 二进制静态污点分析技术

二进制静态污点分析技术按照实现方式可以分

为 2 类: (1)基于数据流分析的静态污点分析, (2)基于符号执行的静态污点分析。后续论文将介绍相关工作的研究进展。

7.1 基于数据流分析的静态污点分析技术

静态污点分析可以在数据流分析的基础上实现, 通过确定污点源并标记污点数据, 基于数据的流向传播污点。因此, 污点分析的精度和效率主要取决于数据流分析算法。

Rawat 等[83]提出针对二进制可执行程序的过程内和过程间的静态污点分析方法。对于过程内分析, 该方法基于 VSA, 通过前向数据流分析计算每个程序点的寄存器和内存访问地址的值集, 并通过变量的数据依赖关系传播污点。对于过程间分析, 该方法对函数建立摘要, 即污点数据对函数参数和返回值的影响。基于函数摘要实现过程间的污点分析同时维持较高的效率。然而, 该方法基于 VSA 算法, VSA 的 3 个缺陷也会导致该污点分析方法引入过污染和欠污染问题。例如, VSA 算法计算值的超近似值集, 会将不是污点的变量标记为污点, 导致过污染。

对于 ARM 可执行二进制程序, Eom 等[84]和 Choi 等[85]提出基于数据依赖的静态污点分析方法, 通过污点分析检测程序崩溃后的可利用点。该方法不依赖中间表示, 而是直接对 ARM 汇编指令进行分析。主要思路是直接从程序崩溃点开始, 结合程序实时运行的快照, 获取程序中寄存器和变量的具体值, 通过具体值的定义和使用生成定义-使用链。然后, 在定义-使用链的基础上传播污点数据来检查崩溃是否可利用。该污点分析方法主要依赖于实时快照获取的寄存器或变量的具体值。然而, 有些二进制程序无法获取实时快照, 例如, 嵌入式设备固件中的二进制程序。因此, 该方法具有较强的局限性。

Feng 等[86]提出针对二进制程序漏洞挖掘的静态污点分析工具 Bintaint。该工具的核心思想是: 首先, 通过构建控制流图, 找到污点源到污点汇聚点的所有路径集合; 其次, 从污点源开始初始化污点数据, 通过解析中间表示并结合污染策略传播污点, 选择所有被污染的基本块重新构建污染控制流图; 再次, 从污染控制流图中选择一条从污点源到污点汇聚点的路径, 并通过符号执行工具 Angr[4]执行该路径并求解路径约束, 获取测试用例; 最后, 通过重放测试用例, 如果程序能正确执行, 则说明污点汇聚点能够被污染。该工作提出的静态污点分析方法是粗粒度的, 没有考虑内存访问地址的别名问题以及被调用函数对污点传播的影响, 会导致欠污染。

7.2 基于符号执行的静态污点分析技术

在静态二进制程序分析技术中, 符号执行是发展最成熟且使用最多的一种技术。在符号执行的基础上, 通过标记来自污点源的具体值或符号值为污点, 利用前向符号执行, 可以很自然地实现污点传播分析。利用符号执行进行污点分析的优势是既可以跟踪具体值, 也可以跟踪符号值。并且符号执行的符号地址具体化策略可以在一定程度上解决互为别名的内存访问之间的污点传播。

Redini 等[81]首次提出基于欠约束符号执行的污点分析方法。其主要思路是: 首先, 该方法识别污点种子和污点汇聚点, 找到那些直接调用污点种子的函数作为入口点; 其次, 该方法对每次调用的污点种子创建唯一的污点标签, 保证相同的污点种子在不同程序点被调用会引入不同的污点数据; 最后, 通过欠约束符号执行, 直接从该函数入口点开始分析。欠约束符号执行可以选择从任何函数开始执行, 提高符号执行的效率。在符号执行中, 该方法通过检查约束来判断是否移除污点标记, 例如, 对于污点变量 x, 如果检查到存在约束 $x<N$, 其中 N 不是污染值, 那么 x 的污点标签被移除。在符号内存地址具体化策略中, 当符号值能被具体化多个值时, 该方法选择更小的值, 这样能允许符号执行探索更深的路径。并且不同的符号变量被具体化为不同的值, 这能一定程度上降低别名内存访问的误报率。

目前比较有影响力的二进制静态污点分析工作主要基于 VSA 算法或符号执行, 其中基于 VSA 算法的污点分析受到 VSA 本身缺陷的影响, 会导致大量的过污染; 而基于欠约束符号执行的污点分析在污点源的选择上受到限制。欠约束符号执行的优势是可以从任何函数开始执行。但是如果直接从污点源所在的函数开始执行, 则污点信息只能传播到被调用的函数, 而无法传播到父函数, 会导致大量的欠污染; 另外, 当污点源到污点汇聚点的路径比较深时, 基于欠约束符号执行的污点分析需要考虑路径剪枝, 防止路径和状态爆炸。并且相比于源代码静态污点分析, 二进制静态污点分析缺少更高效的工具, 例如像 FlowDroid[61]能实现流敏感、上下文敏感、域敏感和对象敏感的高效率和高精度的污点分析工具。

8 二进制静态分析技术在漏洞发现中的应用

二进制静态分析技术有诸多应用, 包括数据结构和类型恢复[87-90]、二进制重写[91]、恶意

代码检测[92]、控制流完整性分析[28,93-94]、代码相似性分析[95]、漏洞签名[96]、漏洞检测[81-82,97]等。

虽然二进制静态分析技术应用广泛, 但是在大部分应用中并没有针对静态分析技术本身进行改进, 这也是导致二进制静态分析技术发展缓慢的原因之一。随着 IoT 设备的普及, 以及 IoT 设备动态分析难以适用, 近几年通过二进制静态分析检测 IoT 设备固件安全的相关工作越来越多。这些工作涉及数据流分析、别名分析、污点分析和符号执行, 并且对不同技术进行改进和提高, 在安全领域取得突出的研究成果。后续论文将介绍二进制静态分析技术在漏洞检测方面的应用。

Balakrishnan 等[98]在 2005 年提出针对 x86 二进制程序的静态分析框架 CoderSurfer/x86, 该研究团队首次提出“WYSINWYX”[68], 即你看到不是你所执行的, 描述二进制程序分析和源代码分析的区别, 以及著名的值集分析 VSA 算法。在框架 CoderSurfer/x86 中, 该团队实现数据依赖分析、变量恢复、别名分析、二进制重写等多种静态分析技术。

Cova 等[99]在 2006 年首次提出针对 x86 可执行程序的基于静态符号执行技术的污点类(taint-style)漏洞检测方法。首先, 该方法通过解决间接跳转和间接调用来生成完整的控制流图。对于间接跳转, 通过后向回溯到跳转表来解决; 对于间接调用, 通过常量传播函数指针来解决。其次, 该方法标记来自污点源的变量为污点, 通过符号执行传播污点标记, 并在污点汇聚点检测污点数据是否经过无害处理来判断漏洞。在该方法中, 只选择函数 system 和 popen 作为污点汇聚点, 即只检测命令注入类漏洞。为了缓解路径爆炸, 该方法采用路径切片的方法, 即只分析从 main 函数到污点汇聚点的所有路径。对于别名问题, 该方法采用最简单的方式: 假设所有不同的符号内存地址指向不同的内存位置。然而, 该假设会同时导致漏报和误报。

对于整数溢出漏洞, Wang 等[100]在 2009 年提出针对 x86 二进制程序的整数溢出漏洞检测工具 IntScope, 其核心思想是通过符号执行和污点分析跟踪来自污点源的数据并收集路径约束, 检查污点数据是否被用于安全敏感操作(例如堆内存分配函数 malloc 的大小参数、内存索引等)以及路径约束是否对污点数据进行安全检查以防止整数溢出。该方法通过只分析那些能从污点源到达安全敏感操作点的路径来缓解符号执行路径爆炸问题。针对整数溢出静态检查误报率高问题, Zhang 等[101]在 2015 年提出改进的整数溢出检查工具 INDIO, 其主要思路是通过静态污点分析找到污点源到安全敏感操作点的可疑路径, 再通过优先级排序和最弱前置条件计算剔除误报的路径, 最后通过 S2E 符号执行验证剩余可疑的路径。相比 IntScope, INDIO 能降低误报率并发现更多的整数溢出漏洞。

Cheng 等[102]在 2011 年提出针对 x86 二进制程序的基于摘要的污点类漏洞检测工具 LoongChecker。在过程内分析中, 该方法首先通过类型推测来恢复可能的变量类型, 包括地址指针和非指针等; 其次通过后向切片将函数的指令分割成和地址指针相关的指令集以及和非指针变量相关的指令集; 再次通过 VSA 算法跟踪变量地址来计算尽可能多的地址值, 并结合数据依赖分析算法来跟踪非指针变量的数据流依赖; 最后生成函数的摘要信息, 包括函数未初始化参数和调用点实参的对应关系, 以及该函数对参数、全局变量以及堆对象的定义和使用结果。在过程间分析中, 采用“自顶向下”遍历调用图的方式, 从程序入口点开始使用实参(包括污点源返回值)和全局变量初始化值迭代的更新被调用函数摘要中未初始化形参和全局变量, 同时在污点汇聚点检测污点标记来判断是否存在漏洞。LoongChecher 在 3 个真实程序中发现 3 个已知漏洞和 2 个 0-day 漏洞。然而该方法存在以下缺陷: (1) 不是域敏感和对象敏感的数据流分析; (2) 没有考虑被调用函数对调用函数的数据流的影响; (3) 在函数内使用 VSA 算法, 无法解决与指针参数和堆指针相关的多层指针别名关系。

Rawat 等[83]在 2011 年提出基于静态污点技术的污点类漏洞检测方法, 其核心是通过 VSA 算法实现流敏感和上下文敏感的污点分析。为了适用于真实的大型程序分析, Rawat 等[103]又提出针对函数调用图的切片方法, 主要是通过分析污点源到安全敏感点路径上相关的函数, 避免分析整个二进制程序而造成的效率低问题。虽然基于函数调用图的切片方法可以提高污点分析方法针对大型程序分析的效率, 但是该切片方法是粗粒度的, 会导致大量的欠污染。

Shoshitaishvili 等[97]在 2015 年首次提出针对固件二进制的基于符号执行技术的认证绕过漏洞自动化检测方法, 其核心思想是利用符号执行探索程序入口点到特权程序点(只有用户认证后才能到达的程序点)的所有路径, 通过求解路径约束来判断是否存在认证绕过漏洞。如果路径约束可解并且求解的输入能够唯一具体化, 代表到达特权程序点所需的输入可由攻击者唯一确定, 那么其对应的路径存在认证绕过漏洞。然而, 该工作存在以下缺陷: (1) 需要人工

指定特权程序点; (2) 固件可通过代码混淆等技术使路径约束无法求解成功。

Cheng 等[71]在 2018 年提出基于单函数符号执行和变量表达式更新的数据流生成方法, 在此基础上实现针对嵌入式设备固件的污点类检测工具 DTaint。在 6 个固件样本上, DTaint 发现 21 个漏洞, 其中包括 13 个 0-day 漏洞, 准确率为 24.7%。Lin 等[104]在 2019 年提出基于条件合并的改进 VSA 方法, 并实现原型系统 RVSA 来检查内存崩溃漏洞, 在 Netgear 路由器的 httpd 二进制程序中发现 25 个 0-day 漏洞, 准确率为 27.0%, 相比 Angr-VSA 的 12.8%的准确率有所提高。

Redini 等[81]在 2017 年提出针对手机设备中引导程序(bootloader)的安全分析工具 BootStomp, 其核心思想是通过欠约束符号执行技术实现多重标签的污点分析来检测引导程序中的漏洞。BootStomp 在 5 个不同的引导程序中发现 7 个漏洞, 其中包括 6 个 0-day 漏洞和 1 个已知漏洞, 准确率为 38.3%。Redini 等[82]在 2020 年又通过该污点分析技术实现针对嵌入式设备固件中多个二进制程序间交互的污点类漏洞检测工具 KARONTE。其核心思想是先找到处理外部请求的边界二进制程序, 例如通过 recv 函数接受网络数据的程序; 再通过识别多个二进制程序之间进程交互关系构建二进制依赖图; 最后通过污点分析跟踪多个二进制程序之间的数据流来检测程序交互导致的漏洞。KARONTE 评估来自 7 个不同厂商的 53 个固件样本, 发现 51 个漏洞(DDoS 和缓冲区溢出), 其中包括 17 个单二进制漏洞和 34 个多二进制交互漏洞, 准确率为 58.6%。BootStomp 和 KARONTE 都采用欠约束符号执行来进行污点分析, 主要原因是符号执行本身存在的路径爆炸缺陷导致其不太适用于真实的大型程序分析。然而, 欠约束符号执行用于污点分析会存在以下缺陷: (1) 若函数参数之间存在别名关系, 欠约束符号执行会导致欠污染问题; (2) 无法直接从常见的污点源, 包括 recv/read 等函数, 传播污点到污点汇聚点。而是需要通过启发式方法选择特殊的污点源。例如在 KARONTE 中, 其主要通过网络编码字符串(例如 soap、HTTP)来推测污点源。但是, 启发式的方法会导致较高的漏报率和一定的误报率。并且启发式的方法需要通过人工分析目标二进制代码来定制化地实现污点源推测策略。

相比二进制程序动态漏洞检测工作, 二进制程序静态漏洞检测工作相对较少, 针对的漏洞主要包括缓冲区溢出、命令注入、整数溢出、未初始化变量、认证绕过漏洞等, 其中大部分工作聚焦于污点类漏洞检测, 即污点数据从污点源到污点汇聚点没有经过无害处理而导致的一类漏洞。

二进制漏洞静态检测大多数是基于已有的二进制分析工具实现的。如表 2 所示, 我们对这些开源或闭源的二进制分析工具进行总结。二进制分析通常是将汇编代码转换成统一的中间表示, 然后基于中间表示进行程序分析。目前大多数开源的二进制分析工具支持的架构比较全面, 包括 x86、ARM、MIPS、PowerPC 等主流的架构。在最近几年, 二进制静态分析研究工作主要是通过 Angr 提供的符号执行技术实现的嵌入式设备固件漏洞检测, 其中包括基于符号执行实现的数据流分析和别名分析[71], 以及静态污点分析[81-82]。

表 2 二进制静态分析工具以及相关应用总结

Table 2 the summary of binary static analysis tools and its applications

工具	实现技术	支持架构	中间表示	应用	开源
CodeSurfer/x86[98]	VSA、数据依赖分析、别名分析	x86	自定义 IR	漏洞检测[98]、变量恢复[89]、二进制重写等	否
LoongChecker[102]	VSA、数据依赖分析、静态污点分析	x86	REIL IR[19]	漏洞检测[102]	否
IntScope[100]	静态污点分析、符号执行	x86	自定义 IR	整数溢出漏洞检测[100]	否
BinNavi[11]	数据依赖分析	x86/MIPS/ARM/PowerPC	REIL IR[19]	污点类漏洞检测[83-103]	是
BAP[3]	符号执行、VSA(目前只支持 x86)	x86/MIPS/ARM/PowerPC	BAP IR	TIE: 变量数据结构和类型恢复[87]、Saluki: 污点类漏洞检测[72]	是
S2E[48]	选择性符号执行	x86	LLVM IR[17]	INDIO: 整数溢出漏洞检测[101]	是
Angr[4]	VSA、数据流分析、符号执行、欠约束符号执行、崩溃重放、利用生成	x86/ARM/MIPS/PowerPC	VEX IR[18]	DTaint: 固件污点类漏洞检测[71]、Firmalice: 固件认证绕过漏洞检测[97]、BootStomp: 手机引导程序漏洞检测[81]、KARONTE: 固件多二进制交互漏洞检测[82]	是

9 总结与展望

静态分析技术作为程序分析中重要的分支, 包括数据流分析、别名分析、符号执行和静态污点分析等技术, 在源代码分析上具有非常成熟的理论基础以及经典的算法。然而不幸的是, 在二进制程序分析领域, 静态分析却发展得相当缓慢。我们总结两个主要的原因: (1) 相比源代码, 二进制汇编代码中变量的符号名和数据结构等语义信息的丢失, 直接导致很多经典的源代码静态分析算法无法直接适用于二进制分析; (2) 在传统 PC 程序上, 相比静态分析, 动态分析不需要考虑别名影响和间接调用这两个静态分析中的关键问题。并且在漏洞挖掘方面, 能获取更多的信息(指令流、寄存器和内存值、堆栈分布等), 以及低的误报率。而静态分析的误报率太高, 导致需要花费大量额外的人工去验证漏洞的可利用性。

但是物联网的快速发展与广泛应用给二进制静态分析带来新的挑战与需求, 其原因有两点: (1)IoT 设备具有多样的指令架构、不同的操作系统、硬件资源受限、大多基于 C 语言开发以及封闭的源代码和文档等特点使得源代码分析技术和动态分析技术都不太适用于 IoT 设备固件安全分析; (2) 虽然目前 IoT 设备固件仿真技术让动态分析方法成为可能, 并逐渐成为当前的研究热点, 但是由于仿真能力的限制, 基于设备仿真的动态分析无法实现对海量异构的 IoT 设备进行安全分析。相比动态污点分析, 静态分析的 3 个优势, 包括(1) 代码覆盖率高, (2) 不依赖于具体的输入, (3) 适合自动化, 使得二进制静态分析能够解决海量异构的 IoT 设备的安全问题。

当前二进制静态分析技术主要包括数据流分析、别名分析、符号执行、静态污点分析等。其中, 对于二进制数据流分析, 主要解决的问题是内存访问之间的数据流跟踪, 最为著名的是值集分析 VSA 算法; 对于二进制别名分析, 目前的主要方法是基于 VSA 算法计算指针地址的值集来识别指针别名关系。但在按需别名分析方面尚缺少相关工作。对于二进制符号执行, 目前以静态符号执行和具体执行相结合的混合执行为主, 并且有不少成熟且流行的开源框架 S2E、BAP、Angr 和 QSYM, 在 IoT 设备固件安全分析方面也有一定的进展。但混合符号执行依赖于 IoT 设备的仿真技术, 属于动态分析。对于二进制静态污点分析, 目前包括基于数据流分析的静态污点分析和基于符号执行的静态污点分析, 其中前者主要通过 VSA 算法实现污点传播, 存在过污染问题; 后者主要通过欠约束符号执行实现污点传播, 存在欠污染问题。

并且, 我们发现数据流分析、别名分析、符号执行和静态污点分析工作大部分都没有解决间接调用问题, 而少部分工作则是通过常量指针前向传播分析来解决间接调用, 其中符号执行也是采用该方法。但是, 该方法存在以下缺陷: (1) 不适合真实大型程序分析; (2) 由于存在符号内存地址, 很难准确地跟踪常量指针; (3) 实际中有很多间接调用的目标地址会和用户输入相关, 常量指针传播不能解决这类间接调用。

总体来说, 当前二进制静态分析工作主要基于两个比较成熟的技术: 一方面, 值集分析 VSA 和符号执行, 但 VSA 和符号执行都存在缺陷, 在针对真实大型程序分析中, 都无法维持效率和精度, 可应用性不高。另一方面, 二进制静态分析面临的两个关键挑战: 指针别名问题和间接调用问题, 都没有很好的解决方案。因此, 我们认为未来针对二进制静态分析的研究重点应该着重研究更加高效且精确的别名分析算法以及过程间控制流图恢复方法, 在此基础上研究数据流分析、静态污点分析等技术。

参考文献

[1] Statista. Internet of Things (IoT) active device connections installed base worldwide from 2015 to 2025[Z]. https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/. Dec. 2020.

[2] Song D, Brumley D, Yin H, et al. BitBlaze: A New Approach to Computer Security *via* Binary Analysis[C]. *International Conference on Information Systems Security.* Berlin, Heidelberg: Springer, 2008: 1-25.

[3] Brumley D, Jager I, Avgerinos T, et al. BAP: A Binary Analysis Platform[C]. *International Conference on Computer Aided Verification*. Berlin, Heidelberg: Springer, 2011: 463-469.

[4] Shoshitaishvili Y, Wang R Y, Salls C, et al. SOK: (State of) the Art of War: Offensive Techniques in Binary Analysis[C]. *2016 IEEE Symposium on Security and Privacy*, 2016: 138-157.

[5] Dinaburg A, Ruef A. Mcsema: Static translation of x86 instructions to llvm[C]. *ReCon 2014 Conference*, 2014.

[6] Ravipati G, Bernat A R, Rosenblum N, et al. Toward the deconstruction of Dyninst[J]. *Univ. of Wisconsin, technical report*, 2007: 32.

[7] Wang S, Wang P, Wu D H. Reassembleable Disassembling[C]. *The 24th USENIX Conference on Security Symposium*, 2015: 627-642.

[8] Wang S, Wang P, Wu D H. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling[C]. *2016 IEEE 23rd International Conference on Software Analysis*, *Evolution*, *and Reengineering*, 2016: 236-247.

[9] Ghidra[Z]. https://www.nsa.gov/resources/everone/ghidra/. 2019.

[10] Radare2[Z]. https://github.com/radareorg/radare2/tree/5a1df188.2019.

[11] Zynamics. BinNavi binary code reverse engineering tool[Z]. https:// www.zynamics.com/binnavi.html. 2020.

[12] BINARYNINJA: A reverse engineering platform[Z]. https://binary.ninja/. 2020.

[13] IDA: About - Hex-Ray[Z]. https://www.hex-rays.com/products/ida/ index.shtml. 2020.

[14] Andriesse D, Chen X, van der Veen V, et al. An In-Depth Analysis of Disassembly on Full-Scale X86/X64 Binaries[C]. *The 25th USENIX Conference on Security Symposium*, 2016: 583-600.

[15] Meng X Z, Miller B P. Binary Code is not Easy[C]. *The 25th International Symposium on Software Testing and Analysis*, 2016: 24-35.

[16] Pang C B, Yu R T, Chen Y H, et al. SoK: All You ever Wanted to Know about X86/X64 Binary Disassembly but were Afraid to Ask[EB/OL]. 2020: arXiv: 2007.14266. https://arxiv.org/abs/2007.14266.

[17] Lattner C, Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation[C]. *The International Symposium on Code Generation and Optimization*: *Feedback-Directed and Runtime Optimization*, 2004: 75-86.

[18] Nethercote N, Seward J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation[C]. *The 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007: 89-100.

[19] REIL - Reverse Engineering Intermediate Language[Z]. https://www.zynamics.com/binnavi/manual/html/reil_language.htm. 2020.

[20] Psi: A platform for secure static binary instrumentation[Z]. http://www.seclab.cs.sunysb.edu/seclab/psi/. 2019.

[21] Uroboros github repo[Z]. https://github.com/s3team/uroboros/tree/986353c0. 2019.

[22] Dyninst github repo[Z]. https://github.com/dyninst/dyninst/ tree/5d2ddacb. 2019.

[23] GNU. Index of /gnu/binutils[Z]. https://ftp.gnu.org/gnu/binutils/. 2019.

[24] Mcsema github repo[Z]. https://github.com/lifting-bits/mcsema/tree/62a1319e. 2019.

[25] Angr github repo[Z]. https://git-hub.com/angr/angr/tree/76da434f. 2019.

[26] Bap github repo[Z]. https://github.com/BinaryAnalysisPlatform/bap/tree/cfeacbfc. 2020.

[27] van der Veen V, Göktas E, Contag M, et al. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level[C]. *2016 IEEE Symposium on Security and Privacy*, 2016: 934-953.

[28] Kim S H, Sun C, Zeng D R, et al. Refining Indirect Call Targets at the Binary Level[C]. *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[29] Wang X F, Zhao K J, Tian Z W. Research of key techniques in data flow analysis[J]. *Computer Science*, 2005, 32(12): 91-93.
(汪小飞, 赵克佳, 田祖伟. 数据流分析的关键技术研究[J]. *计算机科学*, 2005, 32(12): 91-93.)

[30] Reps T, Horwitz S, Sagiv M. Precise Interprocedural Dataflow Analysis *via* Graph Reachability[C]. *The 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995: 49-61.

[31] Zheng X, Rugina R. Demand-Driven Alias Analysis for C[C]. *The 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008: 197-208.

[32] Weihl W E. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables[C]. *The 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1980: 83-94.

[33] Andersen L O. Program Analysis and Specialization for the C Programming Language[D]. Diss. University of Cophenhagen, 1994.

[34] Steensgaard B. Points-to Analysis in almost Linear Time[C]. *The 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996: 32-41.

[35] Zhang, Ryder B G, Landi W. Program Decomposition For Pointer Aliasing: A Step Towards Practical Analyses[C]. *The 4th Symposium on the Foundations of Software Engineering*, 1996.

[36] Shapiro M, Horwitz S. Fast and Accurate Flow-Insensitive Points-to Analysis[C]. *The 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997: 1-14.

[37] Chatterjee R, Ryder B G, Landi W A. Relevant Context Inference[C]. *The 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999: 133-146.

[38] Choi J D, Burke M, Carini P. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects[C]. *The 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993: 232-245.

[39] Emami M, Ghiya R, Hendren L J. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers[C]. *The ACM SIGPLAN 1994 conference on Programming language design and implementation*, 1994: 242-256.

[40] Deutsch A. Interprocedural May-Alias Analysis for Pointers: Beyond *k*-Limiting[C]. *The ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994: 230-241.

[41] Wilson R P, Lam M S. Efficient Context-Sensitive Pointer Analysis for C Programs[C]. *The ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, 1995: 1-12.

[42] Wang L, Li F, Li L, et al. Principle and practice of taint analysis[J]. *Journal of Software*, 2017, 28(4): 860-882.
(王蕾, 李丰, 李炼, 等. 污点分析技术的原理和实践应用[J]. *软件学报*, 2017, 28(4): 860-882.)

[43] Ye Z B, Yan B. Survey of symbolic execution[J]. *Computer Science*, 2018, 45(S1): 28-35.
(叶志斌, 严波. 符号执行研究综述[J]. *计算机科学*, 2018, 45(S1): 28-35.)

[44] Schwartz E J, Avgerinos T, Brumley D. All You ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but might Have been Afraid to Ask)[C]. *2010 IEEE Symposium on Security and Privacy*, 2010: 317-331.

[45] Majumdar R, Sen K. Hybrid Concolic Testing[C]. *29th International Conference on Software Engineering*, 2007: 416-426.

[46] Godefroid P, Klarlund N, Sen K. DART: Directed Automated Random Testing[C]. *The 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005: 213-223.

[47] Cadar C, Ganesh V, Pawlowski P M, et al. EXE: Automatically generating inputs of death[J]. *ACM Transactions on Information*

and System Security, 2008, 12(2): 1-38.

[48] Chipounov V, Georgescu V, Zamfir C, et al. Selective symbolic execution[C]. *The 5th Workshop on Hot Topics in System Dependability*, 2009.

[49] Ramos D A, Engler D. Under-constrained Symbolic Execution: Correctness Checking for Real Code[C]. *24th USENIX Security Symposium*, 2015: 49-64.

[50] Heintze N, Tardieu O. Demand-driven pointer analysis[J]. *ACM SIGPLAN Notices*, 2001, 36(5): 24-34.

[51] Sridharan M, Gopan D, Shan L X, et al. Demand-driven points-to analysis for Java[J]. *ACM SIGPLAN Notices*, 2005, 40(10): 59-76.

[52] Späth J, Nguyen Quang Do L, Ali K, et al. Boomerang: Demand-driven Flow-and Context-Sensitive Pointer Analysis for Java[C]. *30th European Conference on Object-Oriented Programming*, 2016.

[53] Sui Y L, Xue J L. Value-flow-based demand-driven pointer analysis for C and C[J]. *IEEE Transactions on Software Engineering*, 2020, 46(8): 812-835.

[54] Jaiswal S, Khedker U P, Chakraborty S. Bidirectionality in flow-sensitive demand-driven analysis[J]. *Science of Computer Programming*, 2020, 190: 102391.

[55] Debray S, Muth R, Weippert M. Alias Analysis of Executable Code[C]. *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998: 12-24.

[56] Guo B, Bridges M J, Triantafyllis S, et al. Practical and Accurate Low-Level Pointer Analysis[C]. *International Symposium on Code Generation and Optimization*, 2005: 291-302.

[57] Balakrishnan G, Reps T. Analyzing Memory Accesses in x86 Executables[C]. *International Conference on Compiler Construction*. Berlin, Heidelberg: Springer, 2004: 5-23.

[58] Reps T, Balakrishnan G. Improved Memory-Access Analysis for x86 Executables[C]. *International Conference on Compiler Construction*. Berlin, Heidelberg: Springer, 2008: 16-35.

[59] Brumley D, Newsome J. Alias analysis for assembly[R]. Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006.

[60] Guo W, Mu D, Xing X, et al. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis[C]. *28th USENIX Security Symposium*, 2019: 1787-1804.

[61] Arzt S, Rasthofer S, Fritz C, et al. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps[C]. *The 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014: 259-269.

[62] Cifuentes C, van Emmerik M. Recovery of Jump Table Case Statements from Binary Code[J]. *Science of Computer Programming*, 2001, 40(2/3): 171-188.

[63] De Sutter B, De Bosschere K, Keyngnaert P, et al. On the Static Analysis Of Indirect Control Transfers in Binaries[C]. *The International Conference on Parallel and Distributed Processing Techniques and Applications*, 2000, 2: 1013-1019.

[64] Reinbacher T, Brauer J. Precise Control Flow Reconstruction Using Boolean Logic[C]. *The ninth ACM international conference on Embedded software*, 2011: 117-126.

[65] Chen D D, Egele M, Woo M, et al. Towards Automated Dynamic Analysis for Linux-Based Embedded Firmware[C]. *2016 Network and Distributed System Security Symposium*, 2016: 1-16.

[66] Lerch J, Späth J, Bodden E, et al. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths (T)[C]. *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, 2016: 619-629.

[67] Amme W, Braun P, Thomasset F, et al. Data dependence analysis of assembly code[J]. *International Journal of Parallel Programming*, 2000, 28(5): 431-467.

[68] Balakrishnan G, Reps T. WYSINWYX: What you see is not what you eXecute[J]. *ACM Transactions on Programming Languages and Systems*, 2010, 32(6): 1-84.

[69] Lakhotia A, Boccardo D R, Singh A, et al. Context-sensitive analysis without calling-context[J]. *Higher-Order and Symbolic Computation*, 2010, 23(3): 275-313.

[70] Anand K, Elwazeer K, Kotha A, et al. An Accurate Stack Memory Abstraction and Symbolic Analysis Framework for Executables[C]. *2013 IEEE International Conference on Software Maintenance*, 2013: 90-99.

[71] Cheng K, Li Q, Wang L, et al. DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware[C]. *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018: 430-441.

[72] Gotovchits I, van Tonder R, Brumley D. Saluki: Finding Taint-Style Vulnerabilities with Static Property Checking[C]. *2018 Workshop on Binary Analysis Research*, 2018.

[73] Balatsouras G, Ferles K, Kastrinis G, et al. A Datalog Model of Must-Alias Analysis[C]. *The 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, 2017: 7-12.

[74] Smaragdakis Y, Balatsouras G. Pointer analysis[J]. *Foundations and Trends in Programming Languages*, 2015, 2(1): 1-69.

[75] Ganesh V, Dill D L. A Decision Procedure for Bit-Vectors and Arrays[C]. *The 19th International Conference on Computer Aided Verification*, 2007: 519-531.

[76] Bellard F. QEMU, a Fast and Portable Dynamic Translator[C]. *The annual conference on USENIX Annual Technical Conference*, 2005: 41.

[77] Cadar C, Dunbar D, Engler D R. Klee: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Pro-grams[C]. *OSDI*, 2008, 8: 209-224.

[78] Cha S K, Avgerinos T, Rebert A, et al. Unleashing Mayhem on Binary Code[C]. *2012 IEEE Symposium on Security and Privacy*, 2012: 380-394.

[79] Yun I, Lee S, Xu M, et al. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing[C]. *The 27th USENIX Conference on Security Symposium*, 2018: 745-761.

[80] Zaddach J, Bruno L, Francillon A, et al. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares[C]. *Proceedings 2014 Network and Distributed System Security Symposium*, 2014: 1-16.

[81] Redini N, Machiry A, Das D, et al. BootStomp: On the Security of Bootloaders in Mobile Devices[C]. *26th USENIX Security Symposium*, 2017: 781-798.

[82] Redini N, Machiry A, Wang R Y, et al. Karonte: Detecting Insecure Multi-Binary Interactions in Embedded Firmware[C]. *2020 IEEE*

Symposium on Security and Privacy, 2020: 1544-1561.

[83] Rawat S, Mounier L, Potet M L. Static taint-analysis on binary executables[J]. 2011.

[84] Eom K J, Choi C H, Paik J Y, et al. An Efficient Static Taint-Analysis Detecting Exploitable-Points on ARM Binaries[C]. *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, 2014: 345-346.

[85] Choi Y H, Min J W, Park M W, et al. A Framework of Static Analyzer for Taint Analysis of Binary Executable File[C]. *Information Technology Convergence*. Dordrecht: Springer, 2013: 185-192.

[86] Feng Z N, Wang Z Y, Dong W Y, et al. Bintaint: A Static Taint Analysis Method for Binary Vulnerability Mining[C]. *2018 International Conference on Cloud Computing*, *Big Data and Blockchain*, 2019: 1-8.

[87] Lee J H , Avgerinos T , Brumley D. TIE: Principled Reverse Engineering of Types in Binary Programs[C]. *The Network and Distributed System Security Symposium*, 2011.

[88] ElWazeer K, Anand K, Kotha A, et al. Scalable Variable and Data Type Detection in a Binary Rewriter[C]. *The 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013: 51-60.

[89] Balakrishnan G, Reps T. DIVINE: DIscovering Variables IN Executables[C]. *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer, 2007: 1-28.

[90] Noonan M, Loginov A, Cok D. Polymorphic Type Inference for Machine Code[C]. *The 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016: 27-41.

[91] Wang R Y, Shoshitaishvili Y, Bianchi A, et al. Ramblr: Making Reassembly Great again[C]. *Proceedings 2017 Network and Distributed System Security Symposium*, 2017.

[92] Hu X, Shin K G. DUET: Integration of Dynamic and Static Analyses for Malware Clustering with Cluster Ensembles[C]. *The 29th Annual Computer Security Applications Conference*, 2013: 79-88.

[93] Zhang C, Song C Y, Chen K Z, et al. VTint: Protecting Virtual Function Tables' Integrity[C]. *Proceedings 2015 Network and Distributed System Security Symposium*, 2015.

[94] Prakash A, Hu X C, Yin H. VfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries[C]. *Proceedings 2015 Network and Distributed System Security Symposium*, 2015.

[95] Chandramohan M, Xue Y X, Xu Z Z, et al. BinGo: Cross-Architecture Cross-OS Binary Search[C]. *The 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016: 678-689.

[96] Brumley D, Newsome J, Song D, et al. Towards Automatic Generation of Vulnerability-Based Signatures[C]. *2006 IEEE Symposium on Security and Privacy*, 2006: 15-16.

[97] Shoshitaishvili Y, Wang R Y, Hauser C, et al. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware[C]. *Proceedings 2015 Network and Distributed System Security Symposium*, 2015.

[98] Balakrishnan G, Gruian R, Reps T, et al. CodeSurfer/x86—A Platform for Analyzing X86 Executables[C]. *The 14th international conference on Compiler Construction*, 2005: 250-254.

[99] Cova M, Felmetsger V, Banks G, et al. Static Detection of Vulnerabilities in X86 Executables[C]. *The 22nd Annual Computer Security Applications Conference*, 2006: 269-278.

[100] Wang T, Wei T, Lin Z, et al. IntScope: Automatically Detecting In-teger Overflow Vulnerability in X86 Binary Using Symbolic Exe-cution[C]. *Network and Distributed System Security Symposium*, 2009.

[101] Zhang B, Feng C, Wu B, et al. Detecting Integer Overflow in Windows Binary Executables Based on Symbolic Execution[C]. *2016 17th IEEE/ACIS International Conference on Software Engineering*, *Artificial Intelligence*, *Networking and Parallel/Distributed Computing*, 2016: 385-390.

[102] Cheng S Y, Yang J, Wang J J, et al. LoongChecker: Practical Summary-Based Semi-Simulation to Detect Vulnerability in Binary Code[C]. *The 2011IEEE 10th International Conference on Trust*, *Security and Privacy in Computing and Communications*, 2011: 150-159.

[103] Rawat S, Mounier L, Potet M L. LiSTT: An Investigation into Unsound-Incomplete yet Practical Result Yielding Static Taintflow Analysis[C]. *2014 Ninth International Conference on Availability*, *Reliability and Security*, 2014: 498-505.

[104] Lin J, Jiang L H, Wang Y S, et al. A value set analysis refinement approach based on conditional merging and lazy constraint solving[J]. *IEEE Access*, 2019, 7: 114593-114606.

程凯 于 2014 年在西安电子科技大学计算机科学与技术专业获得学士学位。现在中国科学院信息工程研究所网络空间安全专业攻读博士学位。研究领域为 IoT 安全、嵌入式设备安全。研究兴趣包括二进制逆向、静态分析、固件安全分析、漏洞挖掘。Email: chengkai@iie.ac.cn

宋站威 于 2017 年在北京工业大学计算机科学与技术专业获得硕士学位。现任中国科学院信息工程研究所研究实习员。研究领域为物联网安全、工控安全。研究兴趣包括二进制分析、模糊测试、漏洞分析与利用。Email: songzhanwei@iie.ac.cn

刘明东 于 2020 年在中国科学院大学计算机技术专业获得硕士学位。现任中国科学院信息工程研究所助理工程师。研究领域为工控安全、物联网安全。研究兴趣包括二进制分析、模糊测试、漏洞挖掘。Email: liumingdong@iie.ac.cn

于楠 于 2011 年硕士毕业于北京航空航天大学。现任中国科学院信息工程研究所助理研究员。研究领域为物联网安全方向, 主要从事通用服务平台设计研发。Email: yunan@iie.ac.cn

朱红松 于 2009 年在中国科学院计算技术研究所计算机体系结构专业获得博士学位。现任中国科学院信息工程研究所研究员。研究领域为网络空间安全。研究兴趣包括物联网安全、网络对抗、智能攻防、网络空间安全测量和威胁态势感知等。Email: zhuhongsong@iie.ac.cn

孙利民 于 1998 年在国防科技大学计算机科学与技术专业获得博士学位。现任中国科学院信息工程研究所研究员。研究领域为物联网安全、工控安全。研究兴趣包括工控系统漏洞挖掘与关联、在线设备发现与识别、工控系统入侵诱捕与行为分析、工控系统入侵检测与监管。Email: sunlimin@iie.ac.cn