知识、探索与状态平面组织的软件漏洞分析架构研究

袁子牧[1], 肖 扬[1,2], 吴 炜[1,2], 霍 玮[1,2*], 邹 维[1,2]

[1]中国科学院信息工程研究所, 北京 中国 100093
[2]中国科学院大学 网络空间安全学院, 北京 中国 100049

摘要 对于软件漏洞分析复杂度过高的现状问题, 本文认为其主要原因在于当前软件分析知识、技术及数据耦合程度高、各类知识与技术间缺乏有效编程接口连接, 因而提出了将软件漏洞分析解耦合为知识、探索、状态等三层平面的设计。其中, 状态平面可基于基础分析数据和既有的大数据操作接口表征程序分析状态及转换; 知识平面与探索平面分别对应漏洞分析知识与技术/工具集合, 本文从符号执行、污点分析、模式检测、模糊测试等现有技术类别中抽象出两平面间的知识与技术间的交互接口。在阐述三层平面的基础上, 本文例举了实际漏洞分析应用场景, 描绘出通过可编程接口连接各平面、以自由定制的方式发挥各平面间互补优势的愿景; 期望随之努力达到打通各类知识、技术间的互通门槛, 并融合数据处理技术以提升软件漏洞分析效能的效果。

关键词 软件漏洞分析; 知识平面; 探索平面; 状态平面; 可编程接口
中图法分类号 TP311.5 DOI 号 10.19363/J.cnki.cn10-1380/tn.2019.11.02

Research on The Software Vulnerability Analysis Architecture with The Knowledge, Exploration and State Plane

YUAN Zimu[1], XIAO Yang[1,2], WU Wei[1,2], HUO Wei[1,2*], ZOU Wei[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract We consider the reasons of high complexity on current software vulnerability analysis are the software analysis knowledge, tools and data coupling tightly, and lack of effective programming API interface to establish connection between analysis knowledge and technology, and therefore propose decoupled three planes of knowledge, exploration and state. Among them, the state plane can exhibit the status and transformation of software vulnerability analysis based on the basic analysis data and the operation interface on resilient distributed datasets; the knowledge plane and exploration plane map the set of software vulnerability knowledge and technology/tool respectively, and we extract the API interface between knowledge and technology from existing sorts of technology, such as symbolic execution, taint analysis, pattern detection and fuzz. On the basis of the three planes, three vulnerability analysis application scenarios are illustrated to depict the picture that planes are connected through programmatic interface, and their interactions can be freely customized to take the advantages of each of them; the expectation of our work is to lower the barriers between sorts of analysis technologies and knowledges, and combine with the frontiers of data processing technology to promote vulnerability analysis performance with effort devoted.

Key words software vulnerability analysis; knowledge plane; exploration plane; state plane; programmatic interface

1 引言

随着互联网、移动互联网、工业控制系统、物联网的快速发展, 软件呈现种类多样化、数量规模化、形态动态化的特点。相伴而生的安全漏洞数量也不断攀升, 漏洞机理及利用模式渐趋复杂。正因如

通讯作者: 霍玮, 博士, 研究员, Email:huowei@iie.ac.cn。
本课题得到国家自然科学基金(No.61602470, No.61802394, No.U1836209); 国家重点研发计划(No. 2016QY071405); 中国科学院战略先导(No.XDC02040100, No.XDC02030200, No.XDC02020200)资助。
收稿日期: 2017-12-19; 修改日期: 2018-04-03; 定稿日期: 2019-10-28

此, 当前分析人员完成漏洞分析任务的劳动复杂度极高。在具体漏洞分析实践中, 因目标软件工作机制、漏洞成因、各项挖掘技术组合过多, 往往需要已具备一定漏洞挖掘基础的分析人员根据目标软件的特点钻研一段时间, 才能设计出可实施漏洞挖掘组合方案。同时, 分析人员在方案的具体实施中, 也经常需要不断的尝试调整方法, 跨越较高的学习及实践门槛。本文作者认为提出有效的软件漏洞分析解耦合架构是解决当前漏洞分析复杂度极高现状问题的有效途径, 因而将软件漏洞分析分离为知识、探索、状态等三层平面, 设计并论述三层平面的可行性。

(1) 知识平面的提出基于当前软件漏洞分析对人员知识及经验过于依赖的现状。在近年来爆发的数起安全事件中[1], 高价值的漏洞往往仅被极少数黑帽团队所掌握, 依靠有特殊经验和技能的团队成员挖掘。尽管学界与业界不断改进方法, 使得单项漏洞挖掘技术越来越自动化, 但仍无法摆脱过度依赖人员个人判断和调整的现状。如 2016 年美国 DARPA 发起 CGC 比赛[2], 挑战自动化漏洞挖掘与利用, 七支参与决赛的顶级漏洞挖掘团队所研发的系统仅能根据比赛设置的检查点部分完成自动化处理的过程, 还不能处理实际程序。如能将软件漏洞分析知识从具体的分析事例中提取出来, 以知识图谱等具体方式组织知识平面, 将可使海量既有的漏洞分析知识传承下来, 减少对人员个人知识及经验的依赖。

(2) 探索平面的提出基于当前软件漏洞分析工具/技术众多, 需组合不同漏洞分析方法以应对漏洞机理渐趋复杂的现状。一方面, 可融合多项漏洞分析技术形成互补优势, 挖掘漏洞并探究其成因机理; 如可结合静态分析和动态符号执行以寻找释放后重用漏洞[3], 组合符号执行、静态分析以及文件格式知识进行定向符号执行[4], 合并模糊测试和符号执行以实现两种分析方法的优势互补[5], 组合静态分析和定向符号执行进行大规模漏洞挖掘[6-7]。另一方面, 可应用知识平面的所提取并联结的海量漏洞分析知识要素, 指导漏洞分析工具/技术的使用; 当前, 代码分析信息、软件构建原理、已有漏洞模式及已知漏洞机理等大量漏洞知识广泛分布在互联网中, 抽取并联结这些知识要素将为漏洞分析工具/技术的选择提供重要的参考, 如基于已有的漏洞描述信息及其在代码中的位置标注知识可扫描同源代码以发现复用漏洞[8]。

(3)状态平面表征对目标对象的漏洞分析历史轨迹及当前状况, 其可用目标对象的漏洞分析工具/技术的输入数据、输出数据及中间产出数据表示。如某目标软件在漏洞分析的过程中先后应用攻击面分析、静态审计、动态调试、模糊测试、Crash 分析、污点分析及 PoC (Proof of Concept)验证等技术, 其历史分析轨迹可用各项技术的输入、过程产出及输出数据序列表示。某项分析技术的过程产出及最终输出数据可为目标对象的下一步漏洞分析提供线索, 即数据所呈现的状态可作为在知识平面上开展漏洞分析知识推理的依据, 用于决策下一步探索平面应采用的漏洞分析技术以及应用方式。同时, 数据及分析状态的融合互通将有助于分治漏洞分析任务, 应用不同分析技术间的数据进行反馈, 如疑似漏洞、瓶颈路径、种子输入等数据, 可缓解或进一步解决程序分析状态空间爆炸、路径覆盖率低、自动化程度低等现有漏洞挖掘技术面临的局限性问题。

本文将在第 2 章概述并举例说明三层平面, 在第 3 章状态平面部分阐述数据产生及操作接口, 在第 4 章将现有软件漏洞分析相关工作区分知识和技术部分, 在第 5 章举例说明三层平面的交互场景, 在第 6 章总结全文。

2 三层平面及示例

软件漏洞分析的解耦合架构可设想为由知识、探索和状态等三层平面交互的体系, 其意义在于从分析架构层面整合当前漏洞分析过于分散、存在互通门槛的知识、技术/工具及数据。其中, 知识平面表达软件漏洞分析的各类模型、判断、方法等体系知识, 如源码漏洞逻辑推理知识、程序分析中间表示知识等; 探索平面基于知识指导, 针对目标对象特点编排对象分析数据, 使用多项分析工具/技术开展先后分析, 探索目标对象存在漏洞的可能性; 状态平面是探索平面先后各项分析结果的数据呈现, 通过对目标对象数据的组织与转换, 使具体的数据形态在各项分析工具/技术间流转, 最终发现并验证软件漏洞。本文将通过描述软件漏洞分析的三层平面的设想和例举实际漏洞分析场景的应用可能性, 描绘出通过可编程接口连接各平面、能以自由定制的方式发挥知识、探索与状态平面间互补优势的愿景; 并期望随之努力达到打通各类知识、技术间的互通门槛, 并融合数据处理技术以提升软件漏洞分析效能的效果。

在三层平面[①]中, (1)状态平面集中在数据组织与转换上, 在状态平面与探索平面间可定义或使用一套标准数据操作接口实现数据在分析工具/技术上的对接与流转, 如实现弹性分布式数据集(Resilient Distributed Dataset, RDD)的接口定义[9], 将软件漏洞分析技术体系直接移植至大数据处理快速计算引擎上; (2)知识平面与探索平面间因知识在工具/技术中的应用方式多样, 如方法应用、模型映射等, 可集中实现一套可编程控制接口, 达到使用软件自由定制知识与技术融合方式的目的, 类似的思想已在软件定义网络(software defined network, SDN)[10]中得到广泛应用; (3)充分融合软件漏洞分析知识、工具/技术和数据将能形成互补优势, 既能提升程序分析自动化程度, 又能通过剪枝、信息反馈、大数据处理等方式应对当前程序分析状态爆炸、路径覆盖率低等问题[5-7]。

举例来说, 如图 1 右侧所示, 给定一个分析目标(如某软件), 通过将知识平面漏洞模式模型知识映射到具体技术应用上, 开展值集、词法分析、抽象解释、模型检验等模式验证, 扫描目标代码及相应中间转换表示中是否存在既有模式的疑似漏洞, 其输出状态为疑似漏洞位置标注数据; 结合污点路径识别知识, 以疑似漏洞位置为起点, 开展污点传播、影响评估等分析探索, 输出关键字节数据和校验和数据等状态; 在模糊测试环节, 关键字节数据可融合样本评估知识构建测试输入种子变异集合, 校验和数据结合路径知识可植入到代码中, 最终通过漏洞检测知识分析各测试输入的影响并收集可能的漏洞样本数据供进一步验证。

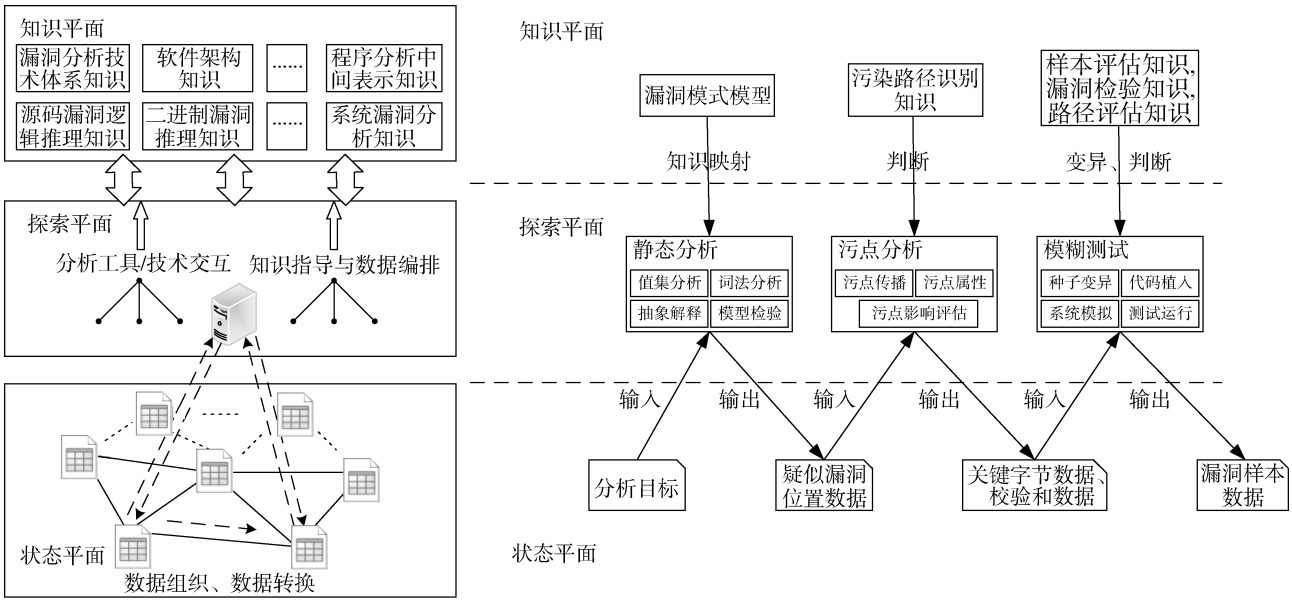

图 1　三层平面及示例

Figure 1　Example of three planes of knowledge, exploration and state

3　状态平面阐述

状态平面主要是通过数据集合表征软件漏洞分析任务的完成情况, 而数据集合的产生依赖于知识平面与探索平面间的交互。具体来说, 可用“状态-动作”序列形容分析任务的执行过程, 由知识与技术交互产生的数据集合所表征的状态将决定下一步交互方式的选择, 完成分析任务离不开三层平面间知识、技术与数据这三类要素的相互合作。本文将首先单独阐述状态平面以直观地表达软件漏洞分析任务的完成过程。

如上一章所述, 状态平面的数据组织与转换可直接应用即有的数据操作接口, 如 RDD 操作接口。本章节将直接沿用大数据处理快速计算引擎 Spark

① 在三层平面的具体形态上, (1)知识平面可选择用知识图谱表示, 即构建一系列的软件漏洞分析相关知识实体和关联, 为分析人员提供关联查询和推理分析功能; (2)探索平面可描述为包含符号执行、模糊测试等分析功能的组件集, 为知识平面提供功能接口层接入; (3)状态平面在本文用 RDD 描述, 也可用随着 Spark 发展而新设计的 DataFrame、DataSet 等表示。其中, 因本文的主要目的在于阐述软件漏洞分析解耦合三层平面架构, 并不涉及知识平面具体形态的描述。

针对 RDD 的动作(Action)和转换(Transform)的操作接口[11]描述状态平面的数据组织(对应动作, Action)和数据转换, 展示将软件漏洞分析技术体系移植至大数据引擎上的可能性。为此, 以下将从两个例子以及 RDD 操作接口的角度展开描述。

3.1 例 1-发现漏洞: 基于漏洞模式的分析→关键输入识别→模糊测试

此处沿用图 1 右侧示例过程, 针对分析目标依次开展静态分析、污点分析、模糊测试得到可能的漏洞样本数据。如图 2 所示, 给定一组目标对象为数据集 RDD1, 该组对象在计算引擎资源允许的情况下可被并行处理。

“基于漏洞模式的分析”: RDD1 中的目标对象调用 map(decompile(.))以及 map(vex(.))可转化为反汇编或 vex[12]等中间表示形式; 其中, map(.)在此处的含义为对调用其的 RDD1 数据集中的每个元素(即目标对象 1~目标对象 n)使用 decompile(.)和 vex(.), 返回新的数据集 RDD2。对于 RDD2 数据集, 可先将程序实现逻辑部分按基本块为单位进行划分, 然后使用基于漏洞模式的分析, 如 filter(vulModel1(.))和

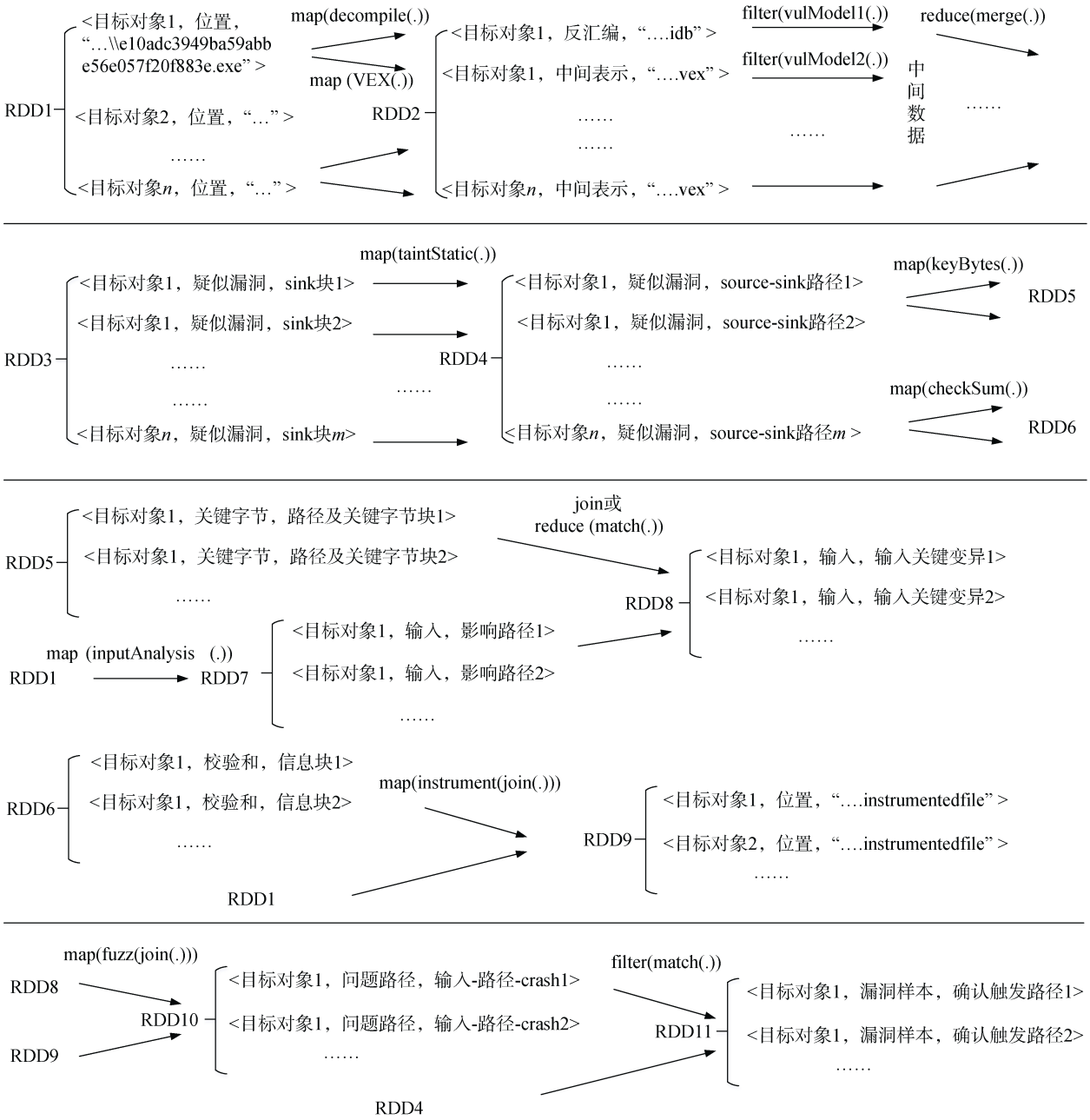

图 2 “例 1-发现漏洞”中数据状态及转换

Figure 2 Data state and its transition in “Example 1 –vulnerability discovery”

filter(vulModel2(.))表示分别对反汇编和中间形式的程序应用漏洞模式 vulModel1(.)和 vulModel2(.)静态扫描检查, 过滤并保留疑似引发漏洞的程序基本块, 通过 reduce(merge(.))聚集重复或应拼接的多基本块形成数据集 RDD3。

“关键输入识别”: RDD3 标注了疑似引发漏洞的单个或拼接起来的程序基本块, 调用 map(taintStatic(.))开展静态污点分析, 计算出从目标对象输入(称为 source 点)至疑似引发漏洞基本块(称为 sink 点)的多条污染路径, 形成数据集 RDD4。RDD4 可用于计算出路径中关键字节块(或称关键变量)以及对路径中的校验进行检查, 作为关键输入信息指导模糊测试; 如调用 map(keyBytes(.))找出影响路径选择的关键字节块并形成数据集RDD5, 调用map(checksum(.))找出路径中存在的校验并形成校验信息块数据集 RDD6。同时, 对数据集RDD1 调用map(inputAnalysis(.))开展输入分析(或称攻击面分析)得到输入参变量及其取值对于路径选择的影响数据集 RDD7。

“模糊测试”: 找出关键的输入变异有助于迅速提升模糊测试有效性, 可通过调用 reduce(match(.))聚合 RDD5 和 RDD7 找到输入中的关键变异字节块及其对疑似触发漏洞路径的影响, 形成数据集 RDD8。同时, 有相当一部分程序应用启动了校验和机制, 从而会导致模糊测试失败, 可通过调用 map(instrument(join(.)))在目标对象中插入一段重新计算校验和的代码, 形成插桩后的目标对象数据集 RDD9 绕过校验和。随后, 调用 map(fuzz(join(.)))连接 RDD8 和 RDD9 开展模糊测试(如多机并行测试)并收集问题路径形成 RDD10。最终, 可通过 RDD10 过滤 RDD4 中疑似路径, 确认可触发漏洞的路径集合 RDD11。

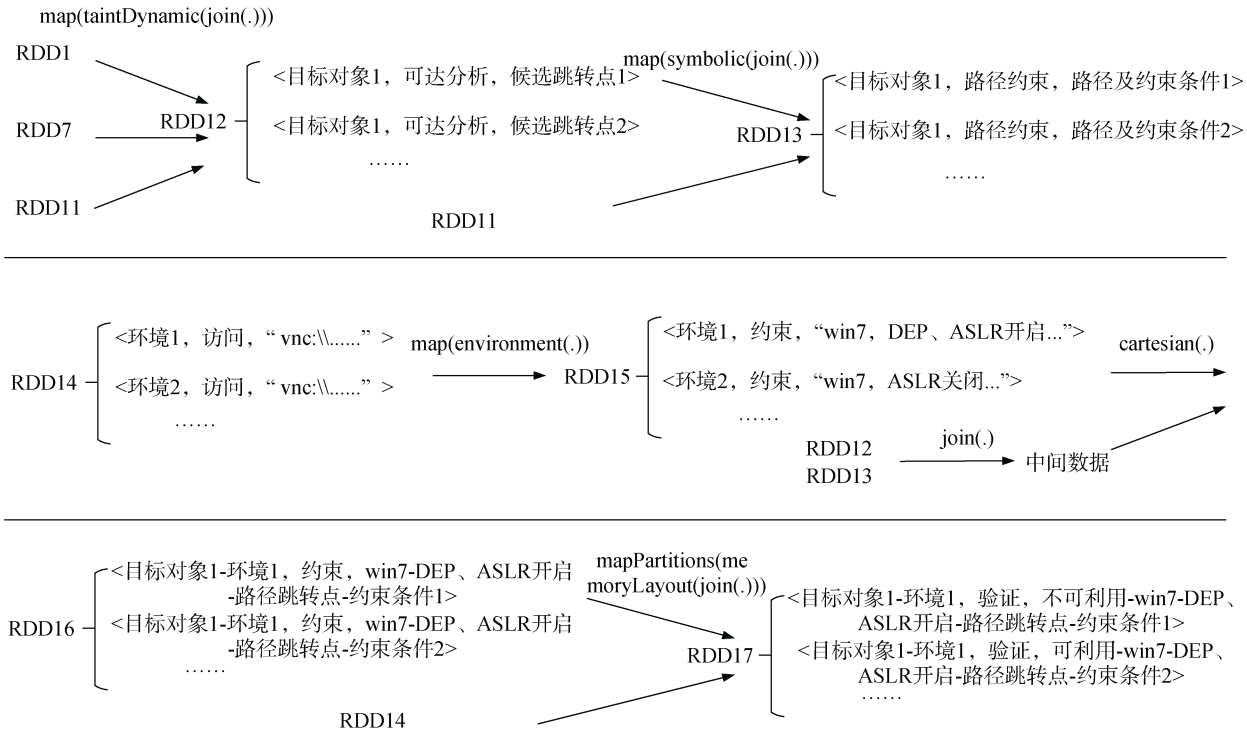

图 3 “例 2-验证漏洞”中数据状态及转换

Figure 3 Data state and its transition in “Example 2 – vulnerability validation”

3.2 例 2-验证漏洞: 动态污点分析→约束分析→利用模式

例 2-验证漏洞用于确认例 1 中所发现的漏洞是否可以在实际中使用, 如仅在 ASLR 机制[13]关闭情况可被利用有效提权等。

“动态污点分析”: 数据集 RDD1 为例 1 中目标对象集合, RDD7 为输入取值与路径选择的关联数据集, RDD11 为可触发漏洞的路径集合, 通过 map(taintDynamic(join(.)))综合这三个数据集结果, 并经过动态污点分析得到候选的控制流跳转集合 RDD12。通过构造特定输入, 可经由 RDD12 中控制流跳转点实现任意代码执行的目的。

“约束分析”: 连接 RDD12 中的候选跳转点以及 RDD11 所关联的触发漏洞路径集合, 调用 map(symbolic(join(.)))求解出各路径至跳转点的约束条件集合 RDD13。同时, 判断目标程序漏洞是否可在实际中使用需要结合其所在的运行环境考量, 如构建数据集 RDD14 包含一组验证环境, 进一步探测该组

环境中的约束, 如操作系统版本、DEP[14]和 ASLR 等机制是否开启, 形成环境约束数据集 RDD15。

“利用模式”: 组合路径约束和环境约束, 如 RDD13 和 RDD15 等, 形成约束集合 RDD16; RDD16 中所列约束可逐一或多机并行在各个环境中运行目标对象程序验证, 如调用 mapPartitions(memoryLayout (join(.)))针对内存建模, 并确认目标对象程序漏洞运行时内存布局是否处于可利用的模式之中[15]; 此处, mapPartitions(.)的用途为指根据 RDD16 中所列目标对象应运行环境做分类处理, 如通过条件判断目标对象的启动运行地址、命令及参数异同, 划分到环境 1、环境 2 或其他环境中; 最终, 结果数据集 RDD17 逐一列出各程序漏洞路径是否可被利用。

3.3 数据操作接口用途例举

本文所沿用描述的 RDD 接口作为通用的大数据操作接口可与探索平面的漏洞分析技术或工具融合, 在状态平面组织或转换目标对象分析数据, 在各项漏洞分析技术和工具间流转, 最终达到发现并验证目标对象程序漏洞的状态。

部分常用 RDD 操作接口及其可能的功能说明例举如表 1 中所示。

表 1 RDD 数据操作接口

Table 1 Operation interface of RDD data

操作接口	功能说明例举
makeRDD(.)/parallelize(.)	创建新数据集, 可用于将软件分析数据分组, 每一个分组应用不同的分析工具/技术进行分析; 如静态提取 PE 格式、ELF 格式可执行文件特征, 可划分为 PE 和 ELF 两个数据处理分区
map(.)/flatMap(.)/mapPartitions(.)/mapPartitions-WithIndex(.)	将可并行化加速执行同类数据元素放置到同一集合, 应用同一软件分析技术同时展开分析; 如规模化分析大量软件时, 可通过该函数部署多个分析线程
filter(.)	将候选数据元素放置到同一集合, 判断数据是否符合某些分析特征, 如疑似漏洞触发路径的可能性; 如判断候选检测路径是否符合预设漏洞模式模型
join(.)/cogroup(.)/cartesian(.)/leftOutJoin(.)/rightOutJoin(.)	用于连接不同的分析结果集合; 如示例 1 中通过 join 连接关键字节和攻击面信息用于模糊测试变异输入, 示例 2 中通过 cartesian 组合所有疑似漏洞路径和环境信息, 用以验证各组合下漏洞可用性
reduce(.)/reduceByKey(.)/aggregateByKey(.)	通过相同的键值聚集元素至同一集合; 如将不同漏洞检测模型所探测的路径合并; 将静态污点和动态污点分析所得出的路径合并

*除此之外, 还有 union、intersection 等集合操作用于分析结果集合的交并; count、foreach 等 rdd action 类函数可用于触发 map、flatMap 等 transform 类函数立即执行。

3.4 基础数据生成接口例举

软件漏洞分析领域涉及多种基础数据技术可生成供分析 RDD 数据集。该类数据集与 RDD 操作接口一道可为知识平面与探索平面(即知识与技术)的交互提供支撑。

部分常用基础数据接口及其可能的功能说明例举如表 2 中所示。

表 2 基础数据生成接口

Table 2 Generation interface of basic data

基础数据生成接口	功能说明例举
decompile(.)	获取 PE、ELF 等可执行格式文件的反汇编代码
LLVM(.)/VEX(.)/JavaBytecode(.)①等	获取源代码或二进制代码的中间表示, 如 Clang[16]用 LLVM 作为 C/C++源代码的中间表示, Valgrind[12]/Angr[17]将二进制代码转为 VEX 表示, Java 则依靠其 Bytecode 作为中间表示实现跨平台运行
lexer(.)/parser(.)/syntax(.)等	利用编译技术对源代码程序进行分析, 如 lexer(.)开展词法分析并获取一系列的符号定义, parser(.)基于词法分析符号输出进行语法分析并生成语义树, syntex(.)则在其基础上开展语义分析
controlFlow(.)/dataFlow(.)/callGraph(.)/pointerAnalysis(.)等	该类接口可对程序开展流分析, 如 controlFlow(.)获取控制流分析路径, dataFlow(.)开展更为精确的数据流分析, callGraph(.)获取函数间调用关系, pointerAnalysis(.)获取指向分析结果, 如针对 Java 反射机制等
instrument(.)	对程序进行插桩, 如可用于开展动态污点分析, 或在模糊测试时用于插入校验绕过程序片段等
environment(.)	获取程序所运行的环境参数, 如当前运行操作系统及其版本号、防御机制是否存在、程序进程信息统计等

4 知识平面与探索平面的现状及交互

相比状态平面的阐述, 本文将更为重点地讨论知识平面与探索平面间的现状与设想交互。这一方面是基于传统软件漏洞分析领域研究更集中在分析方法层面的考虑; 另一方面也因当前各类软件漏洞分析方法中知识与技术耦合程度高, 需重点讨论分解的可能性。在该部分, 本文将就具体的漏洞分析技术和分析知识结合进行探讨, 试图从分析应用中区分技术与知识, 并列举两者间的接口。其中, 在接口探讨部分, (1)集中关注漏洞知识在技术中的应用、使用知识解决技术瓶颈、跨技术

① 为了描述整洁起见, 本文后续也会采用与 IntermediateRepresentation(type), type=LLVM、VEX、JavaBytecode 等类似的接口描述, 其与 LLVM(.)/VEX(.)/JavaBytecode(.)等价。

的知识交互等三方面内容, 而(2)不考虑技术内的方法知识; 如关注漏洞分析规则知识在符号执行中的应用, 不考虑符号执行本身分析中使用约束求解器判断路径条件可满足以及具体采用正向或逆向符号执行的过程①。

以下将分别就符号执行、污点分析、模式检测、模糊测试、代码比对、流分析等漏洞分析关键技术的现状以及知识与技术间的交互接口展开叙述。

4.1 符号执行

符号执行对所指定的代码区域或路径开展符号化约束分析, 获取输入或变量取值与具体路径间的关联。本文在表 3 中列举符号执行相关工作, 区分其中的知识部分(对应知识平面)和技术部分(对应探索平面), 并总结两者间的接口。同时, 因分析工具可能涉及到多项技术和知识交互, 故在表中区分符号执行具体技术和工具。

表 3 符号执行相关知识与技术

Table 3 Enumeration of knowledge and techniques of symbol execution

类型	知识部分	技术部分	接口抽象及参考文献
符号执行技术	危险路径知识、程序缺陷知识等→一阶逻辑知识, 如将程序漏洞检测知识表示为符号变量的取值范围, 即转为可求解的一阶逻辑表达	将路径条件表示为对于符号的取值约束, 并运用求解器将取值约束转化为 SMT (Satisfiability modulo theories)[18]等问题求解	logicalExpression(.) symbolic(.) match(logicalExpression(.)) //参考文献[19-23]
工具		Clang[16]、KLEE[24]均以 LLVM 字节码为输入, Clang 可为目标对象构建语法树上下文, 并开展词法和语法分析; PathFinder[27]基于 Java 字节码和状态图可处理整数及实数类型约束	LLVM(.) JavaBytecode(.) lexer(.) 词法分析 parser(.) 语法分析 //参考文献[16, 24-27]
结合具体执行(混合执行)	程序漏洞的一阶逻辑表达知识, 如根据逻辑表达式构造具备触发漏洞取值的测试用例	使用混合执行(concolic testing)将具体执行与符号执行结合, 并可生成随机化测试用例	symbolicInputofFuzz(.) //参考文献[24, 26, 28-29]
结合数据流或污点分析	由数据流或污点分析得出的漏洞路径知识, 提供为符号执行输入	在通过数据流或污点分析出疑似漏洞后, 可通过符号执行计算触发漏洞的输入范围	symbolicValue(.) //参考文献[30]
结合漏洞验证环节	漏洞验证环节所指定的路径知识, 如控制流跳转点相关路径, 提供为符号执行输入	在漏洞验证环节通过符号执行对触发路径进行可达分析, 如判断路径约束是否可满足	symbolicValue(.) //参考文献[31]
应对路径爆炸(路径选择)	路径选择知识, 如深度优化、覆盖优化知识(KLEE[24]), 最优状态知识[25]、未覆盖控制流区域知识[32]、已探索和翻转分支增益综合知识[33]、测试用例知识[34-35]、上下文知识[36]等	可根据程序执行状态选择定制多种路径制导方式。概括相关工作, 其技术可总结为记录已有执行状态, 基于策略选择某状态继续探索路径	pathSelection(.) //参考文献[24-25, 32-36]
应对路径爆炸(路径约减)	路径约减知识, 其根据目标及程序行为而有所不同, 如用读写内存位置知识[37], 程序新旧版本路径差异知识[38], 与输入关联的程序片段知识[26], 路径后缀知识[39]	发现拥有相同执行效果路径集合, 以此约减需要执行的路径数量。如用读写内存位置判断是否有新的行为[37], eXpress 应用新版本与旧版本的路径差异约减重复路径[38], S2E 分析与输入关联的程序片段约减路径[26], 依据路径后缀判断是否约减[39]	pathReduction(.) //参考文献[26, 37-39]
性能问题(路径爆炸)	符号执行树分割知识, 如静态划分知识[40], 探索过程中的动态划分知识[41]等	通过划分路径并行执行的方法应对路径爆炸	pathPartition(.) //参考文献[40-41]

相关知识与技术交互接口抽象总结如下

- logicalExpression(.): 用于将待检查程序漏洞检测等知识映射为一阶逻辑表达式;
- match(logicalExpression(.)): 比较漏洞检测知识所映射的一阶逻辑表达式与指定路径约束的一阶逻辑表达式, 判断两式取值是否有交集(如有交集, 则可判断疑似存在漏洞);
- symbolic(.): 将指定路径的条件表示为符号取值约束的一阶逻辑表达式;
- symbolicValue(.): 计算一阶逻辑表达式给定约束的取值范围;
- symbolicInputofFuzz(.): 符号执行与模糊测试的交互接口, 根据路径约束取值生成经过目标程序指定路径的输入测试用例;

① 接口抽象以知识是否起引导作用为准, 即是否因不同知识为输入或参数而导致分析结果发生变化。以自动驾驶类比形容, 技术层面提供自动驾驶的基本功能模块, 知识层面起着根据拥堵等信息导航的作用, 因不同的导航判断而使行车路线不同。

- pathSelection(.): 选择特定的路径集合进行探索, 避免处理过多路径;
- pathReduction(.): 约减拥有相同执行效果的路径集合, 避免重复探索;
- pathPartition(.): 划分路径集合, 用以达到并行执行的效果。

LLVM(.)、JavaBytecode(.)、Lexer(.)、Parser(.)等作为基础数据接口已在 3.4 节列出, 如 LLVM 和 Java 字节码均为程序的中间表示, 一阶逻辑表达式可在其表示的基础上构建。

以工具 KLEE 为例, 将其基于符号执行构造测试用例的过程结合接口进行描述, 可如图 4 所示。以 C 源码项目为输入生成 LLVM 中间表示, 在该中间表示的基础上开展基于数据流或污点分析等漏洞检测并指定疑似有问题的路径集合, 通过符号执行与求解计算路径约束, 并以该约束构造针对 C 源码项目的输入测试用例。

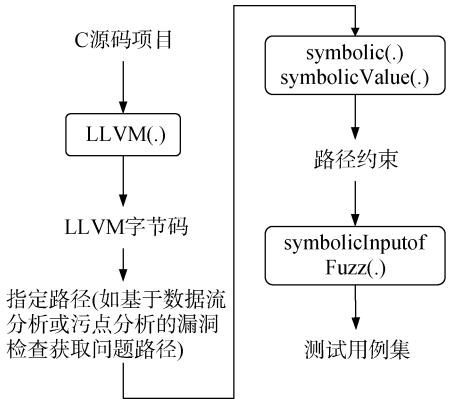

图 4 工具 KLEE 流程描述

Figure 4 The workflow of KLEE

4.2 污点分析

污点分析可用于标记出不被信任的数据在程序中的传播路径信息, 找到疑似存在安全问题的源(source)和泄露点(sink)。与 4.1 节类似, 本文在表 4 中列举污点分析相关工作。

相关知识与技术交互接口抽象总结如下

- vulModel(type=taint,.): 用于直接检测污点类型的漏洞, 如可由外部输入未经检查写入数据库, 则可认为存在 SQL 注入的危险;
- taintStatic(.): 静态分析污点数据在程序路径上的流动, 如以危险操作等 sink 点为起始传播点, 则一般需找到外部输入作为 source 点; 反之, 以外部输入等为 source 点, 则跟踪查找危险操作等 sink 点;
- taintDynamic(.): 以追踪程序进程运行的方式分析污点数据在程序路径上的流动;
- exploitSignature(.): 用于基于污点分析结果生成特征码, 用于漏洞验证利用;
- taintStaticStrategy(.): 可生成上下文、流、域、对象等别名传播规则, 其输出可用于 taintStatic(.)输入;
- taintDynamicStrategy(.): 有选择性的挑选系统指令并形成动态传播规则, 其输出可用于 taintDynamic(.)输入;
- undertaintingStrategy(.): 针对动态污点分析下传播不完全的欠污染问题, 生成融合动静态传播知识的规则, 作为 taintStatic(.)和 taintDynamic(.)的输入;
- overtaintingStrategy(.): 针对静态污点分析下传播路径过多的过污染问题, 生成限制传播规则, 作为 taintStatic(.)和 taintDynamic(.)的输入。

其中表 4 中所列举的 symbolicValue(.)可用于求解污点传播中的约束, 判断传播影响以及解决过污染问题; controlFlow(.)、callGraph(.)、pointerAnalysis(.)、decompile(.)等作为基础数据接口, 为污点分析传播规则提供传播数据。

以工具 TaintCheck 为例, 其污点分析过程如图 5 所示。该工具以 x86 架构下的程序为输入, 首先将非信任数据来源均标记为污点, 随后开展指令级的动态污点分析形成污点数据标记, 最后通过标记和污染内存信息生成特征码用于入侵检测等用途。

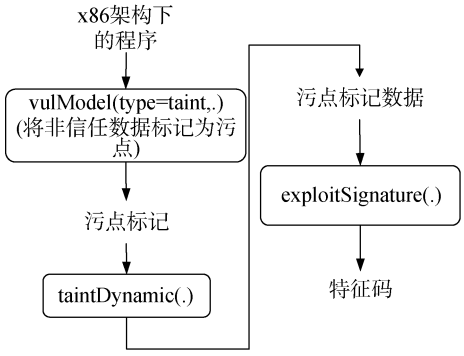

图 5 工具 TaintCheck 流程描述

Figure 5 The workflow of TaintCheck

4.3 模式检测

模式检测基于源代码抽象表示或二进制代码中间表示等, 设计特定模型检测漏洞或证明程序代码证明, 包括源码模型检测、定理证明、二进制漏洞模式检测等。模式检测相关工作如表 5 所示。

表 4　污点分析相关知识与技术

Table 4　Enumeration of knowledge and techniques of taint analysis

分类	知识部分	技术部分	参考文献
静态污点分析	基于漏洞模型的标注知识, 如标注某敏感操作为 sink 点, 且可由外部输入作为 source 点到达, 则判断疑似存在漏洞; 具体知识类型包括 SQL 注入、跨站脚本、恶意执行、隐私泄露等标注	静态分析控制依赖关系或数据依赖关系在程序路径上的流动	vulModel(type=taint) taintStatic(.) //参考文献[42-47]
工具		Pixy[48]将 PHP 源码解析为程序控制流图, 并分析控制流图上的依赖关系; TAJ[49]针对 Java 字节码构建包括反射机制处理的程序调用图和指向分析, 并应用混合切片技术分析依赖关系; IDA[50]、LCLint[51]、Satum[52]等针对二进制反汇编代码开展静态的依赖关系分析	controlFlow(.) callGraph(.) pointerAnalysis(.) decompile(.) //参考文献[48-52]
动态污点分析	基于漏洞模型的标注知识, 如以访问文件或网络行为的系统 API 作为 source, 以可能触发漏洞的系统库函数作为 sink 等在运行程序中的动态标注	通过追踪进程中的数据使用和程序执行[54]、采用动态插桩[55]、影子内存[56]等技术实现动态依赖关系分析	vulModel(.) taintDynamic(.) //参考文献[12, 53-57]
工具		TaintCheck[58]通过指令级的内存污染标记开展污染分析, Argos[59]借助 qemu 分析指令污点标记的继承关系, TaintDroid[60]在 Android Dalvik 虚拟机中标记污染数据。其中 TaintCheck 和 Argos 均有在污点分析完成后生成特征码的功能	exploitSignature(.) //参考文献[58-60]
结合符号执行	由污点分析得出的漏洞路径知识, 提供为符号执行输入	在污点传播中对操作数被污染, 但对最终结果不造成污染的情况进行传播分析, 如判断结果是否为外部输入完全[61]或部分[62]可控	symbolicValue(.) //参考文献[61-62]
降低时空开销(静态别名分析)	别名传播规则知识, 如上下文敏感、流敏感、域敏感、对象敏感等规则, 用于定向分析	基于传播规则, 开展基于需求定制的上下文、流、域、对象等依赖关系分析, 从而减少污点分析的时空开销	taintStaticStrategy(.) taintStatic(.) //参考文献[49, 63-65]
降低时空开销(动态效率优化)	动态选择传播规则知识, 如快速路径优化知识[66], 基于 load 和 store 指令的优化规则知识[67], 延迟例外(deferred exceptions)传播知识[68]等	基于传播规则, 有选择性的对系统指令开展依赖关系分析, 从而减少污点分析的时空开销	taintDynamicStrategy(.) taintDynamic(.) //参考文献[66-68]
解决欠污染问题	动态及静态传播融合规则知识, 解决动态污点分析下的欠污染问题	采用动态执行轨迹以及离线静态分析判断隐式流污点传播	undertaintingStrategy(.) taintStatic(.) taintDynamic(.) //参考文献[69-70]
解决过污染问题	污点限制传播规则知识, 如常数差异分支选择知识[71]、信息完整性限制规则[62]等	基于限制传播知识, 应用符号执行等技术选择程序分支开展依赖关系分析	overtaintingStrategy(.) taintStatic(.) taintDynamic(.) symbolicValue(.) //参考文献[62, 71-72]

相关知识与技术交互接口抽象总结如下

- abstraction(.): 用于建立源代码程序的抽象表示, 如 type=counterexample、predicate、controlFlowGraph、bool 分别指基于反例、谓词、控制流图、布尔表达式的抽象;
- vulModel(type=abstraction,.)①: 在程序源代码的抽象表示上, 映射漏洞模型进行检测;
- vulModel(type=theorem,.): 在程序源代码的逻辑表达式基础上, 通过 SAT、Simplify 等方式证明程序的正确性; 若不正确, 则有可能存在程序漏洞;
- vulModel(type=binary,.): 在二进制代码反汇编或中间表示的基础上, 基于漏洞模式知识进行漏洞检测。

decompile(.)、REIL(.)、VEX(.)、SIL(.)、AST(.)、inputAnalysis(.)、controlFlow(.)等均可作为基础数据接口; 其中, decompile(.)、REIL(.)、VEX(.)、SIL(.)、AST(.)等为反汇编或中间代码表示; inputAnalysis(.)获取输入与路径的关联映射, 与 controlFlow(.)一起构成基于流的漏洞分析基础。

①本文为了描述整洁、清晰, 用 vulModel(type=abstraction)、vulModel(type=theorem)、vulModel(type=binary)、vulModel(type=taint)等统一描述基于模型的漏洞检测并用类型 type 加以区分, 但并不意味着它们必须实现在同一接口下。

表 5 模式检测相关知识与技术

Table 5 Enumeration of knowledge and techniques of pattern detection

分类	知识部分	技术部分	参考文献
源码模型检测	漏洞模型知识, 如分布式软件安全漏洞模型[73]、基于 API 调用规则的安全模型[79]、C 语言程序安全属性约束[80]等	建立程序代码抽象并构建状态转换图, 在图上映射模型知识进行程序漏洞检测	abstraction(.) vulModel(type=abstraction) //参考文献[73-76]
工具		应用多种技术抽象并检测漏洞, 如采用基于反例的自动抽象技术[77-78], 使用谓词抽象自动建模程序控制结构[79], 基于控制流图的抽象[80]	abstraction(type), type=counterExample、predicate、controlFlowGraph… //参考文献[77-80]
定理证明	安全属性刻画知识, 如表示前置/后置条件及循环不变式的断言知识[52, 83]等表达形式, Saturn[52]将安全断言知识转化为布尔表达式; ESC/Java[85]直接使用 Java 安全注释知识作为定理证明输入等	将程序转换为逻辑表达形式, 证明给定安全约束是否被满足。逻辑表达形式包括流程图[81]、布尔公式[52]等。证明方法包括 Floyd 不变式断言[81]、SAT 证明[82]、SMT 证明[18, 23]等	abstraction(.) vulModel(type=theorem) //参考文献[18, 23, 52, 81-83]
工具		Saturn[52]将 C 程序翻译为 SIL 中间语言并转换为布尔表达式, 基于安全属性刻画开展 SAT 证明; ESC/Java[85]将程序转换为抽象语法树, 并应用 Simplify[86]定理证明安全注释是否满足; Coverity[84]也在其工具中部署了基于布尔表达式的 SAT 证明技术	SIL(.) AST(.) abstraction(type=bool) vulModel(type=theorem)如 SAT、Simplify 定理 //参考文献[52, 84-85]
二进制漏洞模式检测	漏洞模式知识, 如整数类溢出漏洞模式[87], strcpy 等不安全函数知识[93], 循环写内容知识[92]等, 也可通过编程调用 BinNavi、Angr、BAP 等平台接口映射实现自定义模式	将二进制程序反汇编, 进一步转换为中间表示, 如 REIL[88]、VEX[12]等, 基于漏洞模式知识抽取可控变量[89]及约束条件[90-91]进行漏洞检测	decompile(.) REIL(.) VEX(.) vulModel(type=binary) //参考文献[12, 87-93]
工具		BinNavi[94]将反汇编结果转换为 REIL 表示, 提供路径定位、数据流分析及控制流分析等用于开展漏洞模式分析的 API 接口; Angr[17]、BAP[95]等平台也提供将二进制程序转为中间表示, 并提供用于漏洞模式分析的丰富编程接口。	inputAnalysis(.) controlFlow(.) dataFlow(.) //参考文献[17, 94-95]

以工具 BinNavi 为例, 其作为分析工具平台可按照如图 6 所示的流程工作。以二进制代码为输入, 借助 IDA pro 获取反汇编代码, 并可在反汇编代码上开展自定义的输入与路径关联、数据流、控制流等分析来获取可控变量及约束条件, 在分析结果的基础上应用基于模式的漏洞分析以发现疑似漏洞。

4.4 模糊测试

模糊测试为目标程序构造输入并监视其运行, 期望找到非预期输入使程序运行异常, 从而发现程序漏洞。模糊测试相关工作如表 6 所示。

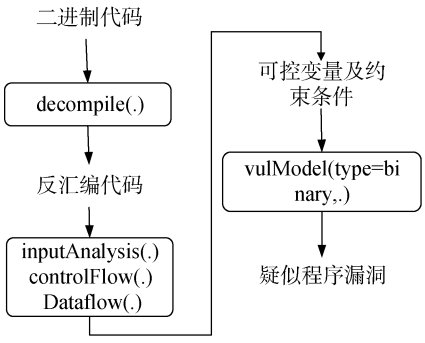

图 6 工具 BinNavi 流程描述

Figure 6 The workflow of BinNavi

相关知识与技术交互接口抽象总结如下

- inputofFuzz(type,.): 基于目标程序输入类型构造测试用例, 如 type=file、networkProtocol 等输入数据模型类别;
- fuzz(strategy, exception,.): 基于测试用例构造策略和异常表征知识开展模糊测试, 以获取程序崩溃等异常;
- smartFuzz(.): 组合 vulModel(.)、taintStatic(.)、taintDynamic(.)、symbolicInputofFuzz(.)、symbolic(.)、heuristicStrategy(.)等函数, 即综合基于模式的漏洞分析、污点分析、符号执行、约束求解构造测试用例、启发式构造测试用例等, 尽可能以最小代价获取程序运行异常;
- heuristicStrategy(.): 在智能模糊测试中, 基于已有的测试结果, 依据启发式策略整理结果数据, 以用于生成新的测试用例;
- keyBytes(.): 结合模糊测试用例和符号执行约束找出影响路径选择的关键输入字节块;
- checkSum(.): 结合模糊测试用例和符号执行约束找出程序中校验和检查部分代码, 生成校验和绕过程序片段;

除上述接口抽象之外, 其他基础数据接口如插桩 instrument(.)后的程序除可作为动态污点分析输入外, 也可插入校验和绕过程序片段, environment(.)可为程序测试运行提供环境参数输入参考。

以工具 Peach 模糊测试流程为例, 如图 7 所示, 其首先根据输入配置文件创建相应组件并开始构造测试用例开展模糊测试, 在测试的同时通过代理获取目标的状态和输出结果, 随后根据启发策略不断变异原有测试用例, 针对测试目标循环反复开展模糊测试。

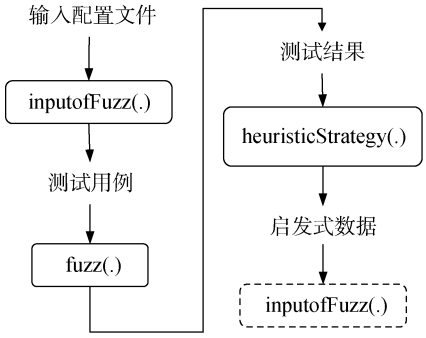

图 7 工具 Peach 流程描述

Figure 7 The workflow of Peach

表 6 模糊测试相关知识与技术

Table 6 Enumeration of knowledge and techniques of fuzz testing

分类	知识部分	技术部分	参考文献
模糊测试	(1) 目标程序输入数据关联知识, 包括输入数据元素和结构等; (2) 测试用例构造策略知识, 包括生成和变异等; (3) 异常表征知识, 如读写或访问内存异常等	基于目标程序输入构造测试用例, 在监视环境中以测试用例执行目标程序, 记录执行中的异常情况	inputofFuzz(.) fuzz(strategy, exception) //参考文献[57, 96-101]
工具		Peach[103]解析输入配置文件生成数据模型并创建相应组件开展测试, 在每一测试用例中通过状态机记录状态, 通过监视器观察异常状态, 通过变异器变换数据模型元素; Sulley[105]针对网络协议, 使用数据生成部件提交协议请求, 基于请求的连接和组合形成测试用例空间, 在测试时检查运行日志并决策下一步请求如何生成	inputofFuzz(type), type=file、networkProtocol 等数据模型 //参考文献[102-107]
智能模糊测试(综合多项分析技术)	智能模糊测试所涉及的交叉知识如下: (4) 漏洞模型知识, 用于识别程序敏感位置; (5) 敏感位置知识, 用于从程序入口到敏感位置的污点分析; (6) 危险输入及路径知识, 用于符号执行获取路径约束求解; (7) 路径启发知识, 用于测试用例构造策略知识	智能模糊测试综合基于模式的漏洞分析、污点分析、符号执行及路径约束求解等技术, 尽可能以最小代价找到可能产生漏洞的路径集合, 并针对该集合开展覆盖测试	vulModel(.) taintStatic(.) taintDynamic(.) symbolic(.) symbolicInputofFuzz(.) //参考文献[29, 108-111]
工具		Dart[28]初始阶段随机生成测试输入, 跟踪捕捉错误并随后导向性生成新测试输入; Smart Fuzzing[112]在测试用例中不断识别输入与程序控制间的关系, 启发式缩小输入测试状态空间; BitBlaze[113]平台综合静态分析、污点分析、符号执行选择测试和求解路径	heuristicStrategy(.) //参考文献[28, 112-113]
结合基于模式的漏洞分析	(4)漏洞模型知识, 包括文件操作是否可被控制、内存写入、逻辑错误、外部命令可被执行等, 用于识别程序敏感位置	通过基于模式的漏洞分析缩减候选验证和执行的路径集合, 减少模糊测试所应覆盖的路径范围	vulModel(.) //参考文献[89-91, 114]
结合污点分析	(5) 敏感位置知识, 可作为基于漏洞模型的 source 和 sink 点识别知识一部分, 用于获取程序入口到敏感位置的路径和相关输入元素	污点分析从程序敏感位置查找影响该位置的程序输入元素, 进一步可用于输入数据变异开展导向测试	taintStatic(.) taintDyanamic(.) //参考文献[115-116]
结合符号执行	(6) 危险输入及路径知识, 用于构造模糊测试用例, 如绕过校验和检查[108-109]	建立程序入口到敏感位置路径的约束关系, 求解出可满足约束的输入	keyBytes(.) checkSum(.) symbolicInputofFuzz(.) symbolic(.) //参考文献[108-110]

4.5 代码比对

代码比对是基于已知的漏洞标注知识进行源代码或二进制代码的比较, 如基于差异比较发现修补漏洞位置, 基于相似比较发现类似代码存在的漏洞; 其可分为补丁比对、源代码比对、二进制代码比对等三类。代码比对相关工作如表 7 所示。

表 7 代码比对相关知识与技术

Table 7 Enumeration of knowledge and techniques of code comparison

分类	知识部分	技术部分	参考文献
补丁比对	(1)代码比对知识, 用于制定比对策略, 如 EBDS 忽略编译和链接选项的差异; (2)基于代码比对差异的漏洞位置及成因识别知识	比对补丁后文件和补丁前文件, 找出修补的关键代码逻辑, 进而发现软件漏洞	patchDiff(.) vulModel(type= patchDiff) //参考文献[15, 117-119]
工具		Bindiff[117]可定制基于比对策略的函数匹配、基本块匹配、图同构匹配; EBDS[121]基于基本块指纹生成的方式构建比对图, 比较结构化差异	extract(type) diffComparison(type), type=function、block、graphIsomorphism、blockSignature //参考文献[117, 120-121]
源代码比对	已知漏洞知识, 如漏洞在源代码中的位置及路径标注	从文件、函数、代码行等粒度比对源代码克隆信息	sourceCodeDiff(.) //参考文献[8, 122-125]
二进制代码比对	已知漏洞知识, 如漏洞在二进制代码中的位置及路径标注	从文件、函数特征等粒度比对二进制代码克隆信息	binaryCodeDiff(.) //参考文献[126-127]

相关知识与技术交互接口抽象总结如下

- patchDiff(.): 以 diffComparison(.)的函数、代码块等多类比较结果为输入, 找出补丁修补前后的代码差异;
- vulModel(type=patchDiff,.): 根据补丁修补前后差异, 找出其中的漏洞修补代码逻辑;
- extract(type,.): 提取比对特征, 如 type=function、block、graphIsomorphism、blockSignature 分别表示函数、基本块、图同构、基本块指纹;
- diffComparison(type,.): 基于 extract(.)的提取结果, 选择的策略或比对类型进行代码比较;
- sourceCodeDiff(.): 可通过定义源代码文件、函数、代码行等的提取 extract(.)和比对 diffComparison(.)函数, 并输出相似性比对结果, 用于建立漏洞模型(如 vulModel(type=source CodeDiff))查找相似代码的漏洞;
- binaryCodeDiff(.): 可通过定义二进制文件、函数特征等粒度的提取 extract(.)和比对 diffCompari-son(.)函数, 并输出相似性比对结果, 用于建立漏洞模型(如 vulModel(type=binaryCodeDiff))查找相似代码的漏洞。

本文在表 7 未能列举出代码比对与其他技术交互的相关工作。但代码比对作为一种重要的静态分析方式, 也可视为一类疑似漏洞发现模式, 结合污点分析、符号执行、模糊测试等技术手段进一步确认漏洞存在, 以及位置和成因。

以工具 Bindiff 为例, 其工作流程可如图 8 所示。该工具以补丁前后版本的二进制文件为差异比对输入, 首先在函数的数据流图结构签名的基础上进行图同构匹配, 并基于该初始比对结果和函数调用图进行函数和基本块等比对, 最终根据图以及节点匹配结果综合获取比对差异。

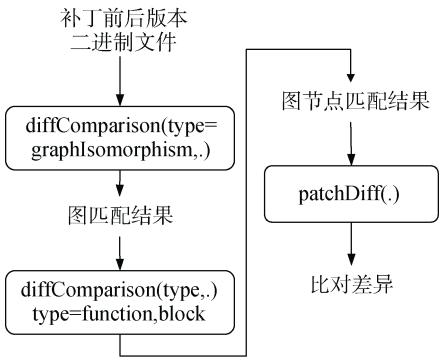

图 8 工具 Bindiff 流程描述

Figure 8 The Process of Bindiff

4.6 流分析

流分析是开展软件程序分析的一种基础方式, 可从控制流、数据流等角度发现或辅助分析软件程序漏洞, 完成漏洞验证。本文将相关工作分为流分析发现漏洞、辅助漏洞发现、基于控制流的漏洞验证、基于数据流的漏洞验证等四类。流分析相关工作如表 8 所示。

相关知识与技术交互接口抽象总结如下

- vulModel(type=flow,.): 检测可用流模型描述的程序漏洞, 如用数据流分析结合变量状态机描述缓冲区溢出漏洞等;
- memoryLayout(type=controlFlow,.): 评价针对程序漏洞的控制流跳转点在内存布局中的可利用情况, 即是否可构造输入改变程序控制路径, 达到执行攻击者代码的目的;

表 8　流分析相关知识与技术

Table 8　Enumeration of knowledge and techniques of flow analysis

分类	知识部分	技术部分	参考文献
流分析发现漏洞	漏洞分析规则知识, 如取值分析规则、用变量状态机描述缓冲区溢出漏洞等	以流敏感、路径敏感等流分析方式匹配漏洞分析规则对应的程序代码	vulModel(type=flow) //参考文献[63, 128-130]
流分析辅助漏洞发现	在基本的词法分析、语法分析、控制流分析产出知识的基础上, 提供语义、数据流等分析知识, 如分析 Java 反射等不易识别的调用	词法分析、语法分析、控制流分析、数据流分析变量别名以及完善程序调用图等	lexer(.) parser(.) syntax(.) controlFlow(.) callGraph(.) dataFlow(.) structure(.) configure(.) //参考文献[131-134]
工具		Fortify[135]将目标程序转为中间表示形式, 采用数据流、语义、结构、控制流、配置等引擎分析程序; Coverity[84]基于编译后生成代码, 也采用数据流、SAT 等引擎分析程序; FindBugs[136]针对 Java 字节码开展词法、控制流、数据流分析	接口同上 //参考文献[84, 135-137]
基于控制流的漏洞验证	关键控制流劫持点识别以及基于路径约束的输入构造知识	综合混合符号执行、动态污点分析、内存模型设计等技术, 利用程序对输入的错误处理改变程序控制路径	symbolic(.) taintDynamic(.) memoryLayout(type), type=controlFlow proofofConcept(.) //参考文献[138-141]
基于数据流的漏洞验证	关键变量识别及相关基于路径约束的输入构造知识	通过构造输入的方式改变程序数据流路径, 达到改变关键变量并访问受影响数据的目的	memoryLayout(type), type=dataFlow proofofConcept(.) //参考文献[142-143]

- memoryLayout(type=dataflow,.): 评价程序数据流中内存错误或关键变量被篡改造成的内存影响范围, 即是否可构造输入改变程序数据流路径, 达到权限提升等目的;
- proofofConcept(.): 基于 memoryLayout(.)的评估, 采用堆喷射等代码注入攻击、代码重用攻击手段完成漏洞验证。

此外, 表 8 中所列举 lexer(.)、parser(.)、syntex(.)、controlFlow(.)、callGraph(.)、dataFlow(.)、structure(.)、configure(.)等接口均可视为基础数据生成接口, 为基于流的知识与技术交互提供支撑; taintDynamic(.)和 symbolic(.)等可为漏洞流验证提供控制和约束分析支持。

以漏洞自动验证方案 Mayhem[139]的工作流程为例, 如图 9 所示。其以二进制程序为输入, 通过动态污点分析找出用户输入可控的 jmp 和 call 指令, 基于路径选择获取用户输入可控的路径集合, 并结合符号化内存建模, 在符号执行部分开展路径约束及可利用约束求解分析, 最终生成可满足约束条件的验证利用样本。

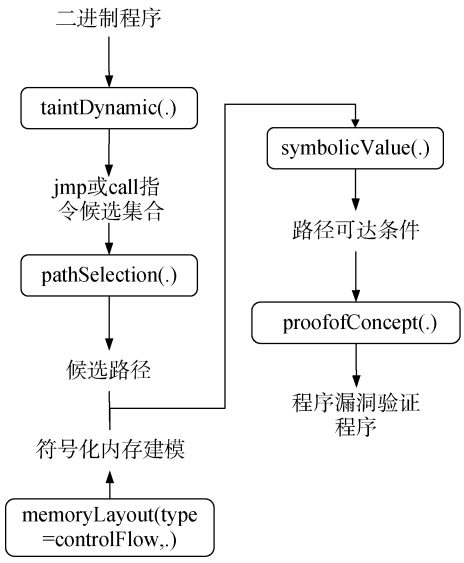

图 9　工具 Mayhem 流程描述

Figure 9　The workflow of Mayhem

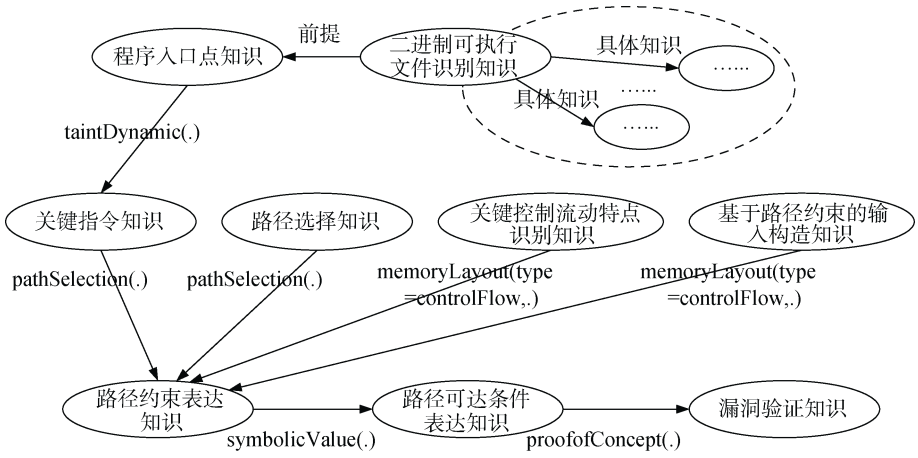

图 10　工具 Mayhem 的相关知识组织示例

Figure 10　Example of knowledge graph of Mayhem

4.7　知识平面与探索平面的交互

本文在第四章尽可能合理地描述出各分类下主要的知识与技术交互关系, 但并不意味着该描述是完整且不可更改的。事实上, 交互接口的设计标准并不统一, 且可随着系统开发深入而逐渐细化、随知识与技术的发展进而扩展或更新。

- 接口设计可更为细化。如 proofofConcept(.)在 4.6 节中指基于可利用性评估完成漏洞验证, 然而漏洞验证本身涉及多种方法, 如注入 Shellcode 攻击、重用代码片段攻击等; 在具体攻击验证方式上, 又可选择攻击代码动态生成的方式以绕过控制流完整性保护机制, 借助即时编译和二进制动态翻译技术实时生成、修改、删除、调用代码, 其中涉及多类知识与技术的交互。
- 技术分类可进行调整。如 4.5 节代码比对是基于已知漏洞知识的比较, 其方法为在代码差异结果上的漏洞模式检测, 因此也可被认为属于 4.3 节模式检测的一部分; 本文基于代码比对侧重于差异比较策略、模式检测侧重于应用已有专家知识而进行区分。同样, 4.6 节流分析也可区分为流分析漏洞发现与漏洞验证两大类。接口所属技术分类中也可基于不同的认识进行调整, 如本文基于 controlFlow(.)/dataFlow(.)等接口常为各项分析技术作为辅助技术使用而将其分类为基础数据接口, 也可将其归类至 4.6 节流分析。

本章的意义在于探讨从软件漏洞分析技术中剥离出知识的可能性, 实现知识平面和技术探索平面分离以拥有可编程定义发现及验证程序漏洞的能力, 从而达到可自主定制知识与技术融合方式、充分发挥知识、工具/技术和数据互补优势的目的。

4.8　知识平面的组织方式

知识平面可以软件漏洞知识图谱的方式组织。从发现或验证漏洞的实用价值角度看, 组织的关键在于知识实体的针对性以及知识实体关联对分析依赖关系的体现。因此, (1)知识实体及相关本体的设计需重点考虑知识平面与探索平面间的交互, 如归类交互接口参数和返回值相关的知识类型以及具体知识; 以图 9 工具 Mayhem 所使用 taintDynamic(.)函数为例, 可考虑依据其参数和返回值设计“二进制可执行文件识别知识”、“程序入口点知识”、“关键指令知识”这三类知识实体, 在识别目标文件为可执行文件的基础上找到程序入口点, 并通过动态污点分析的方式找到关键的 jmp 和 call 指令候选集合; (2)知识实体关联需体现知识平面与探索平面间交互的依赖关系, 如考虑以交互接口或参数依赖来表达知识实体之间的关联; 仍以 taintDynamic(.)函数为例, 可用“二进制可执行文件识别知识 $\xrightarrow{\text{前提}}$ 程序入口点知识”和“程序入口点知识 $\xrightarrow{\text{taintDynamic(.)}}$ 关键指令知识”表示获得关键指令对二进制可执行文件程序入口点知识的分析依赖关系。

基于上述思路, 工具 Mayhem 所用到的知识组织方式如图 10 所示, 图中每一个节点表示一类知识, 其可进一步关联到类别下的具体知识; 每一条边表示知识在软件漏洞分析中的前后依赖关系。

5　应用场景

本文提出将软件漏洞分析分离为知识、探索、状态等三层平面的解耦合架构, 期望通过可编程接口连接该三层平面, 实现漏洞分析自由定制以充分发挥各平面互补优势。因此, 本章将选择三个可能的

漏洞分析应用场景, 分别为发现并验证 0day 漏洞、基于补丁推测并验证已有漏洞、规模化比对并发现漏洞, 举例说明用三层平面解耦合架构描述和实现其分析过程的可行性。其中, 前两个场景分别侧重于从 0day 和 1day 的角度描述可编程式的知识与技术探索交互过程, 后一个场景侧重于揭示数据层面技术对软件漏洞分析的辅助功能。

5.1 可能场景 1: 例 1 和例 2 中组合多项技术分析漏洞的过程

场景 1 分析过程已在 3.1 节和 3.2 节叙述, 其伪代码如算法 1 所示。场景 1 以一组给定的目标对象和一组待验证环境为输入, 最终列出各目标对象在各验证环境下是否可被利用。需要指出的是, 3.1 节和 3.2 节叙述中只列举部分重要 RDD 数据集, 部分并不影响总体过程描述的 RDD 数据集并未被列出, 如 RDD17′。

算法 1: 组合多项技术分析漏洞

输入: 一组目标对象, 一组待验证环境

输出: 验证目标对象漏洞在各验证环境下是否可被利用

```
设 RDD1=一组目标对象;
设 RDD14=一组待验证环境;
//发现漏洞
RDD2′=RDD1.map(x=>decompile(x));
RDD2= RDD2′.union(RDD1.map(x=>VEX(x));
RDD3′=RDD2.filter(x=>vulModel1(x)).union(RDD2.filter(x=>vulModel2(x)));
RDD3=RDD3′.reduce((x,y)=>merge(x,y));
RDD4=RDD3.map(x=>taintStatic(x));
RDD5=RDD4.map(x=>keyBytes(x));
RDD6=RDD4.map(x=>checkSum(x));
RDD7=RDD1.map(x=>inputAnalysis(x));
RDD8=RDD5.join(RDD7);
RDD9=RDD1.join(RDD6).map(x=>instrument(x));
RDD10=RDD9.join(RDD8).map(x=>fuzz(x, …));
RDD11=RDD10.join(RDD4).filter(x=>match(x));
//验证漏洞
RDD12=RDD1.join(RDD7).join(RDD1).map(x=>taintDynamic(x));
RDD13=RDD12.join(RDD11).map(x=>symbolic(x));
RDD15=RDD14.map(x=>environment(x));
RDD16=RDD15.cartesian(RDD12.join(RDD13));
RDD17′=RDD16.join(RDD14).partitionBy(RDD15.keys());
RDD17=RDD17′.mapPartitions(x=>memoryLayout(x));
RETURN RDD17;
```

5.2 可能场景 2: 根据补丁推测漏洞成因并完成 PoC 验证

场景 2 以补丁文件和原始文件为输入, 试图分析漏洞成因并完成验证; 场景 1 与之相比以一组目标对象为输入, 因此可借助 RDD 数据集表示在 Spark 中获得加速, 而场景 2 则因仅限于补丁及原始文件并无必要用 RDD 数据集描述。在场景 2 中, 本文具体以 2017 年迅速爆发“永恒之蓝”(EternalBlue)事件为例展开叙述; 该事件涉及到 SMB 服务漏洞, 微软针对其发布的补丁为 MS17-010[144], 对应的漏洞编号为 CVE-2017-0143 至 0148。场景 2 可用知识与技术的交互式编程方式描述, 其伪代码如算法 2 所示, 以下文字部分揭示了调用相关技术接口并分析结果的知识思考过程:

① 调用 patchDiff(.)分析补丁修补位置, 发现其将某变量 v 从 WORD 类型修补为 DWORD 类型, 依据漏洞分析知识推断可能变量 v 在被读写的时候因长度截取出错;

② 反编译 srv.sys 文件, 并以变量 v 为 sink 点, 采用静态污点分析的方式找寻从输入点至变量 v 的路径集合 paths; 经过分析路径发现某入口处理函数为 SrvSmbOpen2(.), 它是通过 SMB 协议 SMB_COM_TRANSACTION2 命令调用;

③ 将 paths’指定为经函数 SrvSmbOpen2(.)入口到修补变量 v 的路径集合, 将”使变量 v 在 DWORD 中高位两个字节有非零值”作为附加约束条件, 调用 symbolicInputofFuzz(.)求解路径约束并构造 SMB_COM_TRANSACTION2 命令测试输入; 然而却发现测试输入构造为空, 其原因为 SMB_COM_TRANSACTION2 命令允许的 TotalDataCoun-t(即包的总长度)为 USHORT 类型, 无法构造 DWORD 高两个字节非零的值;

④ 无法直接通过 SMB_COM_TRANSACTION2 命令完成构造, 依据漏洞成因知识推断可能为其他命令转化而来; 沿着路径集合 paths 往前追溯, 发现 SMB 服务中的 TRANSACTION 类型由 SMB 头的 TID、PIDLow、UID、MID 决定; 随即调用 inputAnalysis(.)追踪这四个参数影响路径, 发现路径中的其中一处条件判断为采用最后一个包的参数值决定 TRANSACTION 类型; 因此, 推测可将连接中最后一个包的类型置为 SMB_COM_TRANSACTION2_SECONDARY, 而之前的包可选其他任意类型而被当作 SMB_COM_ TRANSACTION2 使得修补变量 v 高两个字节可为非零值, 通过 symbolicInputofFuzz(.)

基于路径约束构造相应输入可进一步验证所分析得出的漏洞成因;

⑤ 在验证过程中, 考虑到现代操作系统的防御机制, 为实现稳定利用, 最好能找到不改变控制流情况下的、基于数据流的利用方式, 因此需要发现触发内存错误的输入及影响范围。因此, 用vulModel(type=heap,…)扫描可用于占位的堆空间申请函数集合, 将候选的函数作为 sink 点开展静态污点分析 taintStatic(.), 并随后根据污点标记路径进行约束求解 symbolicValue(.); 可发现 SMB_COM_SESSION_SETUP_ANDX 命令中 wordcount 为 12 且部分字段缺失, 可调用到堆空间申请函数 SrvAllocate-NonPagedPool(.), 且通过构造输入包可使得其申请空间大小可控;

⑥ 验证“永恒之蓝”漏洞可被利用的思路为: 申请一组 srvnet 对象 buffer(即堆喷射), 紧接着释放掉其中部分 buffer, 并申请 srv 对象 buffer 占位被释放的 buffer, 发包产生越界写至 srvnet 对象 buffer 以触发验证代码执行。该思路能稳定利用的关键在于可控制申请并释放恰好或略大于 srv 对象的空间大小, 通过上一步 SMB_COM_SESSION_SETUP_ANDX 命令可达到该效果; 即构造 PoC 时(proofofConcept(.)), 断开命令 SMB_COM_SESSION_SETUP_ANDX 连接并释放空间, 再通过 SMB_ COM_TRANSACTION2_SECONDARY 命令填充该释放空间, 越界写触发漏洞跳转验证程序指定的代码片段执行。

算法 2: 补丁推测漏洞成因并验证

输入: “永恒之蓝”补丁 MS17-010, 提供 SMB 服务文件 srv.sys/srvnet.sys

输出: PoC 验证

// ①

diff=patchDiff(MS17-010, srv.sys);

// ②

decompile_code=decompile(srv.sys);

paths=taintStatic(decompile_code,diff);

// ③

设 paths’=paths 中从入口处理函数为 SrvSmbOpen2(.)到修补变量 *v*;

input=symbolicInputofFuzz(paths’ and 使变量 *v* 在 DWO -RD 中高位两个字节有非零值);

// ④

paths=inputAnalysis(SMB_Header.TID, SMB_Header.PIDLow, SMB_Header. UID, SMB_Header. MID);

依据 paths 中条件判断完成漏洞发现;

input=symbolicInputofFuzz(paths and 使变量 *v* 在 DWO -RD 中高位两个字节有非零值);

// ⑤

RDD_candidate_placeholders=makeRDD(vulModel(type=heap, decompile_code));

RDD_paths=RDD_candidate_placeholders.map(*x*=>taintStatic(*x*));

RDD_inputs=RDD_paths.map(*x*=>symbolicValue(*x*));

发现可稳定占位 srv 对象大小空间的 SMB 命令;

// ⑥

PoC=proofofConcept(memoryLayout(type=dataflow,environment));

RETURN PoC;

5.3 可能场景 3: 规模化比对软件与源代码库并发现漏洞

场景 3 通过规模化比对软件与源代码库发现复用关系, 并进一步根据源代码库的漏洞标注知识判断相关漏洞是否在软件中存在。本场景侧重于揭示数据层面技术对软件漏洞分析的辅助功能, 在用户层面通过对 RDD 数据集的接口调用即可实现对漏洞分析的规模化加速, 解决可能存在的性能瓶颈问题。该场景关键数据处理过程示意见图 11, 伪代码如算法 3 所示, 具体过程如下:

① 通过调用 mapPartition(.)将 RDD1 数据集(即规模化软件数据集合)映射至多台桌面(如设定分区数据个数上限, 基于云平台按需弹性生成桌面虚拟机), 通过在桌面自动安装方式提取安装过程中新产生的可执行文件(Windows 下 PE 格式文件)形成 RDD3 数据集;

② 基于一组定义的 PE 软件特征, 将 RDD3 中的可执行文件映射至多台处理机器上完成并行提取工作, 通过 reduceByKey(.)聚合各类特征, 并针对每类特征形成数据集(即 RDD8 实际为 RDD 数据集数组);

③ 基于一组定义的源代码特征, 将 RDD2 中的开源代码库映射至多台源代码特征提取机器上完成并行提取工作, 在提取后同时聚合各类特征, 并针对每类源代码特征形成数据集(即 RDD9 实际为 RDD 数据集数组);

④ 基于一组可用于快速比对的二进制代码与源代码共有特征, 如编译不变的 enum array, string array, const number array, switch/case 等特征, 采用分区并行比对的方式筛选出可能存在复用关系的软件与源代码放入 RDD10 数据集用于下一次精确比对;

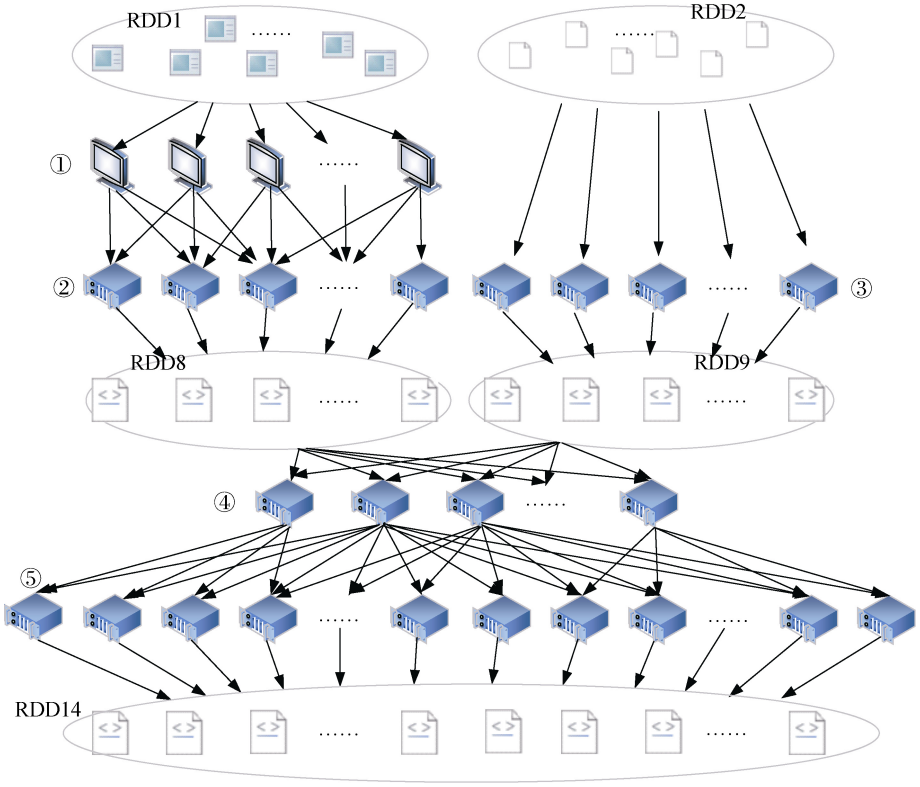

图 11　场景 3 关键数据处理过程示意

Figure 11　illustration of key data processing of scenario 3

⑤ 基于 RDD10 数据集筛选和一组可用于精确比对的二进制代码与源代码共有特征, 如 call graph, control flow graph、ssdeep[145]等特征, 并行比对进一步精确确认软件与源代码库间的复用程度; 最终, 可基于源代码库的已知漏洞标注信息查找相关漏洞在哪些软件中存在。

算法 3: 比对软件与源代码库发现漏洞

输入: 大批量 Windows 桌面安装软件、开源代码库以及漏洞在库中的标注

输出: 软件因引用开源代码库而产生的漏洞

设 RDD1=大批量 Windows 桌面安装软件;

设 RDD2=待比对的开源代码库;

设 binary_feature_set=[file, function, data_structure,…];//一组 PE 软件特征类型

设 source_feature_set=[file, function, data_structure,..];//一组源代码特征类型

设 rapid_feature_set=combination1(binary_feature_set, source_feature_set);//快速比对特征集

设 accurate_feature_set=combination2(binary_feature_ set, source_feature_set);//精确比对特征集

// ①

RDD3=RDD1.mapPartition(*x*=>extract(*x*,type=pe_file));

// ②

RDD4=RDD3.mapPartition(*x*=>extract(*x*,type=binary_feature_set));

RDD6=RDD4.reduceByKey((*x*,*y*)=>*x*.append(*y*));

RDD8=RDD6.map(*x*=>makeRDD(*x*));

// ③

RDD5=RDD2.mapPartition(*x*=>extract(*x*,type=source_feature_set));

RDD7=RDD5.reduceByKey((*x*,*y*)=>*x*.append(*y*));

RDD9=RDD7.map(*x*=>makeRDD(*x*));

// ④

设 RDD10={};

RDD8.foreach(*x*=>{

if(rapid_feature_set.exist(*x*.key)){

RDD8'=RDD8[indexof(*x*.key)].repartitionAndSortWithinPartitions(.);

RDD9'=RDD9[indexof(*x*.key)].repartitionAndSortWithinPartitions(.);

RDD10= RDD10.joinByID(RDD8'.map(*x*=>diffComparison(type=*x*.key, RDD9')));

}

})

// ⑤

```
RDD11=RDD10.filter(x=>binarySourceDiff(x));
RDD12=RDD8.filter(x=>exist(x,RDD11));
RDD13=RDD9.filter(x=>exist(x,RDD11));
设 RDD14={};
RDD12.foreach(x=>{
if(accurate_feature_set.exist(x.key)){
RDD12′=RDD12[indexof(x.key)].repartitionAndSortWithinPartitions(.);
RDD13′=RDD13[indexof(x.key)].repartitionAndSortWithinPartitions(.);
RDD14= RDD14.joinByID(RDD8’.map(x=>diffComparison(type=x.key, RDD13’)));
}
})
RDD15=RDD14.map(x=>vulModel(type=diff,x,漏洞标注));
RETURN RDD15;
```

6 可行性讨论与展望

本章将从降低分析技术间的门槛、应对软件分析规模提升、漏洞分析自动化这三个角度讨论软件漏洞分析三层平面架构的实现可行性及其预期效果展望。

6.1 降低分析技术间的门槛

一直以来, 不论对于工具/方法(探索平面)的融合, 还是对于知识与技术间的交互来说, 软件漏洞分析技术间的门槛是其均需应对的难题。举例来说, (1)每种语言在共性之外都有其自身的程序分析语言特性或需重点关注的机制, 如 C 语言的堆分配机制, Java 语言的序列化/反序列化方法等; 甚至于同一份源代码在经过不同编译器选项优化之后, 在二进制代码控制结构上会有较大差异; (2)每种工具都有其特定的服务提供模式, 而对工具所使用的技术进行拆分、独立或重现则可能需要较长的开发时间, 甚至在无工具源代码时无法完成相应的技术改造工作, 如通常的商用漏洞分析工具 Coverity[84]等。

因此, 需要指出的是, 降低知识与探索平面交互时分析技术间的门槛是一项劳动复杂度极高的工作。当前业界主要采用群智的方法来解决此类问题, 如 ImageNet 数据集标注[146]。尽管与群智方法通常对参与人员专业知识要求不高的应用场景不同, 但可以注意到的是每年都有大量的软件分析领域专业人员共享其所研发的分析工具, 如能对此加以有效使用则能大幅降低消弭技术门槛的劳动复杂度。本文作者设想采用“收集工具—映射工具—加工工具”这一技术思路融合工具间的技术, 以降低实现自由定制(可选取模块/接口)的难度。具体如下, (1)收集公开的软件分析工具版本、源代码及其相关的技术文档, 能抽取其接口部分及接口说明; (2)基于三层平面的交互, 设计软件漏洞分析本体及关联关系(如 4.8 节及图 10), 能部分自动化的将手机工具的接口以及其参数返回值、接口功能说明映射至所设计的本体及关联, 批量生成基于交互的实体及关联关系; (3)映射阶段基于部分自动化的方法往往不能完全适配所设计的本体及关联关系, 仍需依赖分析人员依据经验对工具及所映射的结果进行加工。在实现此技术思路的基础上, 可进一步采用 docker[147]等现有技术制作工具容器, 实现工具的自动部署与映射更新。

6.2 应对软件分析规模提升

软件分析的过程可视为“状态-动作”序列, 即根据分析状态选择(知识平面与探索平面交互下)相应的下一步分析动作。对于分析一个软件来说, 在组合不同的分析技术条件下, 已有不可阈值的状态可能性; 而开展规模化分析, 将既有可能面对一个迅速爆炸式增长的状态空间。在这种情况下, 虽然有大量的分析功能存在, 但多数现有工具在性能上无法直接支撑如此增长的分析规模。因此, 本文采用弹性分布式数据集 RDD 进行描述, 期望将软件漏洞分析技术体系直接移植到大数据处理快速计算引擎上。同时, 这也对工具(或分析技术)部署的可扩展性提出了要求, 即底层的计算支撑平台可弹性按需部署工具实例。特别需要注意的是, 工具部署的可划分性和运行过程中的瓶颈分析步骤发现对应对规模化提升至关重要, 具体叙述如下:

- 工具部署的可划分性指用同一工具容器生成多个运行实例, 且运行实例间需通过分析对象的分发并行完成尽可能多的同一类型任务。如图 1 中可通过键值划分目标软件、目标路径、目标字节等分析对象, 使得不同的运行实例可并行加速完成针对对象集合的分析。同时考虑到运行性能, 存储资源和网络带宽资源可能也许依据划分重新分配;
- 运行过程中的瓶颈分析步骤指分析任务“状态-动作”序列中等待完成时间明显更长的动作。如对应到图 10, 动作指每一条关联表所代表的调用函数, 分析时间更长的调用函数将对图中整体分析任务的完成造成影响。可建立以关联边所表示的分析函数为中心的计算资源弹性分配机制, 当检测到性能瓶颈时, 在重新划分分析对象的基础上生成新的相关函数运行实例以缓解性能瓶颈问题。

6.3 软件漏洞自动化分析与三层平面的关联

如表 9 所示, 本文将软件漏洞自动化分析从 0→5 分为完全人工分析、协助人工分析、部分自动化、有条件的自动化、高度自动化、完全自动化共 6 个阶段, 采用 0→3 谁负责分析子任务、判断分析状态、重新设计分析任务、总体自动化程度共 4 项表征指标衡量自动化程度。

表 9 软件漏洞自动化分析各阶段及相关衡量指标

Table 9 Stages and indicators of automatic analysis of software vulnerabilities

	0 完全人工分析	1 协助人工分析	2 部分自动化	3 有条件的自动化	4 高度自动化	5 完全自动化
0 谁负责分析子任务	人	人-知识 人/系统-探索	人/系统-知识 人/系统-探索	人/系统-知识 人/系统-探索	人/系统-知识 人/系统-探索	系统
1 谁负责判断当前分析状态	人	人	人	人/系统-状态	人/系统-状态	系统
2 处于分析异常状态时, 由谁重新设计分析任务	人	人	人	人	人/系统-知识、探索、状态交互	系统
3 总体上的辅助分析或自动化分析程度	完全依赖分析人员	由系统承担部分的分析子任务	在部分的分析任务下, 系统能自动实现知识与探索平面的交互	在部分的分析任务场景下, 系统能感知分析状态, 并选取交互接口	在部分的分析任务场景下, 系统能重新设计分析任务	在理想情况下, 所有工作均由系统完成

- 在完全人工分析的阶段, 所有的工作均由分析人员完成;
- 在系统协助人工分析的阶段, 具体的分析任务(探索部分)由人或基于三层平面设计的系统承担, 但知识与探索部分的交互接口调用仍由分析人员根据其经验知识以及对分析状态的判断具体决定;
- 在部分自动化阶段, 系统相比前一阶段能组合部分工具和方法, 在一定程度上自动实现知识与探索平面的交互;
- 在有条件的自动化阶段, 系统相比前一阶段新增能部分判断当前软件漏洞分析状态的功能, 其能部分根据当前状态选取知识与探索平面的交互方式;
- 在高度自动化阶段, 系统相比前一阶段能在软件漏洞分析任务执行异常时(如分析突然中止、包含无意义的结果等), 能重新设计知识与探索平面间的交互分析任务序列, 如接口函数的选取及调用顺序等;
- 在理想的完全自动化阶段, 所有的分析任务执行、状态判断以及重新设计分析任务均有系统实现。

7 总结

本文针对当前软件漏洞分析劳动复杂度极高的现状问题, 认为有效的解耦合架构是解决这一问题的有效途径, 因而提出软件分析知识、探索、状态三层平面分离架构, 通过可编程接口连接该三层平面并基于现有相关工作论述其可行性, 最后实际例举应用场景以描述平面间的交互方式。

同时, 有两方面内容值得特别注意:

(1) 知识平面的具体组织方式。鉴于软件漏洞产生机理及相关分析知识过于庞杂, 本文在第 4.8 节考虑用软件漏洞知识图谱的方式定义相关的本体框架, 基于交互接口提取并识别知识实体和关联以组织知识平面。基于知识图谱既有的基础设施和前沿技术, 可实现软件漏洞分析状态识别-状态解释-知识建议的推荐过程。

(2) 漏洞自动化分析与三层平面的关联。本文在第 6.3 节对自动化分析阶段与基于三层平面的衡量指标进行讨论。自动化可用“识别-动作”的范式来描述, 其中识别可对应知识平面, 即用软件分析知识分辨当前产生数据集合所表征的漏洞分析状态, 动作则对应探索平面, 进而可调用知识与技术的交互接口继续探查下一步分析状态。

在未来工作中, 本文作者将沿用三层平面框架, 推进实现基于软件漏洞知识图谱的推荐(识别-解释-建议)以及自动化(识别-动作)。

参考文献

[1] kbandla. Aptnotes data. https://github.com/aptnotes/data. Accessed December 6, 2017.

[2] Cyber grand challenge. http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/. Accessed December 6, 2017.

[3] Josselin Feist, Laurent Mounier, Sastien Bardin, Robin David, and Marie-Laure Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free[C]// In *SSPREW@ACSAC*, 2016.

[4] Van Thuan Pham, Boon Ng Wei, K Rubinov, and A Roychoudhury.

Hercules: Reproducing crashes in real-world application binaries[C]// In *IEEE International Conference on Software Engineering*, pages 891–901, 2015.

[5] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution[C]//*NDSS*. 2016, 16: 1-16.

[6] Muhammad Riyad Parvez. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. Master Thesis, University of Waterloo, 2016.

[7] Abhik Roychoudhury Van-Thuan Pham, Marcel Böhme. Model-based white-box fuzzing for program binaries. *ASE*, 2016.

[8] Kim S, Woo S, Lee H, et al. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery[C]//*Security and Privacy (SP), 2017 IEEE Symposium on. IEEE*. 2017.

[9] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C]//*Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012: 1-14.

[10] McKeown N, Anderson T, Balakrishnan H, et al. OpenFlow: enabling innovation in campus networks[J]. *ACM SIGCOMM Computer Communication Review*, 2008, 38(2): 69-74.

[11] Spark API guide, http://spark.apache.org/docs/latest/rdd- programming-guide.html. Accessed December 6, 2017.

[12] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation[C]//*ACM Sigplan notices*. ACM, 2007, 42(6): 89-100.

[13] Team P X. PaX address space layout randomization (ASLR)[J], http://pax. grsecurity. net/docs/aslr. txt. 2003.

[14] Date execution prevention. http://h10032.www1.hp.com/ctg/Manual/c00387685.pdf. Accessed December 6, 2017.

[15] Brumley D, Poosankam P, Song D, et al. Automatic patch-based exploit generation is possible: Techniques and implications[C]//*Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008: 143-157.

[16] Clang, http://clang.llvm.org. Accessed December 6, 2017.

[17] Shoshitaishvili Y, Wang R, Hauser C, et al. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware[C]//*NDSS*. 2015.

[18] Barrett C W, Sebastiani R, Seshia S A, et al. Satisfiability Modulo Theories[J]. Handbook of satisfiability, 2009, 185: 825-885.

[19] King J C. Symbolic execution and program testing[J]. *Communications of the ACM*, 1976, 19(7): 385-394.

[20] Boyer R S, Elspas B, Levitt K N. SELECT—a formal system for testing and debugging programs by symbolic execution[J]. *ACM SigPlan Notices*, 1975, 10(6): 234-245.

[21] Gallaire H. Logic Programming: Further Developments[C]//*SLP*. 1985, 85: 88-96.

[22] Ganesh V, Dill D L. A decision procedure for bit-vectors and arrays[C]//*CAV*. 2007, 4590: 519-531.

[23] De Moura L, Bjørner N. Z3: An efficient SMT solver[J]. Tools and Algorithms for the Construction and Analysis of Systems, 2008: 337-340.

[24] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C]//*OSDI*. 2008, 8: 209-224.

[25] Cadar C, Ganesh V, Pawlowski P M, et al. EXE: automatically generating inputs of death[J]. *ACM Transactions on Information and System Security (TISSEC)*, 2008, 12(2): 10.

[26] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems[J]. *ACM SIGPLAN Notices*, 2011, 46(3): 265-278.

[27] Anand S, Péséreanu C, Visser W. JPF－SE: A symbolic execution extension to java pathfinder[J]. Tools and Algorithms for the Construction and Analysis of Systems, 2007: 134-138.

[28] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing[C]//*ACM Sigplan Notices*. ACM, 2005, 40(6): 213-223.

[29] Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C[C]//*ACM SIGSOFT Software Engineering Notes*. ACM, 2005, 30(5): 263-272.

[30] Lattner C. LLVM and Clang: Next generation compiler technology[C]//*The BSD Conference*, 2008: 1-2.

[31] Zhang J, Wang X. A constraint solver and its application to path feasibility analysis[J]. *International Journal of Software Engineering and Knowledge Engineering*, 2001, 11(02): 139-156.

[32] Burnim J, Sen K. Heuristics for scalable dynamic test generation[C]//*IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008: 443-446.

[33] Xie T, Tillmann N, de Halleux J, et al. Fitness-guided path exploration in dynamic symbolic execution[C]//*Dependable Systems & Networks, IEEE/IFIP International Conference on. IEEE*, 2009: 359-368.

[34] Marinescu P D, Cadar C. KATCH: high-coverage testing of software patches[C]//*Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013: 235-245.

[35] Zhang C, Groce A, Alipour M A. Using test case reduction and prioritization to improve symbolic execution[C]//*Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014: 160-170.

[36] Seo H, Kim S. How we get there: A context-guided search strategy in concolic testing[C]//*Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014: 413-424.

[37] Boonstoppel P, Cadar C, Engler D. RWset: Attacking path explo-

sion in constraint-based test generation[J]. *Tools and Algorithms for the Construction and Analysis of Systems*, 2008: 351-366.

[38] Taneja K, Xie T, Tillmann N, et al. eXpress: guided path exploration for efficient regression test generation[C]//*Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011: 1-11.

[39] Yi Q, Yang Z, Guo S, et al. Postconditioned symbolic execution[C]//*Software Testing, Verification and Validation (ICST), IEEE International Conference on*. IEEE, 2015: 1-10.

[40] Staats M, Păsăreanu C. Parallel symbolic execution for structural test generation[C]//*Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010: 183-194.

[41] Bucur S, Ureche V, Zamfir C, et al. Parallel symbolic execution for automated real-world software testing[C]//*Proceedings of the sixth conference on Computer systems*. ACM, 2011: 183-198.

[42] Denning D E. A lattice model of secure information flow[J]. *Communications of the ACM*, 1976, 19(5): 236-243.

[43] Denning D E, Denning P J. Certification of programs for secure information flow[J]. *Communications of the ACM*, 1977, 20(7): 504-513.

[44] Shankar U, Talwar K, Foster J S, et al. Detecting Format String Vulnerabilities with Type Qualifiers[C]//*USENIX Security Symposium*. 2001: 201-220.

[45] Qin F, Wang C, Li Z, et al. A low-overhead practical information flow tracking system for detecting general security attacks[C]//*Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006: 135-148.

[46] King D, Hicks B, Hicks M, et al. Implicit flows: Can't live with 'em, can't live without 'em[J]. *Information Systems Security*, 2008: 56-70.

[47] Ceara D, Mounier L, Potet M L. Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences[C]//*Software Testing, Verification, and Validation Workshops (ICSTW)*, 2010 Third International Conference on. IEEE, 2010: 371-380.

[48] Jovanovic N, Kruegel C, Kirda E. Pixy: A static analysis tool for detecting web application vulnerabilities[C]//*Security and Privacy, IEEE Symposium on*. IEEE, 2006: 6 pp.258-263.

[49] Tripp O, Pistoia M, Fink S J, et al. TAJ: effective taint analysis of web applications[C]//*ACM Sigplan Notices*. ACM, 2009, 44(6): 87-97.

[50] IDA pro. https://www.hex-rays.com. Accessed December 8, 2017.

[51] Evans D, Guttag J, Horning J, et al. LCLint: A tool for using specifications to check code[J]. *ACM SIGSOFT Software Engineering Notes*, 1994, 19(5): 87-96.

[52] Xie Y, Aiken A. Saturn: A SAT-Based Tool for Bug Detection[C]//CAV. 2005, 3576: 139-143.

[53] Hsieh C S. A fine-grained data-flow analysis framework[J]. *Acta Informatica*, 1997, 34(9): 653-665.

[54] Drewry W, Ormandy T. Flayer: Exposing Application Internals[J]. *WOOT*, 2007, 7: 1-9.

[55] Nethercote N. Dynamic binary analysis and instrumentation[R]. University of Cambridge, Computer Laboratory, 2004.

[56] Nethercote N, Seward J. How to shadow every byte of memory used by a program[C]//*Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007: 65-74.

[57] Pin tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. Accessed December 8, 2017.

[58] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[C]//*NDSS*. 2005.

[59] Argos. http://www.few.vu.nl/argos. Accessed December 8, 2017.

[60] Enck W, Gilbert P, Han S, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones[J]. *ACM Transactions on Computer Systems (TOCS)*, 2014, 32(2): 5.

[61] 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]//*Security and privacy (SP), IEEE symposium on*. IEEE, 2010: 317-331.

[62] Kang M G, McCamant S, Poosankam P, et al. Dta++: dynamic taint analysis with targeted control-flow propagation[C]//*NDSS*. 2011.

[63] Livshits V B, Lam M S. Finding Security Vulnerabilities in Java Applications with Static Analysis[C]//*USENIX Security Symposium*. 2005, 14: 8-18.

[64] Tripp O, Pistoia M, Cousot P, et al. Andromeda: Accurate and Scalable Security Analysis of Web Applications[C]//*FASE*. 2013, 7793: 210-225.

[65] Arzt S, Rasthofer S, Fritz C, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps[J]. *Acm Sigplan Notices*, 2014, 49(6): 259-269.

[66] Qin F, Wang C, Li Z, et al. Lift: A low-overhead practical information flow tracking system for detecting security attacks[C]//P*roceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006: 135-148.

[67] Yoon M K, Salajegheh N, Chen Y, et al. Pift: Predictive information-flow tracking[C]//*Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016: 713-725.

[68] Chen H, Wu X, Yuan L, et al. From speculation to security: Practical and efficient information flow tracking using speculative hardware[C]//*ISCA*, 2008: 401-412.

[69] Clause J, Li W, Orso A. Dytan: a generic dynamic taint analysis framework[C]//*Proceedings of the 2007 international symposium*

on Software testing and analysis. ACM, 2007: 196-206.

[70] Vogt P, Nentwich F, Jovanovic N, et al. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis[C]//*NDSS*. 2007, 2007: 12.

[71] Bao T, Zheng Y, Lin Z, et al. Strict control dependence and its effect on dynamic information flow analyses[C]//*Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010: 13-24.

[72] Cox L P, Gilbert P, Lawler G, et al. SpanDex: Secure Password Tracking for Android[C]//*USENIX Security Symposium*, 2014, 2014.

[73] Holzmann G J. The model checker SPIN[J]. *IEEE Transactions on software engineering*, 1997, 23(5): 279-295.

[74] Huth M, Ryan M. Logic in Computer Science: Modelling and reasoning about systems[M]. Cambridge university press, 2004.

[75] Clarke E M, Grumberg O, Peled D. Model checking[M]. MIT press, 1999.

[76] Bérard B, Bidoit M, Finkel A, et al. Systems and software verification: model-checking techniques and tools[M]. Springer Science & Business Media, 2013.

[77] Beyer D, Henzinger T A, Jhala R, et al. The software model checker Blast[J]. *International Journal on Software Tools for Technology Transfer*, 2007, 9(5-6): 505-525.

[78] Chaki S, Clarke E M, Groce A, et al. Modular verification of software components in C[J]. *IEEE Transactions on Software Engineering*, 2004, 30(6): 388-402.

[79] Burch J, Clarke E M, Long D. Symbolic model checking with partitioned transition relations[J]. *Computer Science Department*, 1991: 435.

[80] Chen H, Wagner D. MOPS: an infrastructure for examining security properties of software[C]//*Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002: 235-244.

[81] Floyd R W. Assigning meanings to programs[M]//*Program Verification*. Springer, Dordrecht, 1993: 65-81.

[82] Schaefer T J. The complexity of satisfiability problems[C]// *Proceedings of the tenth annual ACM symposium on Theory of computing*. ACM, 1978: 216-226.

[83] Xie Y, Chou A, Engler D. Archer: using symbolic, path-sensitive analysis to detect memory access errors[J]. *ACM SIGSOFT Software Engineering Notes*, 2003, 28(5): 327-336.

[84] Coverity. https://www.coverity.com. Accessed December 8, 2017.

[85] ESC/Java2. http://www.kindsoftware.com/products/opensource/ESCJava2. Accessed December 8, 2017.

[86] Detlefs D, Nelson G, Saxe J B. Simplify: a theorem prover for program checking[J]. *Journal of the ACM (JACM)*, 2005, 52(3): 365-473.

[87] TIS. http://refspecs.linuxbase.org/elf/elf.pdf. Accessed December 8, 2017.

[88] Dullien T, Porst S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis[C]//*CanSecWest*. 2009.

[89] Lin Z, Zhang X, Xu D. Automatic reverse engineering of data structures from binary execution[C]//*Proceedings of the 11th Annual Information Security Symposium*. CERIAS-Purdue University, 2010: 5.

[90] Yu F, Alkhalaf M, Bultan T. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses[C]//*Proceedings of the 2009 IEEE/ACM International Conference on automated software engineering*. IEEE Computer Society, 2009: 605-609.

[91] Zhang M, Sekar R. Control Flow Integrity for COTS Binaries[C]//*USENIX Security Symposium*. 2013: 337-352.

[92] Rawat S, Mounier L. Finding buffer overflow inducing loops in binary executables[C]//*Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012: 177-186.

[93] Slowinska A, Stancescu T, Bos H. Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation[C]//*USENIX Annual Technical Conference*. 2012: 125-137.

[94] BinNavi. https://www.zynamics.com/binnavi.html. Accessed December 8, 2017.

[95] Brumley D, Jager I, Avgerinos T, et al. BAP: A binary analysis platform[C]//*International Conference on Computer Aided Verification*. Springer, Berlin, Heidelberg, 2011: 463-469.

[96] Sutton M, Greene A, Amini P. Fuzzing: brute force vulnerability discovery[M]. Pearson Education, 2007.

[97] Takanen A, Demott J D, Miller C. Fuzzing for software security testing and quality assurance[M]. Artech House, 2008.

[98] Duran J W, Ntafos S C. An evaluation of random testing[J]. *IEEE transactions on Software Engineering*, 1984 (4): 438-444.

[99] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. *Communications of the ACM*, 1990, 33(12): 32-44.

[100] Cohen M B, Snyder J, Rothermel G. Testing across configurations: implications for combinatorial testing[J]. *ACM SIGSOFT Software Engineering Notes*, 2006, 31(6): 1-9.

[101] Jalbert N, Sen K. A trace simplification technique for effective debugging of concurrent programs[C]//*Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010: 57-66.

[102] Aitel D. An Introduction to SPIKE, the Fuzzer Creation Kit. www: blackhat: com/presentations/bh-usa-02/bh-us-02-aitel-spike: ppt, 2002[J]. Black Hat US.

[103] Peach. http://peachfuzzer.com. Accessed December 8, 2017.

[104] HotFuzz. http://hotfuzz.sourceforge.net. Accessed December 8, 2017.

[105] Amini P, Portnoy A. Sulley: Pure python fully automated and unattended fuzzing framework, https://github.com/OpenRCE/sulley. Accessed December 8, 2017.

[106] FileFuzz. http://labs.idefense.com/software/fuzzing. Accessed December 8, 2017.

[107] Cohen M B, Snyder J, Rothermel G. Testing across configurations: implications for combinatorial testing[J]. *ACM SIGSOFT Software Engineering Notes*, 2006, 31(6): 1-9.

[108] Wang T, Wei T, Lin Z, et al. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution[C]//*NDSS*. 2009.

[109] Wang T, Wei T, Gu G, et al. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection[C]//*Security and privacy (SP), IEEE symposium on*. IEEE, 2010: 497-512.

[110] Godefroid P, Levin M Y, Molnar D. SAGE: whitebox fuzzing for security testing[J]. *Queue*, 2012, 10(1): 20.

[111] Molnar D A, Wagner D. Catchconv: Symbolic execution and run-time type inference for integer conversion errors[J]. UC Berkeley EECS, 2007.

[112] Lanzi A, Martignoni L, Monga M, et al. A smart fuzzer for x86 executables[C]//*Software Engineering for Secure Systems, 2007. SESS'07: ICSE Workshops 2007*. Third International Workshop on. IEEE, 2007: 715.

[113] Song D, Brumley D, Yin H, et al. BitBlaze: A new approach to computer security via binary analysis[J]. *Information systems security*, 2008: 1-25.

[114] Sheyner O, Haines J, Jha S, et al. Automated generation and analysis of attack graphs[C]//*Security and privacy, IEEE Symposium on*. IEEE, 2002: 273-284.

[115] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing[C]//*Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009: 474-484.

[116] BuzzFuzz. https://ece.uwaterloo.ca/~vganesh/buzzfuzz.html. Accessed December 8, 2017.

[117] Flake H. Structural Comparison of Executable Objects[C]// *IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. 2004: 161-173.

[118] Sabin T. Comparing binaries with graph isomorphisms[J]. Bindview. http://www.bindview.com/Support/RAZOR/Papers, 2004.

[119] D ULLIEN T, OLLES R. R. Graph-based comparison of Executable Objects[J]. 2011-08-10]. http://actes.sstic.org/SSTICOS/ Analyse_differentielle de binaires/SSTIC05-article-Flake-Graph_ based_comparison of Ex-ecutable_Objects, pdf.

[120] IDACompare. https://github.com/dzzie/IDACompare. Accessed December 8, 2017.

[121] eEye binary diffing suite. http://www.woodmann.com/collaborative/tools/index.php/EEye_Binary_Diffing_Suite_%28EBDS%29. Accessed December 8, 2017.

[122] Bauer V, Heinemann L, Deissenboeck F. A structured approach to assess third-party library usage[C]//*Software Maintenance (ICSM), 2012 28th IEEE International Conference on. IEEE*, 2012: 483-492.

[123] Cadariu M D. Tracking known security vulnerabilities in third-party components[J]. Master Thesis, TUDelft. 2014.

[124] Alqahtani S S, Eghan E E, Rilling J. Tracing known security vulnerabilities in software repositories–A Semantic Web enabled modeling approach[J]. *Science of Computer Programming*, 2016, 121: 153-175.

[125] Pang Y, Xue X, Namin A S. Early Identification of Vulnerable Software Components via Ensemble Learning[C]//*Machine Learning and Applications (ICMLA)*, 2016 15th *IEEE International Conference on*. IEEE, 2016: 476-481.

[126] Xuefeng Lv, Deng Li, Qing Yin. Firmware vulnerability detection in embeded device based on homology anlaysis[J]. *Journal of Computer Engineering*, 2017.

[127] Chuanda Ding. State of windows application security: shared libraries. *CanSecWest*, 2017.

[128] Livshits V B, Lam M S. Tracking pointers with path and context sensitivity for bug detection in C programs[J]. *ACM SIGSOFT Software Engineering Notes*, 2003, 28(5): 317-326.

[129] Wagner D, Foster J S, Brewer E A, et al. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities[C]//*NDSS*. 2000: 2000-02.

[130] Hovemeyer D, Pugh W. Finding bugs is easy[J]. *ACM Sigplan Notices*, 2004, 39(12): 92-106.

[131] Bravenboer M, Smaragdakis Y. Strictly declarative specification of sophisticated points-to analyses[J]. *ACM SIGPLAN Notices*, 2009, 44(10): 243-262.

[132] Hallem S, Chelf B, Xie Y, et al. A system and language for building system-specific, static analyses[C]//*ACM SIGPLAN Notices*. ACM, 2002, 37(5): 69-82.

[133] Callahan D. The program summary graph and flow-sensitive interprocedual data flow analysis[M]. ACM, 1988.

[134] Whaley J, Lam M S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams[C]//*ACM SIGPLAN Notices*. ACM, 2004, 39(6): 131-144.

[135] Fortify. http://fortify.software.informer.com. Accessed December 9, 2017.

[136] FindBugs. https://sourceforge.net/p/findbugs. Accessed December 9, 2017.

[137] Appscan. https://www.ibm.com/software/products/en/appscan-source.

Accessed December 9, 2017.

[138] Avgerinos, S.K.Cha, B.L.T.Tao and D.Brumley. AEG: Automatic Exploit Generation. *In Proceedings of the Network and Distributed System Security Symposium* (NDSS), 2011.

[139] Cha S K, Avgerinos T, Rebert A, et al. Unleashing mayhem on binary code[C]//*Security and Privacy (SP), IEEE Symposium on. IEEE*, 2012: 380-394.

[140] Wang M, Su P, Li Q, et al. Automatic polymorphic exploit generation for software vulnerabilities[C]//*International Conference on Security and Privacy in Communication Systems*. Springer, Cham, 2013: 216-233.

[141] Schwartz E J, Avgerinos T, Brumley D. Q: Exploit Hardening Made Easy[C]//*USENIX Security Symposium*. 2011: 25-41.

[142] Hu H, Chua Z L, Adrian S, et al. Automatic Generation of Data-Oriented Exploits[C]//*USENIX Security Symposium*. 2015: 177-192.

[143] Hu H, Shinde S, Adrian S, et al. Data-oriented programming: On the expressiveness of non-control data attacks[C]//*Security and Privacy (SP), IEEE Symposium on*. IEEE, 2016: 969-986.

[144] MS17-010. https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2017 /ms17-010. Accessed December 9, 2017.

[145] Ssdeep. http://ssdeep.sourceforge.net/. Accessed December 9, 2017.

[146] ImageNet. http://www.image-net.org/. Accessed March 12, 2018.

[147] The Docker Platform. http://www.docker.com/. Accessed March 12. 2018.

袁子牧 于 2015 年在中国科学院大学计算机系统结构专业获得博士学位。现任中国科学院信息工程研究所副研究员。研究领域为软件数据分析、领域知识图谱构建等。Email: yuanzimu@iie.ac.cn

肖扬 于 2015 年在中山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攻读博士学位。研究领域为网络与软件安全。研究兴趣包括: 基于软件复用的漏洞挖掘、知识图谱构建等。Email: xiaoyang@iie.ac.cn

吴炜 于 2014 年在中国科学技术大学信息安全专业获得学士学位。现在中国科学院信息工程研究所攻读博士学位。研究领域为网络与软件安全。研究兴趣包括: 漏洞挖掘与利用、程序分析及网络对抗。Email: wuwei@iie.ac.cn

霍玮 于 2010 年在中国科学院计算技术研究所获得博士学位, 现任中国科学院信息工程研究所研究员。主要研究方向为软件漏洞挖掘和安全评测、基于大数据的软件安全分析、智能终端系统及应用安全分析等.。Email: huowei@iie.ac.cn

邹维 研究员、博士生导师。现任中国科学院信息工程研究所副所长。中国科学院“百人计划”人选, 中国计算机学会优秀博士论文指导教师。研究领域包括网络与软件安全、攻防对抗理论与技术等。Email: zouwei@iie.ac.cn