Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

二进制代码补丁存在性测试

硕士研究生 徐程柯

2025年03月02日

问题回溯

- 总结反思
  - 讲解语速较快，时间较短，语气词较多
  - 内容详略不当，算法部分讲解深度不足、浅尝辄止
  - 互动部分较少
- 相关内容
  - **2024.10.08** 高玺凯 《二进制代码相似性检测技术》
  - **2022.11.27** 沈宇辉 《二进制函数相似性分析》

内容提要

- 预期收获
- 内容引入
- 内涵解析与研究目标
- 研究背景与研究意义
- 研究历史
- 知识基础
- 算法原理
  - **PatchDiscovery**
- 特点总结与工作展望
- 参考文献

预期收获

- 预期收获
  - 掌握二进制代码、补丁存在性测试的基本概念
  - 了解二进制代码补丁存在性测试发展历程
  - 理解二进制代码补丁存在性测试的基本原理
  - 明确二进制代码补丁存在性测试的应用和前沿发展

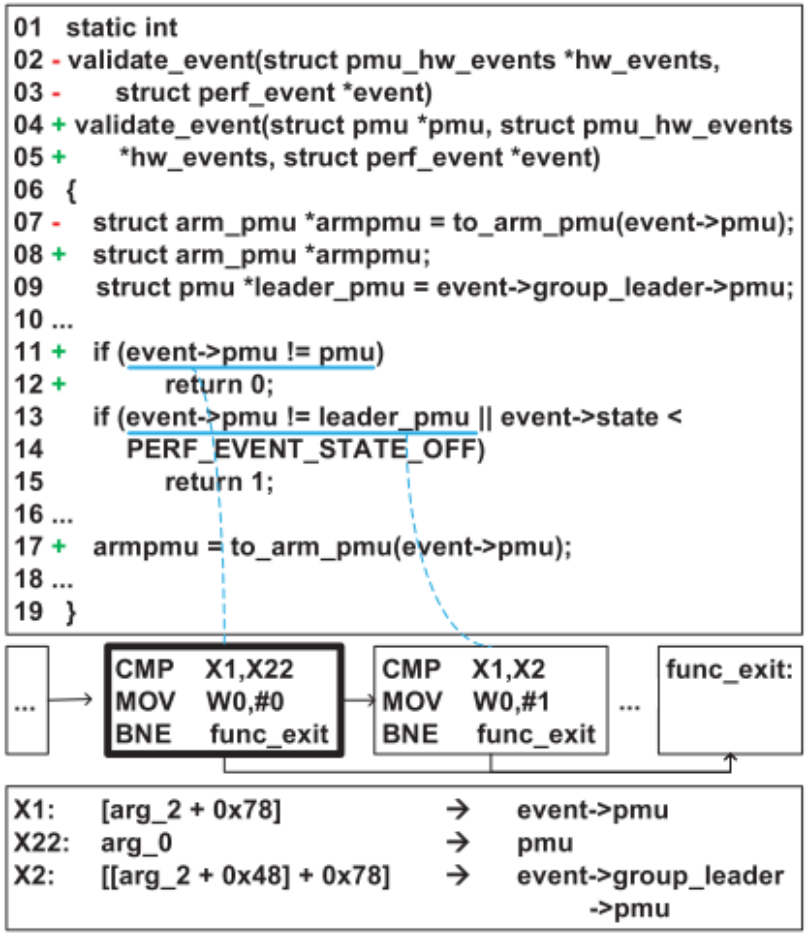
内容引入

- 下游厂商的补丁延迟与报告不可靠性
  - 对9个主流厂商的715个基于Linux的内核镜像统计研究结果
  - 聚焦于152个Linux内核漏洞的补丁状态，验证厂商发布的补丁报告的真实性
  - 补丁遗漏：在所有厂商的镜像中，平均 **4.37%** 的补丁未被及时更新，其中魅族、**Vivo** 和 **D-Link** 的遗漏率超过 **13%**
  - 补丁报告的缺失与误导

Vendor	# of Images	# of [Image, Vul] Pairs[1]	# of Omitted Patches	Max Patch Age (day)	# of Wrong Patch Reports[2]
Google	152	4,690	0/0%	0	2/0.04%
Samsung	120	3,414	133/3.89%	643	0/0%
Xiaomi	52	1,585	57/3.60%	1,018	0/0%
Vivo	22	652	94/14.42%	893	4/0.61%
Huawei	186	3,911	9/0.23%	373	3/0.08%
Meizu	102	2,563	349/13.62%	1,085	235/9.17%
Oppo	29	852	25/2.93%	935	15/1.76%
D-Link	25	422	97/22.99%	1,451	N/A
NETGEAR	27	496	48/9.68%	1,322	N/A
Total	715	18,585	812/4.37%	-	259/1.39%

内容引入 补丁

- 基本概念
  - 补丁（**Patch**）是一种对软件、操作系统或二进制程序进行修改的更新文件或代码片段
  - 安全补丁（**Security Patch**）
    - 修复缓冲区溢出漏洞
    - 关闭未授权访问的后门
  - **Bug** 修复补丁（**Bug Fix Patch**）
    - 解决软件崩溃问题
    - 修复内存泄漏
  - 功能增强补丁（**Feature Patch**）
    - 提高软件的运行效率
    - 增加新接口或**API**
    - 增强兼容性



内容引入 上下游内核

- 上游内核
  - 上游内核通常指的是**Linux** 内核的官方主线版本（**Mainline Kernel**），由**Linux** 内核社区（**Linux Kernel Community**）在 **kernel.org** 上维护
  - 与硬件无关的通用版本——没有针对特定设备的修改，适用于所有架构（如 **x86, ARM, RISC-V**）
- 下游内核
  - 各个厂商或组织基于上游内核进行定制和维护的版本，通常用于特定硬件、操作系统或业务需求，例如 **Android**
  - 基于上游内核，可能包含额外的驱动、功能或安全补丁
  - 版本更新通常比上游慢，需要进行额外的测试、优化和兼容性调整

特性	上游内核(Upstream Kernel)	下游内核(Downstream Kernel)
维护者	Linux内核社区	Linux发行版、设备厂商(Google等)
更新速度	持续更新，每6-8周一个新版本	更新较慢，可能基于老版本长期维护
适用范围	通用，适用于所有硬件	适用于特定的设备或发行版
是否定制	无厂商定制	深度定制，可能有私有补丁
安全补丁	及时应用最新安全补丁	可能滞后，厂商自行维护

内容引入 应用场景

- 源码对源码（**Source-to-Source**）
  - 依赖源代码：这类方法仅在源代码级别进行比较，不涉及二进制分析
  - 对比源代码的**AST**（抽象语法树）、代码语义或代码结构，分析补丁是否被应用
- 二进制对二进制（**Binary-to-Binary**）
  - 无需源代码：这类方法仅依赖二进制文件，不涉及源代码
  - 函数级比对以及指令级比对分析代码逻辑的变化
- 源码对二进制（**Source-to-Binary**）
  - 参考（**Reference**）是源码，目标（**Target**）是二进制
  - 结合开源软件的补丁历史和闭源软件的二进制分析，提供更高效的补丁存在性检测能力
  - 源代码中的变量名、函数名等信息有助于定位二进制中的补丁代码

内涵解析与研究目标

- 内涵解析
  - 二进制代码补丁存在性测试（**Binary Patch Presence Testing, BPPT**）旨在验证特定二进制文件是否已应用指定的补丁
  - 对编译后的二进制文件进行分析，主要用于安全性评估、软件合规性检查以及漏洞管理
- 研究目标
  - 支持二进制代码的下游任务，如补丁状态验证、供应链安全等，利用自动化分析方法挖掘补丁状态，提高漏洞管理的效率
  - 优化二进制代码的补丁匹配机制，结合控制流分析、指令序列匹配等方法，提高跨编译器、跨优化级别的补丁识别能力

研究背景与意义

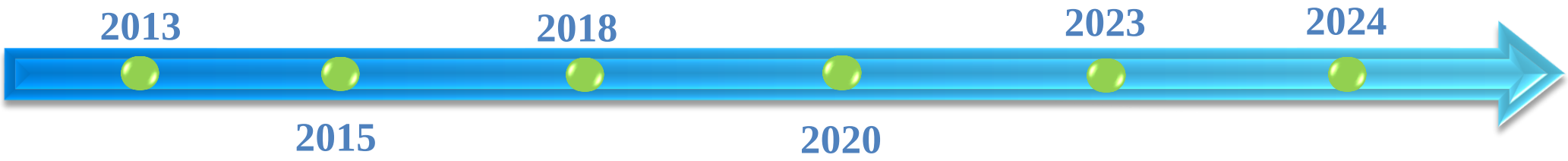
- 研究背景
  - 二进制补丁存在性测试的广泛应用：操作系统安全、供应链安全管理、企业级软件与服务器安全
  - 在复杂的软件生态系统和供应链中，许多厂商未能及时应用关键补丁
  - 不同编译器（**GCC**、**Clang**）、不同架构（**x86**、**ARM**）、不同优化级别（**O0~O3**）会导致二进制代码在指令级别发生变化
- 研究意义
  - 降低已知漏洞的安全风险
  - 优化补丁验证技术，减少误报和漏报
  - 增强供应链安全管理

研究历史 二进制代码补丁存在性测试

Andy等人提出**BinSlayer**方法**CFG**（**Control Flow Graph**）和**CG**（**Call Graph**）两种不同的程序结构表示二进制代码

Zhang等人提出了**FIBER**方法，构建局部**CFG**，通过构建粗细力度检测过程，表征细粒度的补丁变化

Xu等人提出了**PatchDiscovery**方法，为补丁前后的代码以及目标二进制代码均生成各自的特征签名，关注重要的基本块



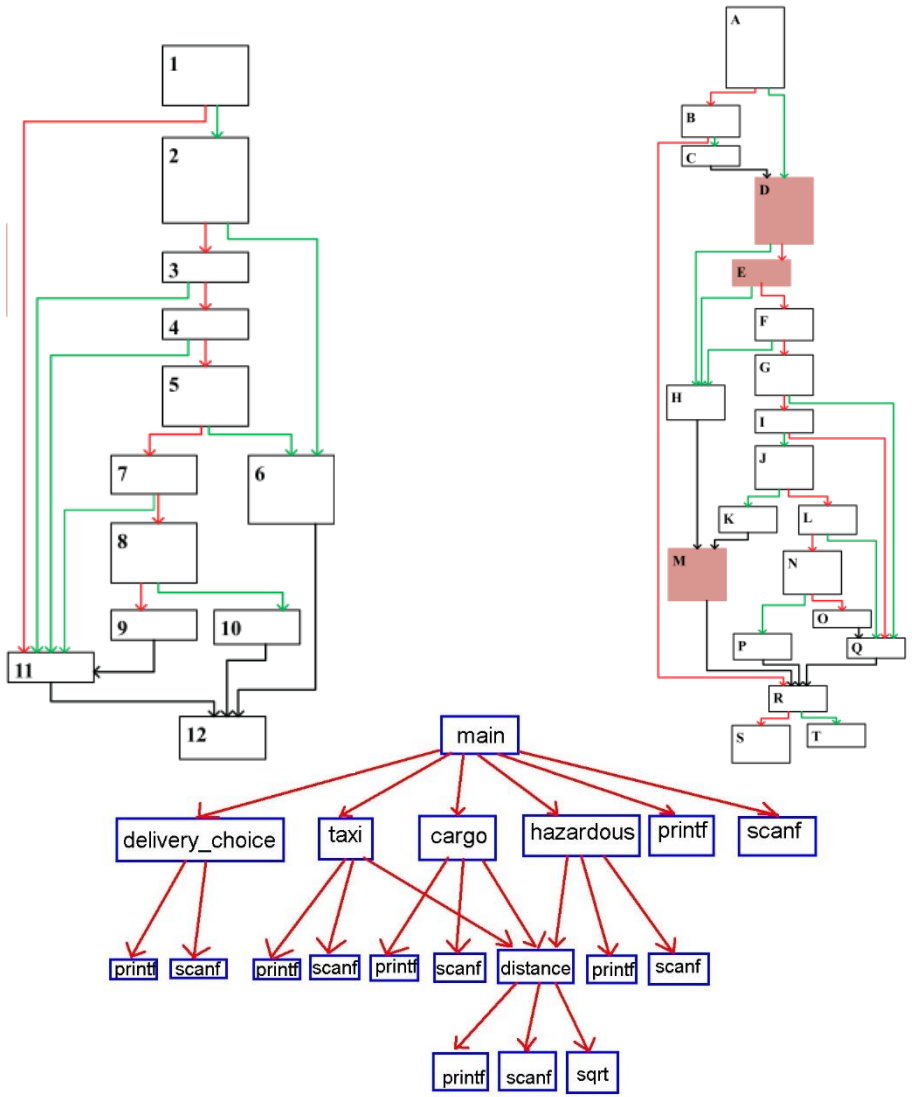
Jannik 等人提出利用**IR**（**intermediate representation**）的中间变量，将各个体系中的二进制表示为**IR**

Jiang等人首次提出语义级别的补丁存在性测试方法，关注代码语义信息，生成补丁语义摘要信息，提升模型鲁棒性

Zhan等人提出**REACT**，利用**IR**构建源代码和二进制代码的中间表示，并分析向前向后数据流提取二进制代码特征，进行功能匹配

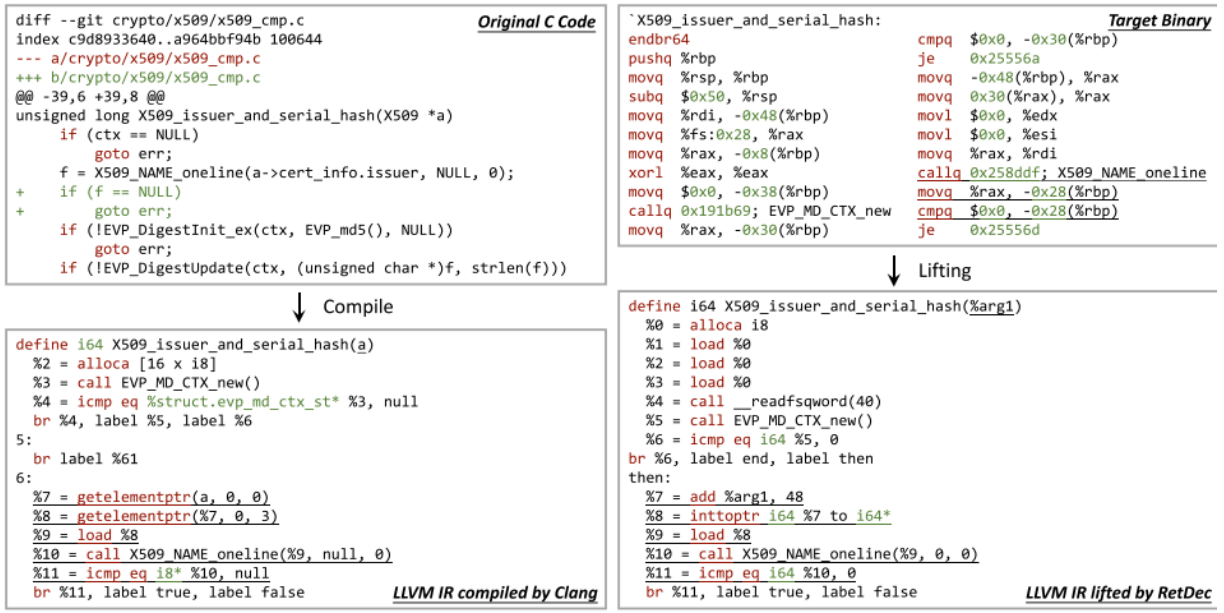
知识基础 控制流图与调用图

- 控制流图（CFG, Control Flow Graph）
  - 一种有向图（Directed Graph），用于表示程序的控制流结构
  - 节点（Node）：表示程序的基本块（Basic Block），即一组顺序执行的指令
  - 边（Edge）：表示程序的控制流跳转，即基本块之间的执行顺序，可以是顺序执行、条件跳转或无条件跳转
- 调用图（CG, Call Graph）
  - 一种有向图，表示函数之间的调用关系
  - 节点（Node）：表示一个函数（Function）
  - 边（Edge）：表示一个函数调用了另一个函数



知识基础 中间表示

- **IR（Intermediate Representation，中间表示）是一种介于源代码和二进制代码之间的抽象表示，用于优化、分析和转换代码**
  - 二进制补丁检测：将目标二进制提升到 **IR** 级别，方便与补丁特征匹配
  - 程序分析和漏洞检测：在 **IR** 层面进行控制流和数据流分析，查找漏洞
  - 二进制翻译：将 **x86/ARM/MIPS** 等指令集转换为通用的 **LLVM IR**，以便跨平台执行



知识基础 符号执行

- 符号执行（**Symbolic Execution**） 用符号变量（**symbolic variables**） 代替输入，并跟踪程序的执行路径和条件约束
  - 传统执行：使用具体输入值来运行程序，例如 **x = 5, y = 10**
  - 符号执行：使用符号变量（如 **x = α, y = β**），并在执行过程中推导出表达式
- 优势
  - 自动分析程序所有路径
  - 更全面的代码覆盖
  - 可以用于无源代码的二进制分析
- 劣势
  - 路径爆炸（**Path Explosion**）
  - 符号求解困难

优点	解释
自动化漏洞检测	可用于分析未知二进制代码，检查补丁是否修复了漏洞
完整路径覆盖	探索所有可能的执行路径，提高测试覆盖率
自动生成攻击输入	结合 SMT 求解器，寻找可触发漏洞的具体输入
适用于二进制分析	可用于分析未知二进制代码，检查补丁是否修复了漏洞

知识基础 编码方式

- 编译器是一种将高级语言代码（如 C、C++、Rust）转换为可执行的二进制代码的软件工具
  - 语法分析（Lexical & Syntax Analysis）：解析源代码，检查语法是否正确
  - 代码优化（Optimization）：优化代码，使其执行更快或占用更少资源
  - 代码生成（Code Generation）：将优化后的代码转换为目标 CPU 指令
- 编译优化级别控制了编译器如何优化代码，能够调整代码执行效率、二进制大小和运行时性能
  - 不同的编译优化级别会导致二进制结构变化

编译选项	描述	影响
-O0	无优化，直接翻译C代码	代码可读性强，但性能低
-O1	基本优化，移除几余代码	代码体积略减小，性能提升
-O2	较高优化，常用于发布版本	更好的指令选择，减少不必要的计算
-O3	最高优化，启用额外优化(如循环展开)	最大化性能，但可能导致更长的编译时问
-Os	优化二进制大小，适用于嵌入式系统	牺牲部分性能以减少程序体积

PatchDiscovery: Patch Presence Test for Identifying Binary Vulnerabilities Based on Key Basic Blocks

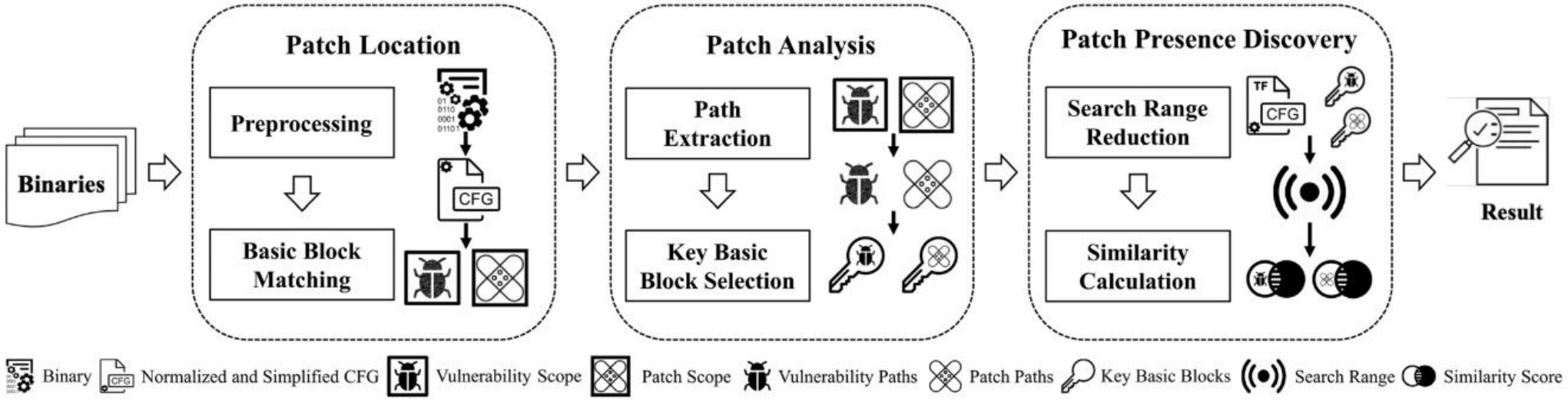
PatchDiscovery TIPO

T	目标	检测目标二进制代码是否已修补漏洞
I	输入	目标二进制代码（**the function that needs to be tested，TF**） 补丁前二进制代码（**function that contains a vulnerability，VF**） 补丁后二进制代码（**function with the vulnerability being patched，PF**）
P	处理	**1.**补丁位置：提取输入的二进制代码的**CFG**，并进行指令归一化 **2.**补丁分析：提取**PF**和**VF**的漏洞和补丁路径，并选取重要基本块 **3.**补丁存在判断：在**TF**中搜索**PF**和**VF**的重要基本块
O	输出	目标二进制代码是否存在补丁

P	问题	难以快速找到漏洞及其补丁
C	条件	需要目标二进制代码对应的补丁前后代码
D	难点	如何捕获最重要和最具代表性的二进制代码基本块
L	水平	**TRANSACTIONS ON SOFTWARE ENGINEERING 2023（SCI一区）**

PatchDiscovery 算法原理

- 算法原理图
  - 补丁位置：提取输入的二进制代码的CFG，并进行指令归一化
  - 补丁分析：提取PF和VF的漏洞和补丁路径，并选取重要基本块
  - 补丁存在判断：在TF中搜索PF和VF的重要基本块



PatchDiscovery 创新分析

- 现有方法存在问题
  - 由于整个函数级别的匹配方法无法精确区分漏洞和补丁，仅依赖结构相似度导致错误检测
  - 仅仅检测某些变更的基本块，可能无法正确识别漏洞是否被修复，因为代码升级（非补丁改动）也会引入额外变更
- **PatchDiscovery的解决方法**
  - 使用关键基本块匹配
  - 按变更指令数量排序，筛选出补丁关键基本块

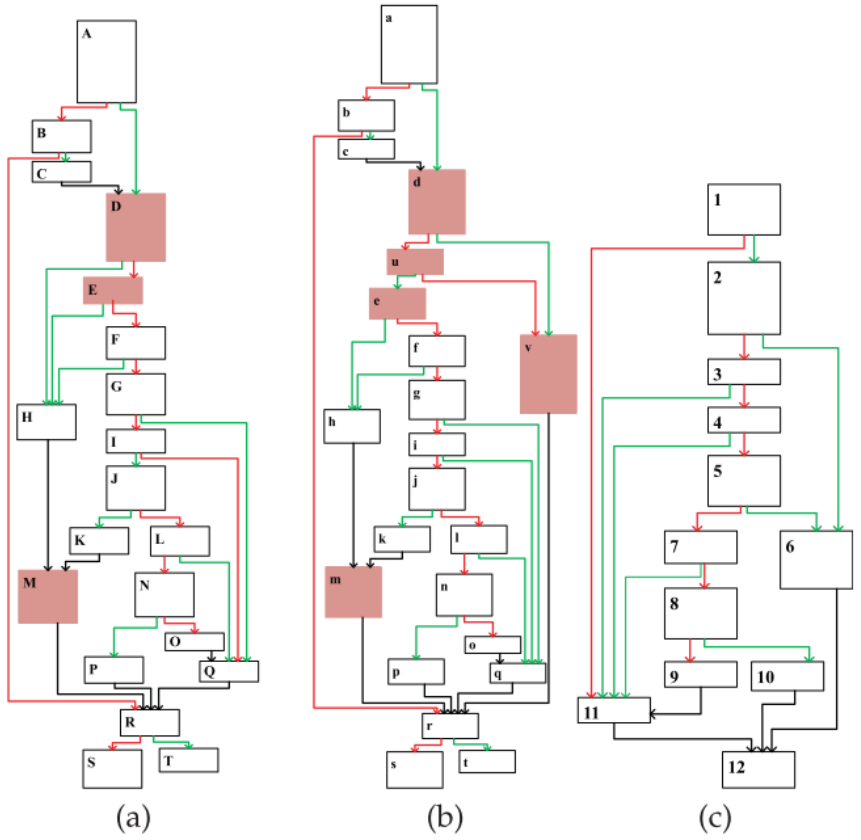


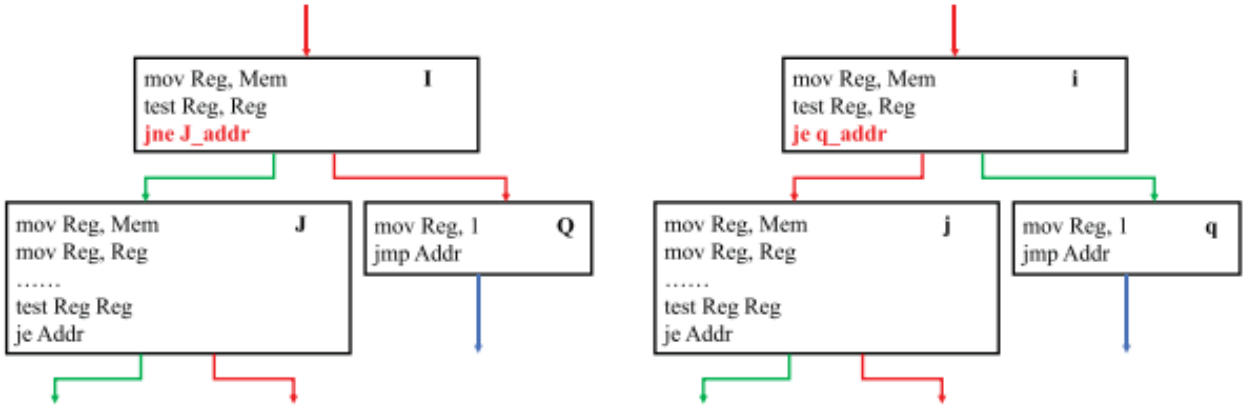
Fig. 1. Motivating example: CFGs of the function *load_specific_debug_section* in Binutils versions 2.31 (a), 2.32 (b), and 2.29 (c).

PatchDiscovery 创新分析

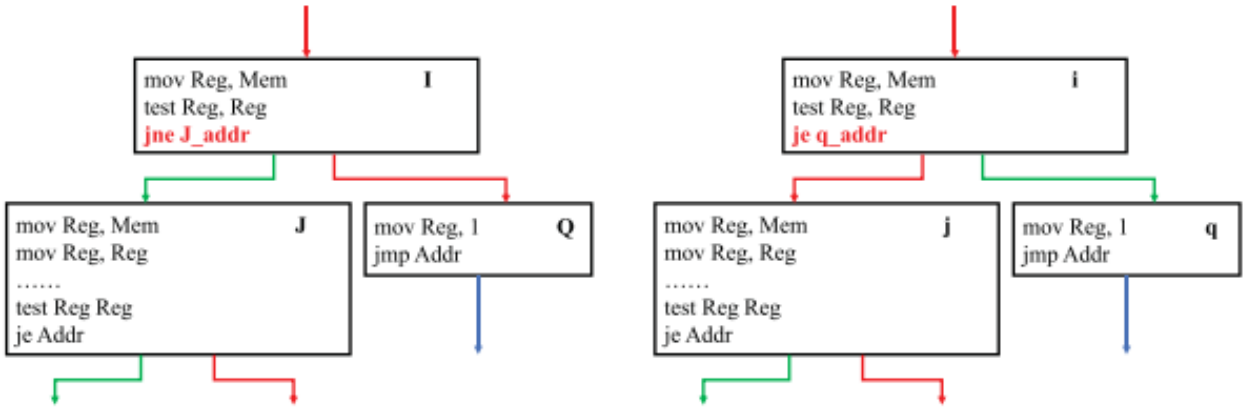
- 补丁位置
  - **CFG构建：获得二进制代码的静态信息**
    - **利用IDA Pro获取二进制代码的CFG**
  - 指令归一化和简化：维持指令的语义并提高了汇编效果的鲁棒性
    - 寄存器（**Register**） → 统一替换为 "**Reg**"
    - 内存地址（**Memory**） → 统一替换为 "**Mem**"
    - 地址（**Address**） → 统一替换为 "**Addr**"
    - 字符串引用（**String Reference**） → 直接保留具体字符串
    - 库函数地址（**Library Function Address**） → 替换为库函数名（如 "**call strlen**"）
    - 用户定义函数地址（**User-defined Function Address**） → 归一化为 "**UDFunc**"
    - 删除 **nop** 指令

PatchDiscovery 创新分析

- 补丁位置
  - 基本块匹配：为了提供足够细粒的分析，将分析范围从整个函数级别缩小到补丁级别，并将基本块作为分析单元
    - 基本块的两两匹配导致计算量剧增
    - 由于编译器优化或不同编译配置，匹配的基本块可能存在语法差异
  - 特征基本块标记：函数入口；函数出口；字符串相关的基本块；函数调用块
  - 匹配搜索：从已经匹配基本块的邻居节点中找到新匹配的基本块
    - 向前向后搜索
    - 合并候选集
    - 删除已匹配的基本块



- 补丁位置
  - 向前向后搜索
    - b_B：当前在补丁函数（PF）中优先级最高的基本块合并候选集
    - B_P（候选前驱集合）：对 b_B 在 PF 中的每个前驱基本块 b_A，如果 b_A 已经在 VF 中找到了匹配的基本块 b_a，那么 b_a 的尚未匹配的后继基本块加入 B_P（作为可能的匹配对象）
    - B_S（候选后继集合）：对 b_B 在 PF 中的每个后继基本块 b_C，如果 b_C 已经在 VF 中找到了匹配的基本块 b_c，那么 b_c 的尚未匹配的前驱基本块加入 B_S（作为可能的匹配对象）



PatchDiscovery 创新分析

- 补丁分析
  - 路径提取：孤立的基本块缺乏控制流等信息，需要分析整体路径信息
    - 补丁函数（PF）中提取关键的补丁路径
    - 漏洞函数（VF）中提取关键的漏洞路径
  - B_P 补丁基本块集合（Patch Basic Blocks）
  - BB_{PV}匹配的基本块集合（Matched Basic Blocks）
  - P_P补丁路径集合（Patch Paths）
  - P_V漏洞路径集合（Vulnerability Paths）

Algorithm 1: Patch Path Extraction

Input:
B_P //the patch basic block set of PF
BB_{PV} //the matched basic block set between PF and VF
Output: P_P //the set of all patch paths

```
P_P ← ∅
for each b ∈ B_P do
    if b.Pres = ∅ or ∃b_{i∈b.Pres} ∈ BB_PV then
        p_now ← ∅
        EXTRACT(b, p_now, P_P)
return P_P
Function EXTRACT(b, p_now, P_P):
    if b ∈ B_P then
        p_now += b
    if b ∉ B_P or b.Sucs = ∅or ∃loop then
        P_p∪ = p_now
    else
        for b_{j∈b.Sucs} ∈ B_P do
            p_new =Copy(p_now)
            EXTRACT(b_j, p_new, P_P)
```

漏洞函数（VF）：

A → B → C → D（漏洞在 C）

补丁函数（PF）：

A → B → X → C' → D（补丁修改了 B 之后的路径）

PatchDiscovery 创新分析

- 补丁分析
  - 构建基本块提取：更改指令的数量以反映变化的程度
    - 对于P_p中的每个p_p，在P_V中找到对应的p_V，比较基本块的指令变化情况
  - 粗力度匹配（**Rough Matching**）
    - 消除无关路径，减少搜索空间
    - 利用邻居匹配信息（**matched neighbor blocks**），确定可能匹配的路径范围
    - 基于邻居匹配的信息，确定候选漏洞路径 p_V
  - 精确匹配（**Precise Matching**）
    - 采用 **LSH Forest**（局部敏感哈希） 进行相似度搜索
    - 在候选路径中，找到最相似的 p_V
  - 变化程度计算
    - 采用最长公共子序列(**Longest Common Subsequence , LCS**)比较两个路径的指令变化程度

漏洞函数（VF）：

A → B → C → D（漏洞在 C）

补丁函数（PF）：

A → B → X → C' → D（补丁修改了 B 之后的路径）

PatchDiscovery 创新分析

- 补丁存在判断
  - 对于给定TF，判断是否已经存在补丁修复漏洞
  - 搜索范围缩小
    - 通过基本块匹配，匹配出 TF 和 PF/VF 之间的相似基本块
  - 构造子控制流图（sub-CFG）
    - 通过匹配基本块，将 TF 进行划分，形成多个小的子控制流图
  - 相似性计算
    - $sim(kb, b) = \frac{len(lcs(kb,b))}{\frac{len(kb)+len(b)}{2}}$
    - $SIM(PF, TF) = \sum_{kb \in KB_P} \frac{\omega_{kb} \times sim(kb, kb_{h \in TF})}{\sum_{kb \in KB_P} \omega_{kb}}$

PatchDiscovery 实验设计

- 数据资源
  - 数据集：由**CVE Details、CVE List、NVD网站收集**
  - 二进制编译：默认优化（**-O2**）的**GCC编译为二进制可执行文件**
  - 二进制代码提取：使用 **IDA Pro** 反汇编工具
  - 计算指标： **Precision (P)、Recall (R)、F-Measure (F1) 、Accuracy (A)**

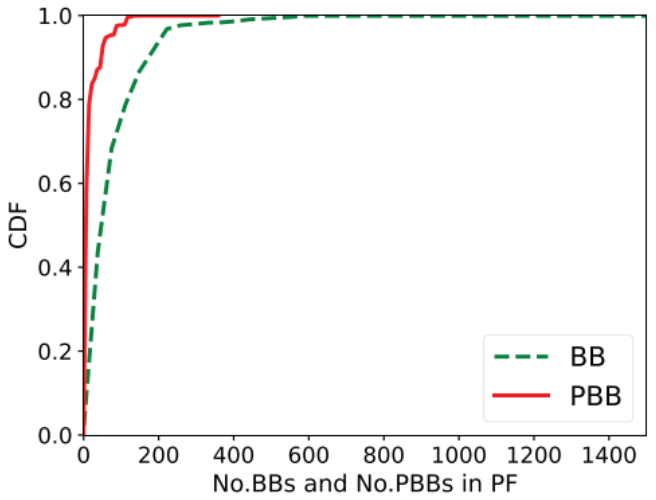
https://www.cvedetails.com/.
https://cve.mitre.org/index.html.
https://nvd.nist.gov/.

	Program	No. of CVEs	No. of VF-PFs	No. of Versions	No.of TFs
Dataset-I	Binutils	147	296	10	1061
	Expat	2	2	4	8
	FFmpeg	52	232	55	5570
	FreeType	57	82	19	776
	Libexif	7	8	12	77
	Libpng	4	11	18	162
	Libxml2	37	124	12	744
	Libxslt	3	3	5	15
	OpenSSL	70	142	22	2201
	Openvpn	6	13	7	46
	Sqlite3	6	14	10	97
	Tcpdump	88	150	3	252
Dataset-II	OpenSSL	45	72	8	598

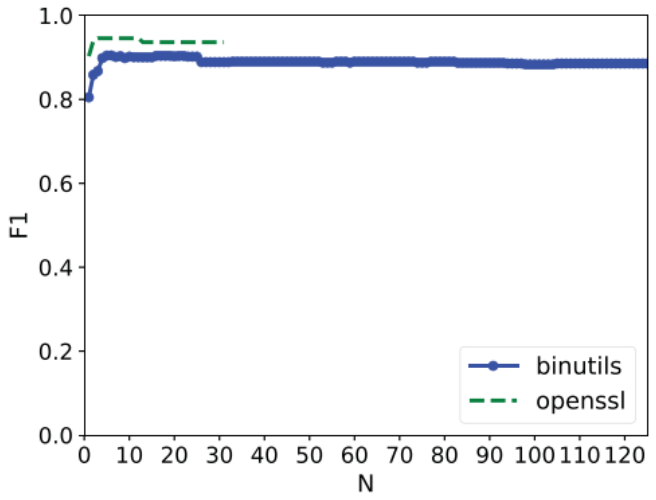
Term	Abbr	Definition
True Positive	TP	the number of patched functions that are correctly detected as patched.
True Negative	TN	the number of vulnerable functions that are correctly detected as vulnerable.
False Positive	FP	the number of vulnerable functions that are incorrectly detected as patched.
False Negative	FN	the number of patched functions that are incorrectly detected as vulnerable.
Precision	P	$TP/(TP+FP)$
Recall	R	$TP/(TP+FN)$
F-measure	F_1	$2PR/(P+R)$
Accuracy	A	$(TP+TN)/(TP+TN+FP+FN)$

PatchDiscovery 实验设计

- 参数实验
  - **N 控制用于生成补丁签名（Patch Signature）和漏洞签名（Vulnerability Signature）的关键基本块的数量**
  - **61.4% 的补丁函数（PF）包含 ≤ 8 个补丁基本块（PBBs）**
  - **38.6% 的补丁函数包含 > 8 个 PBBs，最多可达 359 个 PBBs**
  - **N = 5 为 PatchDiscovery 提供最佳性能**



(a) CDF of the number of patch basic blocks and all basic blocks in patched functions



(b) F_1 values regarding different N values

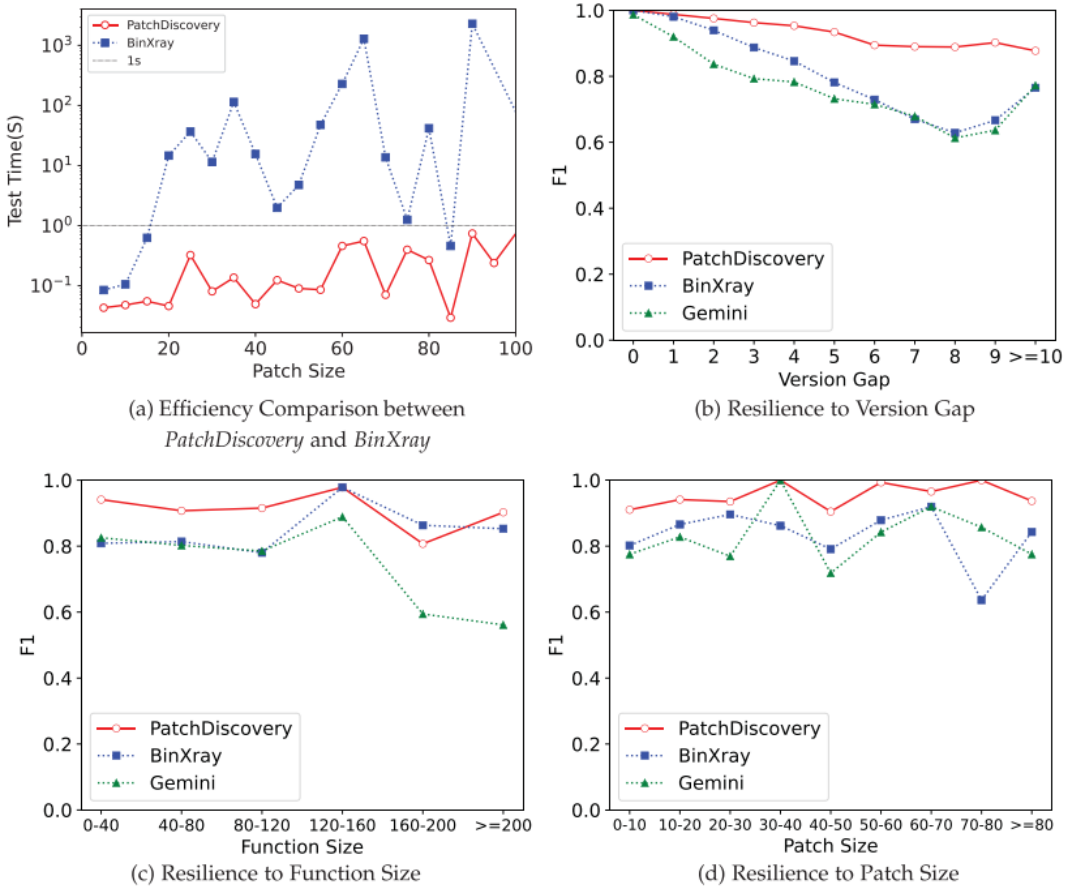
PatchDiscovery 对比实验

- 实验结果
  - **BinXray(2020)**：基于补丁签名和基本块追踪进行漏洞检测
  - **Gemini(2017)**：基于神经网络计算二进制函数相似度
  - **BinXray** 受限于计算开销，**143** 个目标函数超时（**>2** 小时），无法完成检测
  - **BinXray** 随着补丁大小增加，检测时间呈线性增长，而 **PatchDiscovery** 的检测时间基本不受补丁大小影响
  - 当版本差异增加时（**Version Gap > 9**），**Gemini** 和 **BinXray** 的 **F1** 分数显著下降

	P	R	F_1	A	TestRate	Total Time(s)
PatchDiscovery	0.943	0.903	0.922	0.930	1.000	4606.437
Gemini	0.793	0.784	0.788	0.797	1.000	3922.668
BinXray	0.831	0.809	0.820	0.833	0.987	159328.891

PatchDiscovery 对比实验

- 实验结果
  - **Function Size**：**VF**中基本块数量
  - **Patch Size**：**PF**中的补丁基本块数量
  - 补丁块大小：**PatchDiscovery** 仅搜索关键基本块，检测时间与补丁大小无关，因此效率更高
  - 版本差异性对比：版本升级引入了大量与补丁无关的代码变更，影响补丁匹配的准确性
  - 基本块数量：**PatchDiscovery**比**Gemini**和**Binxray**更具弹性



(a) Efficiency Comparison between *PatchDiscovery* and *BinXray*

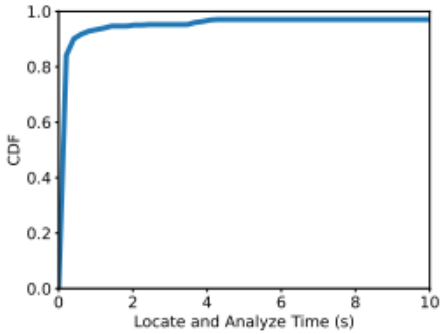
(b) Resilience to Version Gap

(c) Resilience to Function Size

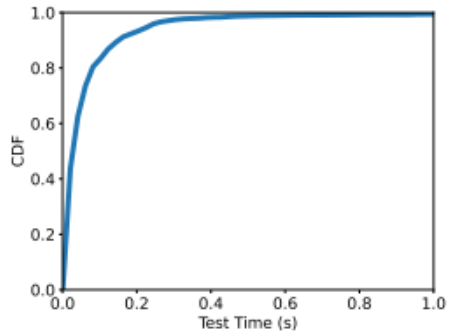
(d) Resilience to Patch Size

PatchDiscovery 有效性

- 实验结果
  - 绝大多数测试程序的准确率高于 **0.87**，整体准确率达到 **0.930**
  - 版本升级带来的大量代码变更，使得 无法匹配相同基本块来缩小搜索范围
  - 基本块匹配阶段：**92%** 的漏洞-补丁函数对计算时间 ≤ **1s**
  - **TF**补丁检测阶段：**77%** 的目标函数检测时间 ≤ **0.1s**，平均约**0.091s**



(a) CDF of the Run-Time Overhead of Patch Location–Patch Analysis (s)



(b) CDF of the Run-Time Overhead of Patch Presence Discovery (s)

Program	P	R	F_1	A	Total Time(s)
Binutils	0.867	0.945	0.905	0.931	471.598
Expat	1.000	1.000	1.000	1.000	0.131
FFmpeg	0.931	0.866	0.897	0.897	837.859
FreeType	0.983	0.954	0.968	0.976	130.621
Libexif	1.000	1.000	1.000	1.000	26.566
Libpng	1.000	1.000	1.000	1.000	3.142
Libxml2	0.983	1.000	0.991	0.991	42.144
Libxslt	1.000	1.000	1.000	1.000	1.196
OpenSSL	0.972	0.930	0.951	0.962	166.54
Openvpn	1.000	1.000	1.000	1.000	1.98
Sqlite3	0.932	0.873	0.902	0.876	153.052
Tcpdump	1.000	1.000	1.000	1.000	2771.608
Total	0.943	0.903	0.922	0.930	4606.437

PatchDiscovery 对版本差异的适应性

- 实验结果
  - 小版本差异（**0,1,2**）时，**PatchDiscovery** 检测效果最佳，准确率较高
  - 当版本差异变大（≥**3**），检测准确率下降，因为代码变更导致补丁/漏洞签名难以匹配
  - 当版本差异较大（≥**7**），部分目标函数的代码变更较少，因此漏洞签名仍然可以匹配

Version Gap	P	R	F_1	A
0	1.000	1.000	1.000	1.000
1	1.000	0.976	0.988	0.990
2	0.982	0.969	0.976	0.979
3	0.983	0.944	0.963	0.970
4	0.966	0.941	0.953	0.965
5	0.930	0.938	0.934	0.950
6	0.882	0.907	0.894	0.922
7	0.879	0.901	0.890	0.916
8	0.902	0.876	0.889	0.917
9	0.924	0.882	0.902	0.918
≥10	0.918	0.840	0.877	0.889
Average	0.943	0.903	0.922	0.930

PatchDiscovery 功能与补丁大小

- 实验结果
  - **Function Size**：**VF**中基本块数量
  - **Patch Size**：**PF**中的补丁基本块数量
  - 约**96%**的**VF**包含不超过**200**个基本块
  - **PatchDiscovery** 只关注补丁级别，而不是整个函数级别
  - 由于 **95%** 的**PF**中补丁基本块≤ **80**
  - 小补丁修改的代码量较少，容易被代码演化和编译优化消除或融合
  - 补丁规模增加后，补丁影响范围更大，不易被代码演化或编译优化抹去

Function Size	P	R	F_1	A
0-40	0.955	0.927	0.941	0.944
40-80	0.931	0.884	0.907	0.929
80-120	0.930	0.901	0.915	0.919
120-160	0.958	1.000	0.978	0.981
160-200	0.881	0.744	0.807	0.828
≥200	0.947	0.862	0.903	0.917
Average	0.943	0.903	0.922	0.930

Patch Size	P	R	F_1	A
0-10	0.942	0.880	0.910	0.924
10-20	0.938	0.945	0.941	0.940
20-30	0.929	0.942	0.935	0.953
30-40	1.000	1.000	1.000	1.000
40-50	0.928	0.883	0.905	0.919
50-60	0.987	1.000	0.993	0.991
60-70	0.933	1.000	0.966	0.948
70-80	1.000	1.000	1.000	1.000
≥80	0.932	0.942	0.937	0.930
Average	0.943	0.903	0.922	0.930

特点总结与未来展望

特点总结与未来展望

- 特点总结

优势	劣势
引入关键基本块来表示补丁签名	易受到代码演化和编译优化的影响
提高检测精度和计算效率	难以应对小补丁
减少因版本演化导致的误检	-

- 未来展望
  - 细粒度捕获补丁代码变化
  - 增强指令归一化，消除语法层面的编译优化影响

参考文献

[1] X. Xu et al. PatchDiscovery: Patch Presence Test for Identifying Binary Vulnerabilities Based on Key Basic Blocks[J]. IEEE Transactions on Software Engineering, vol. 49, no. 12, pp. 5279-5294.

道德经

知人者智，自知者明。胜人者有力，自胜者强。知足者富。强行者有志。不失其所者久。死而不亡者，寿。

谢谢！