计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2017,26(10):1−10 [doi: 10.15888/j.cnki.csa.006013]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

# 基于 Clang 编译前端的 Android 源代码静态分析技术①

曹原野[1,2], 丁丽萍[1]

[1](中国科学院 软件研究所 基础软件实验室, 北京 100190)
[2](中国科学院大学, 北京 100049)

**摘　要**: Android 手机在全球占有很大的市场份额, 基于 Android 衍生的第三方系统也为数不少. 针对 Android 系统重大安全问题频发的现状, 提出一种使用 Clang 编译前端对 Android 源码进行静态分析的方法. 该方法从已公布的 CVE 漏洞中提取规则和模型, 通过改进的 Clang 编译前端, 对 Android 源码进行静态分析, 从而检测出有潜在安全风险的代码片段. 在对 Android 源码进行污点分析时, 调用新加入的 stp 约束求解器, 通过符号执行, 对敏感数据进行污点标记, 并对敏感函数、敏感操作、敏感规则进行污点分析, 如果存在潜在的安全隐患, 则进行报告. 经过实验分析, 该方法可以找出 Android 源代码中存在的同类型有安全风险的代码片段, 可以检出 libstagefright 模块 5 个高危 CVE 漏洞.

**关键词**: Clang 编译器; 安卓; 静态分析; 污点分析; 符号执行

引用格式: 曹原野,丁丽萍.基于 Clang 编译前端的 Android 源代码静态分析技术.计算机系统应用,2017,26(10):1–10. http://www.c-s-a.org.cn/1003-3254/6013.html

## Android Source Code Static Analysis Technology Based on Clang Compiler Front-Ends

CAO Yuan-Ye[1,2], DING Li-Ping[1]

[1](Lab of Fundamental Software, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)

**Abstract**: Android phones have a large market share in the world, and the third-party system based on Android-derived is also very popular. As the security issues appear in Android systems frequently, this paper uses Clang to compile Android source code for static analysis. This analysis extracts rules and models from published CVE vulnerabilities, and uses the improved Clang to statically analyze Android source code to detect potentially unsafe code snippets. During the analysis of the Android source code, the Clang static analyzer taints attack surface, and calls the new added STP constrained solver. Then it taints sensitive data through the symbolic execution, and makes taint analysis on the sensitive functions, sensitive operations, sensitive rules, finally reports unsafe code snippets if there are potential security risks. Through experimental analysis, this method can accurately identify unsafe source code snippets that exist in the Android source code with the same type of security risk, and this method can detect five high-risk CVE vulnerabilities in the libstagefright module.

**Key words**: Clang compiler; Android; static analysis; taint analysis; symbolic execution

## 1　引言 111

Android 系统因其开源性和开放性, 在智能手机市场占有很大的市场份额. 近些年来, Android 系统不断发展, 在智能家电、物联网等领域迅速攻占市场, 与人们的日常生活越来越紧密相关. 然而, Android 系统的安全问题日益突出, 不断爆出影响极大的安全漏洞,

① 基金项目: 国家高技术研究发展计划 (“863”计划)(2015AA016003)
收稿时间: 2017-01-16; 采用时间: 2017-02-23



© 中国科学院软件研究所 http://www.c-s-a.org.cn

不仅影响了用户的使用体验, 更严重威胁到了用户的隐私, 例如: 具备麦克风或摄像头的 Android 智能电视被远程控制, 将会造成非常可怕的隐私泄露. 因此, Android 硬件厂商在硬件发布之前, 非常有必要对定制的 Android 系统进行安全审计, 以尽可能地减少潜在的安全隐患. 对定制的 Android 源代码进行安全审计在保护用户的隐私、减少企业的开发成本方面有重要意义.

在研究 Android 源代码的 CVE(Common vulnerabilities and exposures) 漏洞补丁的过程中, 发现部分漏洞存在特定的规律和模式. 本文对这些规律和模式进行了总结分析, 提出了一种基于改进的 Clang 编译前端进行安全分析的静态检测方法, 该方法通过 Clang 编译前端对指定的 Android 模块进行符号执行[1,2], 通过污点传播对敏感数据进行标记, 通过对特定的函数[3,4]、条件分支、语法结构进行污点分析, 找到有潜在安全风险的代码片段, 并给出相应的检查报告. 企业在对 Android 系统进行定制修改之后, 如果能够在系统发布之前, 对 Android 定制系统的源代码进行安全审计, 则可以有效降低定制系统的安全风险所带来的损失[5], 同时因为审计的自动化, 也可以大幅降低人力成本.

## 2 相关研究

代码审计的最终目标是挖掘软件中潜在的有安全风险的漏洞代码. 目前, 针对 Android 系统的漏洞挖掘已经有很多成熟的方法. 例如: 人工审查、模糊测试、动态分析、静态分析等方法.

人工审查就是通过人工阅读代码的方式对源代码进行逐行逐行地分析, 判断是否有安全问题. 这要求审查人员有相当深厚的审查经验, 而且很费时间, 适用于简短但是易错的代码, 如驱动代码.

模糊测试 (Fuzz) 是指通过给目标接口输入大量的随机数据, 来测试接口是否能够正常处理这些畸形数据的测试方法. 如果接口或系统出现了异常, 则认为接口极有可能存在缺陷. 模糊测试的优点是简单有效, 但是相当耗费时间, 因为是随机生成的数据, 所以具有一定的概率性. 而且测试速率要受制于目标接口的响应速率. 对应的工具有 Peach 等. Peach 是一个遵守 MIT 开源许可证的模糊测试框架, 用户通过编写 Peach Pit 配置文件, 可以定义 Fuzz 过程的配置和原始数据结构的定义. 通过定义原始数据结构生成结构化的部分随机数据, 来实现精确的模糊测试.

动态分析是指对二进制程序进行插桩, 来达到运行时对程序的控制和监控. 动态分析的优点是准确率高, 覆盖面大, 但是缺点是依赖于运行平台, 有路径爆炸的风险, 并且在编译过程中损失了一些代码的细节信息. 对应的工具有 Pin、KLEE、Android_S2E[20]等. Pin 是 Intel 公司提供的二进制插桩工具[6], 它允许在可执行程序的任何地方插入任意代码[7]. KLEE 是一款源代码动态符号执行工具. KLEE 需要修改源代码, 通过对源代码内特定的输入数据进行标记, 插入自己的符号执行代码, 然后编译执行并收集变量信息. Android_S2E 是一款全系统符号执行工具, 它预装 QEMU 模拟器和 KLEE 在虚拟机上运行 Android, 并执行符号执行, 可以遍历 Android 系统上小型 C 语言程序的所有路径.

静态分析是指在不运行代码的情况下, 通过多种方式对源代码进行分析之后, 得出源代码是否存在问题的结论. 分析方法涵盖简单的正则、词法分析、语法分析、上下文路径敏感分析等. 对应的工具有 Coverity、Clang 等. Coverity 是一个先进的、可配置的用于检测软件缺陷和安全隐患的静态代码分析解决方案, 它能自动化地检测和解决 C、C++、Java、C#源代码中多种类型的缺陷, Coverity 对 Android 系统有针对性优化, 但是 Coverity 是收费且闭源的. Clang 是 LLVM 开源编译套件的一个编译前端, 负责将 C、C++、Object-C 源代码翻译为中间代码[8,9], 而且 Clang 自身具备静态分析的能力, 可以在不运行代码的情况下对代码进行编译分析.

现有 Android 系统安全问题的文献研究多数集中在 Android 应用层上的研究. 对于 Android 系统自身底层代码的安全研究的文献研究其实不常见[10,11]. 移动安全会议上提到的工业界常用的方法多集中于模糊测试的方法. 使用模糊测试的方法虽然具有一定的随机性, 但是简单便捷, 并且短期内容易看到成效. 但是, 安全研究员如果想要建立一种长效的安全机制, 这种机制是稳定的, 而且可以逐步完善、逐步归纳吸收已有的安全风险的话, 使用静态分析是一种非常合适的方案.

静态分析没有模糊测试的随机性, 分析结果比较稳定. 并且, 通过对现有的安全风险进行归纳吸收, 可以不断完善静态分析工具, 那么静态分析工具就能够在检测能力上得到持续加强. 而一款能够持续加强的静态分析工具必须是开源的, 所以本文主要调研了 Clang



© 中国科学院软件研究所 http://www.c-s-a.org.cn

编译前端对 Android 源代码进行静态分析的难点.

本文经过调研, 使用 Clang 成功对 Android 源代码进行了分析, 并主要克服了 3 个方面的困难. 1) 新版的 Clang 3.8 无法直接对 Android 源代码进行分析, 需要做兼容处理. 2) Clang 自带的约束求解器的求解能力过弱, 不适合应用在 Android 源代码这种复杂项目上. 3) Clang 自带的检测规则是通用检测规则, 没有针对 Android 源代码的检测规则, 导致检测能力过弱.

本文克服了上述 3 个困难, 最终实现了目标: 实现一种工业界可用的, 基于改进的 Clang 编译前端的, 用于 Android 系统源代码的静态分析方法.

## 3 基于 Clang 编译前端的静态分析原理

Clang 作为 LLVM 编译套件的编译前端, 自带一个静态代码分析工具, 可以编译分析 C、C++、Object-C 源代码, 此静态分析工具属于 Clang 的一部分因而完全开源.

Clang 在对源代码进行编译的时候, 会通过静态分析, 对代码风格、语法错误和潜在的风险进行 warning 警告. Clang 静态分析器通过自定义代码可以实现更复杂的检测机制来进行静态分析. Clang 的静态分析器首先通过程序代码, 生成 AST(语法树), 然后根据 AST 生成 CFG(控制流图), 通过符号执行生成扩展图(ExplodedGraph), 具体控制流程如图 1[12]所示.

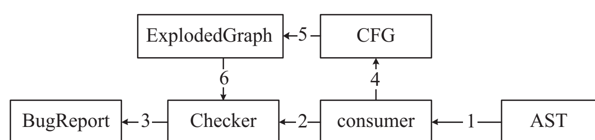


图 1 Clang 的静态分析过程

通过自定义 Checker(Clang 中的检查器, 用来检测自定义缺陷), 可以实现对 AST 和 ExplodedGraph 的分析和控制. Checker 不仅仅只能是被动的调用, Checker 也能主动地修改 ExplodedGraph. 如果 Checker 认为存在特定的缺陷, 就可以调用 BugReport 来报告存在的缺陷, 方便测试人员可视化地观察检测信息.

Clang 的 AST 树近似于常规的编程语法结构. 通过“clang-Xclang-ast-dump example.cpp”可以对源代码生成的 AST 进行观察. AST 语法树主要由两类结点(Stmt 与 Decl) 和派生自他们的子结点构成. Stmt 是指 statement 语句, Decl 是指 declarations 声明.

Clang 的 CFG 控制流图由若干个 CFGBlock 块组成, 每个 CFGBlock 均包含有 EXIT 块和 ENTRY 块. 并且每个 CFGBlock 块由一组 CFGElement 组成, 其中的 CFGElement 代表 AST 树中的一个结点.

Clang 的 ExplodedGraph 图是在通过符号执行遍历 CFG 图的时候产生的. ExplodedGraph 中的每个结点, 都包含了 ProgramPoint 点和 State 信息. ProgramPoint 点, 用来定义源代码运行到该点时所有变量的值. State 信息用来表示源代码在符号执行过程中的状态信息, 包括: 表达式到值的映射关系、各种变量到值的映射关系、路径条件约束等信息.

通过 Clang 的源码, 可以发现, 如果将 Clang 的静态分析直接应用到 Android 系统源码之上, 基本没有效果. 一个原因是 Clang 自带的 range 约束求解器的求解能力过于薄弱, 只能处理简单约束. 另一个原因是 Clang 作为一个通用编译前端, 更注重的是通用性, 所以特定问题的检测能力不强. 需要对 Clang 的符号执行能力和检测能力进行增强. 所以, 以下小节将逐个描述对 Clang 进行加强的实现原理.

### 3.1 改进 Clang 静态分析 Android 源码的兼容性

Android 源码内, 含有 LLVM 的预编译工具链, 但是直到最新的 Android7.1 的源代码, LLVM 预编译工具链中 Clang 的版本仍旧是 3.3 版本. 本文需要定制较新的 Clang3.8 版本进行静态分析. 所以需要了解 Clang 静态分析指令和 Android 源代码的编译脚本工作原理.

Clang 静态分析由两个指令构成, 分别是“scan-build”和“scan-view”命令. “scan-build”命令后面紧接着正常的编译命令, 例如“scan-build g++ example.cpp”或者“scan-build make-j4”这种编译命令. 其中“scan-build”命令会对后面的编译命令进行分析, 替换其中的编译器为 Clang 本身, 则 Clang 可以顺利地进行静态分析. “scan-build”在分析完毕之后, 会留下一个特殊的目录, 使用“scan-view”打开这个目录, 则可以显示分析报表.

Android 源代码的编译脚本主要集中在源代码根目录的 build/envsetup.sh 文件里. 其中主要有 4 个编译命令, 分别是 mmm、mm、m、make 命令. mmm 命令主要用于编译指定路径下的模块, mm 命令用于编译当前路径下的模块, m 命令编译当前目录下的所有模块, make 命令则是编译整个 Android 系统. 为了静态分析的需要, 主要进行单个模块的分析, 所以选用 mmm 命令. 通过更换 mmm 命令后的路径, 可以实现不同模块的静态分析. “mmm path-B”其中的“-B”参数则是无论



© 中国科学院软件研究所 http://www.c-s-a.org.cn

源码修改与否都强制编译 path 目录下的所有源码.

直接使用“scan-build mmm path-B”则会直接报错. 通过研究 envsetup.sh 的代码, 发现在 getdriver 函数声明了静态分析的使用规则和“scan-build”的二进制路径, 将 getdriver 函数的“scan-build”路径改为改进的 Clang 源代码生成的“scan-build”路径. 同时执行“WITH_STATIC_ANALYZER=1 mmm path-B”则可成功启动改进 Clang 的静态分析. 但是, 分析途中一定会出错, 因为高版本的 Clang 已经不支持一些旧命令参数, 所以需要对“scan-build”编译脚本调用的其他编译脚本进行兼容修改. 经过修改的的编译脚本可以正确对某个模块进行静态分析.

### 3.2 增强 Clang 的符号执行能力

SMT(Satisfiability modulo theories) 可以用来解决逻辑公式的决策问题[13], 可以用来处理布尔运算、量词、算术运算、比较运算、位运算等约束性求解问题. SMT 模型有一些具体的约束求解器实现, 例如: z3 约束求解器、stp 约束求解器等. 这些求解器具备强大的求解能力, 可以处理编程语言中常见的复杂条件判断、位运算、数组操作等.

符号执行是 CFG 生成 ExplodedGraph 时使用的. Clang 静态分析程序读取 CFG, 当遇到条件分支时, 会根据不同的条件分支, 依次执行不同的条件分支. 在执行不同的条件分支之前, 符号执行会把进去当前分支所需要的条件放入约束求解器查询, 如果当前条件满足约束求解器, 则执行分支并把分支条件放入约束求解器, 否则不执行当前分支.

Clang 具备符号执行的能力, 但是其自带的 range 约束求解器能力不强, 只支持简单约束条件, 遇到多元约束条件不能处理则忽略, 这大大削弱了符号执行的准确性. 在实际的复杂代码中, 这个自带的约束求解器在精度方面经常不能满足预想的要求.

但是 Clang 本身支持约束求解器的扩展, 可以通过仿照 range 约束求解器的编写方式, 修改 Clang 配置文件, 增加新的约束求解器, 达到增强约束求解器的目的.

本文通过修改 Clang 源码, 加入了 stp 约束求解器, 使 Clang 可以处理复杂的多元约束求解. 通过增强 Clang 符号执行的约束求解能力, 提高其检测精度.

### 3.3 增强 Clang 的静态检测能力

作为一款通用编译前端, Clang 所面对的是通用场景的代码. 其静态分析里的检测规则面向的是常见的缺陷问题. 所以, Clang 对于特殊模式的问题无法进行正确检测而得出结果. 于是, 在使用 Clang 对 Android 的源码进行分析之前, 需要预先增强 Clang 的静态检测能力.

对于 Android 系统, 如果 Clang 想要检测出安全风险, 就需要预先知道可能的攻击面. Android 系统因为丰富的交互功能, 其本身具有很多潜在的攻击面. 攻击面可以分为 4 类. 1) 远程攻击面: 主要包含了网络协议栈、对外暴露的网络服务、短信、彩信、基础软件(浏览器接口、多媒体和文档处理、电子邮件) 等. 2) 物理相邻的攻击面: 无线、蓝牙、NFC 等. 3) 本地攻击面: 文件系统、系统调用等. 4) 物理连接攻击面: USB 连接等.

可被用户控制的数据从这些攻击面, 流入到系统的控制流中, 如果处理不当, 则可能会引入安全风险. 所以, 静态分析的主要目标, 就是需要频繁从攻击面获取数据的模块, 例如: libstagefright 模块. 为此, 需要设定一段污点传播程序, 对 Android 源码中有可能从攻击面引入的危险数据加上污点标记. 并对污染的数据进行污点传播, 即图 2 的“敏感函数污点标记”. 敏感函数例如: 内存操作函数、网络操作函数、缓冲区管理函数、文件管理函数等.

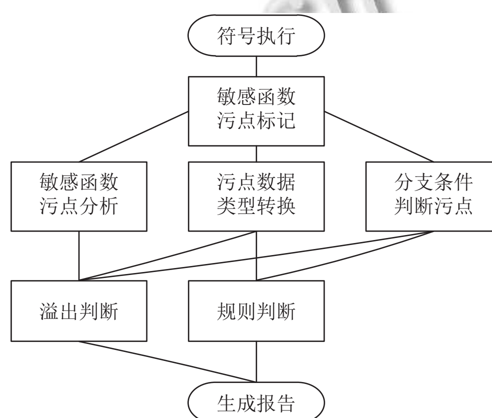


图 2 增强的静态检测逻辑

在 Android 源代码里, 充满了大量的宏定义和复杂的工具类, 给开发人员带来了巨大的便利, 但是也给开发人员带来了困扰: 有时候会不记得当前的变量到底是什么类型, 有时候会忘记一些必须的操作. 以下总结一些在 Android 源代码漏洞中常见的错误模式.

(1) 敏感函数的参数错误



© 中国科学院软件研究所 http://www.c-s-a.org.cn

memcpy(dest, src, len);

在复杂逻辑中, 开发人员可能会忽略对 len 的正确约束, 导致 len 可以赋值为一个超大的整数值.

(2) 隐式转换

memcpy(dest, src, len);

在开发中, 开发人员混淆了 len 的类型, 直接将有符号数直接传入参数, 同时没有限制 len 必须大于等于 0.

long long a = int_b + int_c + (long long)d;

在开发中, 遇到一个复杂计算式, 而且数据类型不一致时, 开发人员意识到需要进行强制类型转换. 但是强制类型转换过晚, int_b 和 int_c 已经发生了溢出并进行了隐式转换之后, 才与 d 变量进行相加.

(3) 分支条件

if(int_e+100<int_f)

在开发中, 开发人员忘记 int_e 的范围没有进行约束, 有可能发生溢出.

if(int_j < uint_k)

在开发中, 开发人员忘记 int_j 的类型, 且没有约束 int_j 必须大于等于 0.

为了对上述 3 种错误模式进行追踪检查, 要预先设定钩子函数进行捕获, 当钩子函数捕捉到其中之一的模式正在发生, 且包含有污点数据的时候, 启动进一步分析. 例如: 内存分配函数、变量隐式转换、条件判断表达式这三种情况. 这三种情况依次对应图 2 中的"敏感函数污点分析", "污点数据类型转换", "分支条件判断污点". 启动进一步分析之后, 对应到图 2 中的"溢出判断"和"规则判断".

"溢出判断"是指通过获取污点变量的取值范围[14,15], 判断是否存在于合法的区间. 首先, 通过 Clang 获取污点变量的类型, 根据变量类型生成对应的最大值和最小值. 通过 stp 约束求解器的查询功能, 查询污点变量是否能够取到最大值的同时再取到最小值, 如果可以同时取到, 则认为存在风险.

"规则判断"是指通过设定若干组语法树规则, 对含有污点数据的语法树进行匹配判断. 例如: 当发现污点数据在二元算式之后发生了类型提升, 而且是隐式类型提升的时候, 说明在编程语法上存在安全风险, 此时可提示有潜在的安全风险.

## 4 Android 源码静态分析的实现

### 4.1 对 Android 源代码的修改

建立 Android 源代码的本地 mirror, 从 mirror 拉取 branch 为 android-5.1.1_r14(为了进行对比实验, 需要拉取若干个 branch) 的分支, 安装好编译工具, 配置 Android5.1.1 的编译环境和二进制输出目录, 进入 Android 代码目录, 执行代码 1 的指令.

代码1. 编译所有Android源代码

```
$ source build/envsetup.sh
$ lunch aosp_arm-eng
$ make -j4
```

在静态分析之前, 必须先整体编译一次 Android, 因为一个模块在编译的时候经常会依赖其他模块, 如果在静态分析前先全部编译一次, 就不会出现找不到依赖模块的情况.

Android 整体编译完成之后, 在 Android 的源代码目录下编辑 build/envsetup.sh 下按照代码 2 进行编辑. 修改--use-analyzer 参数后面的路径, 更新为修改过的 Clang 程序的二进制文件.

Android 源代码的工具链自带了一套 Clang3.3 编译前端, 但是其版本太低, 需要如代码 3 里所示, 修改 /prebuilts/misc/linux-x86/analyzer/bin/ccc-analyzer 文件, 做下列兼容工作, 否则改进的 Clang3.8 无法正常运行. 其主要操作是: 通过参数指定 Clang3.8 使用新加入的 stp 约束求解器, 并且替换一些传递的过时的命令行参数.

代码2. 替换新的分析工具

```
function getdriver(){
…
--use-analyzer $T/prebuilts/misc/linux-x86/analyzer/bin/analyzer \
…
}
```

代码3. 针对Clang3.8做兼容

```
  push @Args, "-Xclang", "-analyzer-viz-egraph-ubigraph";
  }
+ //强制Clang编译器使用本实验新加入的stp求解器
+ push @Args, "-Xanalyzer", "-analyzer-constraints=stp"; my
$AnalysisArgs = GetCCArgs("--analyze", \@Args); @CmdArgs =
@$AnalysisArgs;
+ //对参数进行修改, 以适应新的Clang编译器
+ my @NewCmdArgs;
+ foreach my $one (@CmdArgs){
+     if($one eq "-Qignore-c-std-not-allowed-with-cplusplus"
+         || $one eq "-fobjc-default-synthesize-properties"
+         || $one eq "-fno-cxx-missing-return-semantics")
+     {next;}
+     if($one eq "-disable-global-ctor-const-promotion"){
+         push(@NewCmdArgs, "-disable-cgp-ext-ld-promotion");
+     next;
+     }
```



© 中国科学院软件研究所 http://www.c-s-a.org.cn

```
+    push(@NewCmdArgs, $one);
+    }
+ @CmdArgs = @NewCmdArgs;
```

代码4. 替换新的Clang编译前端

```
if($pid == 0) {
   close FROM_CHILD;
   open(STDOUT, ">&", \*TO_PARENT);
   open(STDERR, ">&", \*TO_PARENT);
+    //置为实验修改的Clang的二进制的地址
+    $Cmd = "xxx";
   exec $Cmd, @CmdArgs;
}
```

再如代码 4 所示, 修改/prebuilts/misc/linux-x86/analyzer/bin/ccc-analyzer 文件, 修改内部使用的$Cmd 变量. 从而完成对 Android 源码的修改. 此时, 已经可以使用 Clang3.8 正常编译分析 Android 系统源代码.

### 4.2 添加 stp 约束求解器

首先, 安装 stp 约束求解库, 需要安装 minisat 库和 stp 库. 然后, 进入 llvm 源码, 在 tools/clang/lib/StaticAnalyzer/Core/目录下仿照现有的 RangeConstraintManager.cpp 编写调用 stp 求解器的 STPConstraintManager.cpp, 并对外提供接口 CreateSTPConstraintManager 返回 STPConstraintManager 类的实例供静态分析核心使用.

STPConstraintManager.cpp 可以参考网络上已有的样例约束求解器代码进行修改, 但是需要注意因为本文使用的 Clang 是新版的, 必须注意代码的兼容性问题 (过时的接口、类或结构体的成员的变动), 否则程序无法正常执行.

最后, 如代码 5 所示, 在 tools/clang/include/clang/StaticAnalyzer/Core/Analyses.def 中加入如下一行声明 (注意要和 STPConstraintManager 类的接口 CreateSTPConstraintManager 名称相同), 可以支持在命令行选择实际使用的求解器 (range 求解器或者 stp 求解器), 方便进行对比实验分析.

代码5. 增加新的约束求解器

```
ANALYSIS_CONSTRAINTS(STPConstraints, \
"stp", "Use STP solver", \
CreateSTPConstraintManager)
```

### 4.3 改进 Clang 的 Checker

如图 1 所示, Clang 编译前端在静态分析中, 真正执行缺陷判断的是一系列的 Checker. 所以, 为了增强 Clang 的静态检测能力, 必须对 Clang 的一系列的 Checker 进行增强.

污点传播部分, 修改 LLVM 代码里的 tools\clang\lib\StaticAnalyzer\Checkers\GenericTaintChecker.cpp. 这个 Checker 支持对 C 的函数进行污点标记和传播, 但是不支持对 C++类的处理. 对这个 Checker 进行增强, 加入对 C++类的判断和处理.

因为 Android 系统代码的复杂性, 系统代码对相当一部分的文件和内存操作都进行了封装. 在对 C++的成员方法进行分析时, 可以对参数或者返回值进行污点标注. 在实际标注的时候可以根据待分析代码的具体特征 (例如: 通用 buffer 缓冲类, 及其他文件操作、内存操作的通用类) 添加特定处理, 来进行污点设定. 开启 (默认 Clang 不开启) 这个修改过的 Checker 即可实现数据的污点标记.

"敏感函数污点分析"部分可以通过表 1 中 check::CheckPreCall 抢在函数调用之前, 进行分析. "污点数据类型转换"可以通过 check::CheckPreStmt<Expr>进行拦截判断, 通过对 Expr 的判断, 抽取其中的隐式转换和显式转换, 进行分析. "分支条件判断污点"可以通过 check::CheckBranchCondition 对条件分支进行拦截, 并对其中的 Stmt 进行判断.

表 1　Clang 的路径敏感分析接口

| 分析接口名称 | 调用时机 |
| --- | --- |
| check::PreStmt<T> | 声明之前调用 |
| check::PostStmt<T> | 声明之后调用 |
| check::PreCall | 调用接口之前调用 |
| check::PostCall | 调用接口之后调用 |
| check::Location | 读写变量时调用 |
| check::Bind | 绑定变量时调用 |
| check::BranchCondition | 发生条件分支时调用 |
| check::EndAnalysis | 整体分析结束时调用 |
| check::EndFunction | 函数分析结束时调用 |
| check::LiveSymbols | 垃圾回收之前调用 |
| check::DeadSymbols | 完成垃圾回收之后调用 |
| check::PointerEscape | 指针不能追踪时调用 |
| check::RegionChanges | 内存操作时调用 |
| eval::Assume | 有新的条件约束之前调用 |
| eval::Call | 函数过程分析之前调用 |

图 2 中的"溢出判断"可以通过 Clang 内置的 SValBuilder 类的 evalBinOpNN 方法对污点变量的范围进行估计, 测试污点变量是否经过正确约束.

图 2 中的"规则判断"则可以通过 ASTContext 类的 getParents(node) 方法获取父节点, 或者通过 node 本



© 中国科学院软件研究所 http://www.c-s-a.org.cn

身自带的各种方法获取其子结点. 甚至可以使用 Clang 内置的 RecursiveASTVisitor 类实现对某个结点的子结点进行自动递归查询.

### 4.4 漏洞代码的特征分析及处理

代码 6 展示了 CVE-2015-1538 的部分补丁.

代码6. CVE-2015-1538部分补丁代码

```
  mTimeToSampleCount = \
U32_AT(&header[4]);
- uint64_t allocSize = \
mTimeToSampleCount * 2 * sizeof(uint32_t);
+ uint64_t allocSize = \
mTimeToSampleCount * 2 *
(uint64_t)sizeof(uint32_t);
```

代码 6 中的缺陷存在着固定的模式: 污点数据由 uint32 类型参与运算提升为 uint64 类型, 但程序员没有意识到溢出发生在类型的隐式转换之前. 所以引入了潜在的溢出风险, 故补丁加入了强制转换.

但是, 代码 6 里的补丁仍旧是错误的！ 这导致了后序的漏洞 CVE-2015-6601. 仔细观察代码 6 中的补丁, 程序员的补丁里显然意识到了污点数据隐式转换的危险性, 但是程序员忘记了计算是从左到右执行的. 强制转换太迟了, 污点数据仍旧可以先溢出, 再提升类型.

如代码 6 所示, U32_AT 是 libstagefright 从缓冲区读取数据的公共方法, 类似的还有 U16_AT、U64_AT. U32_AT 函数会触发图 2 中的"敏感函数污点标记", 使 mTimeToSampleCount 变量被标记. 代码 6 中对 allocSize 的赋值操作中, 实际上发生了类型提升 (未打补丁是隐式提升, 打了错误的补丁是显式提升). 如果表达式中包含污点数据且存在类型提升, 则 Chekcer 触发图 2 中的"污点数据类型转换"会对其中的类型转换进行检查.

代码 7 展示了 CVE-2015-6604 的部分补丁.

代码7. CVE-2015-6604部分补丁代码

```
      return false;
  }
- if (offset + dataSize + 10 > mSize) {
+ if (dataSize > mSize-10- offset) {
      return false;
  }
```

代码 7 中的缺陷存在着固定的模式: 在一个条件分支语句中, 程序员忽视了污点数据的存在, 过分相信变量, 直接让污点数据参与了计算, 之后再参与比较判断.

如代码 7 所示, 其中 dataSize 的数据, 由 U32_AT 方式获取, 故被标记为污点数据. 但是在 if 条件判断语句中, 因为条件判断表达式含有污点数据, 故触发图 2 中的"分支条件判断污点", 对条件表达式进行求值分析.

### 4.5 Clang 的报告生成

在 4.3 章中使用 Clang 的 Checker 对待检测代码进行分析之后, 如果发现潜在的问题, 则使用 Clang 自带的 BugReport 进行风险代码的报告. 通过定义一个自定义 BugReport, 可以实现特定格式内容的 Bug 报告. 在分析结束之后, 通过 scan-view 命令打开分析报告网页, 对分析结果进行查看.

## 5 实验及结果分析

### 5.1 试验环境

本文使用 Ubuntu16.04 LTS X64 操作系统, 处理器 Intel 酷睿 i7 4750HQ(3.2 GHz), 16 GB 内存, 80 GB 交换分区, 1000 GB 机械硬盘. 基于 Clang3.8 进行测试修改. 静态分析期间, 不同时运行其他程序, 以免影响时间和内存使用的实验结果.

Clang 运行参数方面, 除了第 3 章节所做的代码修改和实验过程中输入的动态参数, 其余配置保持默认参数不变. 因为符号执行会出现路径爆炸问题, Clang 会使用一个默认最大循环执行次数来进行限制, 默认值是 4, 本实验不修改.

### 5.2 使用测试集对改进后的 Clang 进行验证

经过改进的 Clang 不能直接用于 Android 系统的代码分析, 需要首先确保修改的正确性, 保证改进后的 Clang 编译前端本身没有逻辑错误. 一个错误的 Clang 编译前端在分析 Android 源代码时的结果是不可信的. 而改进的 Clang 编译前端的代码修改集中在符号执行和整形溢出检测上, 所以需要一组整形溢出测试集验证 Clang 的符号执行精度和整形溢出检测能力.

测试集的选取, 选用的是从 NIST 网站下载的美国 NSA (National security agency) 开发的 Juliet_Test_Suite[24]测试集. Juliet_Test_Suite 测试集包含多种 CWE (Common weakness enumeration) [16]类型的常见漏洞代码. 本实验选取其中的"CWE190_Integer_Overflow"测试集. 其测试集下总共有 5 种类型的子测试集, 分别编号 S01, S02, S03, S04, S05.

通过特定的宏定义, 可以调整测试集的检测特性.



© 中国科学院软件研究所 http://www.c-s-a.org.cn

如: 只检测存在有安全缺陷的函数、只检测不存在安全缺陷的函数、检测全部函数. 表 2 显示了 5 个子测试集的函数用例数量.

表 2 测试集溢出漏洞用例数量 (个)

| | 安全用例数量 | 不安全用例数量 |
|---|---|---|
| S01 | 550 | 550 |
| S02 | 550 | 550 |
| S03 | 550 | 550 |
| S04 | 550 | 550 |
| S05 | 500 | 500 |

测试过程中, 采用子测试集逐个进行测试的方式执行. 单个子测试集在测试的时候, 分为六次测试. 前两次使用未修改的原生的 Clang 且使用自带的 range 约束求解器分别测试安全函数和不安全函数. 中间两次使用改进的 Clang 且使用自带的 range 约束求解器分别测试安全函数和不安全函数, 后两次使用改进的 Clang 且使用新加入的 stp 约束求解器分别测试安全函数和不安全函数. 运行静态分析的时候, 记录分析所需要的时间, 并分析所得到的结果, 测试所需时间如表 3 所示. 其中, good_origin_range 指原生的 Clang 使用自带的 range 求解器只检测安全函数, good_improved_range 指改进的 Clang 使用自带的 range 求解器只检测安全函数, good_improved_stp 指改进的 Clang 使用新加入的 stp 求解器只检测安全函数. bad_origin_range, bad_improved_range, bad_improved_stp 指相同的情况下只检测不安全函数.

表 3 静态分析所需时间 (秒)

| | S01 | S02 | S03 | S04 | S05 |
|---|---|---|---|---|---|
| good_origin_range | 122 | 119 | 118 | 118 | 108 |
| good_improved_range | 299 | 328 | 318 | 301 | 283 |
| good_improved_stp | 321 | 355 | 360 | 324 | 307 |
| bad_origin_range | 105 | 107 | 106 | 104 | 96 |
| bad_improved_range | 273 | 296 | 297 | 276 | 258 |
| bad_improved_stp | 284 | 309 | 306 | 287 | 269 |

通过表 3 分析发现, 不管是只检测安全函数还是只检测不安全函数, 在都使用自带的 range 求解器的情况下, 改进的 Clang 比原生的 Clang 检测时间增长了约两倍, 这是因为改进的 Clang 增加了污点传播等更多的检测规则, 导致检测时间的增大. 同样是改进的 Clang, 使用新增加的 stp 求解器比使用 range 求解器耗费的时间都要多, 但是基本不多于 10%, 这是因为 stp 的求解能力更强, 规则更复杂, 导致了耗费时间的增加.

表 4 中"good_origin_range"和"bad_origin_range"分别指原生 Clang 使用自带的 range 求解器测试安全用例和不安全用例的情况. 通过测试发现, 基本只能找到"Dead assignment", "Dead initialization", "Memory leak"这三种错误. 不管是安全测试用例还是不安全测试用例, 均无法检出任何溢出漏洞. 这是因为原生的 Clang 本身无对应的检测规则.

表 4 检出溢出类型漏洞数量 (个)

| | S01 | S02 | S03 | S04 | S05 |
|---|---|---|---|---|---|
| good_origin_range | 0 | 0 | 0 | 0 | 0 |
| good_improved_range | 568 | 590 | 653 | 568 | 569 |
| good_improved_stp | 45 | 84 | 63 | 45 | 63 |
| bad_origin_range | 0 | 0 | 0 | 0 | 0 |
| bad_improved_range | 363 | 452 | 453 | 363 | 404 |
| bad_improved_stp | 99 | 162 | 115 | 99 | 115 |

表 4 中的"good_improved_range"和"good_improved_stp"分别指改进的 Clang 使用自带 range 求解器或 stp 求解器分别对安全用例的检测情况. 改进的 Clang, 因为检测规则的增加, 已经可以检测溢出漏洞. 但是, 安全测试用例中是没有不安全的用例的. 所以, 通过比较, 在使用 range 求解器的情况下, 有超高的误报, 而 stp 的误报则是非常的低. 具体原因是 range 求解器的能力过弱, 很多复杂约束不能处理, 导致实际上没有成功加入约束条件, 导致了大量的误报. 其中误报的数量甚至超过了测试用例的数量, 一个原因是某些用例有多种测试方法, 此外如图 2 所示, 改进的 Clang 有 3 种触发方式进行缺陷分析, 一个函数由若干行的代码构成, 所以一个函数体可能检出不少于 1 个漏洞.

不安全测试用例在源代码的对应代码行的注释中, 进行了标注. 表 4 中的"bad_improved_range"和"bad_improved_stp"分别指改进的 Clang 使用自带 range 求解器或 stp 求解器分别对不安全用例的检测情况. 通过对测试报告的排查, stp 求解器虽然检出的结果比较少, 但是非常准确. range 求解器检出的结果存在大量的误报, 将测试用例的源代码中, 没有标记为缺陷位置的代码检测为存在缺陷.

收集测试用例的静态分析结果, 同时收集静态分析日志, 结合在一块进行分析, 可以得出结论: stp 求解器虽然在时间耗费上比 range 求解器稍多, 但是在约束求解精度上, stp 求解器有很大的提高; 同时结合静态分析日志和测试用例源代码, 对比 stp 求解器的约束区间, 可以发现求解器计算出了正确约束区间.



© 中国科学院软件研究所 http://www.c-s-a.org.cn

### 5.3 使用改进的 Clang 编译前端对 Android 源码进行分析

参考第 4 节的“Android 代码静态分析的实现”里的实现, 整个 Android 源代码在编译和修改完毕之后, 运行代码 8 里的指令, 即可使用改进的 Clang 且使用 stp 约束求解器对 Android 源码进行静态分析.

代码8. 执行静态分析

```
$source build/envsetup.sh
$lunch aosp_arm-eng
$WITH_STATIC_ANALYZER=1 mmm model_path -B
```

本实验主要通过“WITH_STATIC_ANALYZER=1 mmm frameworks/av/media/libstagefright/-B”对 Android 源代码的 libstagefright 模块进行分析. 通过对 Android 不同版本的源码进行静态分析, 对于未打补丁的源码可以找出以下高危漏洞, 其漏洞 CVE 编号分别如下: CVE-2015-1538, CVE-2016-6601, CVE2015-3832, CVE-2015-3831, CVE-2015-6604.

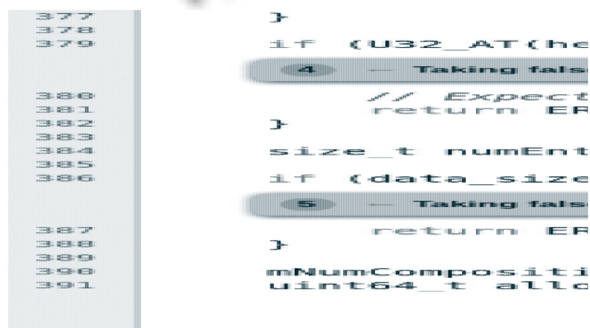

图 3 CVE-2015-1538 漏洞检测结果界面

### 5.4 分析过程的时间复杂度、空间复杂度及优化

本实验的分析工具在时间复杂度上耗时较大, 分析简单的 libmedia 模块需要十几分钟, 但是分析复杂的 libstagefright 模块就需要几十个小时. 通过使用 linux 下的 perf 性能调优工具对分析过程 (不能直接分析第 5.3 节中的 mmm 命令, 分析 mmm 函数解析后的命令) 进行分析, 发现耗时的 TOP-50 函数均是来自 libstp.so 和 libminisat.so 里的函数, 两个库文件里的函数总共占用了 95% 以上的运行时间. 其中 libstp.so 是分析工具所依赖的 STP 求解器的库文件, libminisat.so 是 libstp.so 所依赖的库文件. 运行时间主要耗费在求解表达式.

本实验的分析工具在空间复杂度上耗费也较大. 如分析 libmedia 模块需要 1–2 GB 内存, 分析 libstagefright 模块需要 2–4 GB 内存, 短时间会使用 16 GB 以上内存. 通过使用 linux 下的 gdb 和 valgrind 内存分析工具对分析过程进行分析, 主要发现 libstp.so 库文件与加入的 STP 求解器代码存在内存泄露问题, 其中存在少数没及时释放的大内存块, 导致了大量的内存占用.

针对时间复杂度和空间复杂度的问题, 对本实验的分析工具进行对应的优化, 在加入的 STP 求解器的代码部分, 加入了遗漏的释放资源的代码, 降低内存占用, 减少了部分内存泄露. 同时对 Checker 里的检测代码进行优化, 尽量用循环来取代递归, 同时将符号执行里的污点数据按照符号执行地址存放在公共 map 数据结构里, 进行加速, 以此节约内存和时间.

## 6 结语

本文提出了使用改进的 Clang 对 Android 源代码进行静态分析的研究和实现, 通过新增 Clang 的约束求解器并对静态检测功能进行改进, 实现了对 Android 部分类型漏洞的静态分析检测, 并成功检测到部分漏洞. 最终实现了可用的对 Android 源代码进行分析检测的实用静态分析工具.

但是分析工具自身还存在一些问题, 例如: 时间耗费较长、内存耗费较大, 经过第 5.4 节的优化, 解决了部分问题, 但是内存泄露问题没有彻底去除, 耗时仍旧偏长; 此外静态分析自身的符号执行存在一定偏差, 面对特别复杂的约束条件和调用过程, 不能准确地进行约束求解. 而且, 图 2 的“溢出判断”中的判断方法有些粗糙, 容易造成漏报, “溢出判断”需要对加减乘除等各种情况进行仔细地处理, 这些处理需要大量的细节进行完善.

## 参考文献


1 Dietz W, Li P, Regehr J, *et al*. Understanding integer overflow in C/C++. ACM Trans. on Software Engineering and Methodology (TOSEM), 2015, 25(1): 2.

2 Bush WR, Pincus JD, Sielaff DJ. A static analyzer for finding dynamic programming errors. Software-Practice & Experience, 2000, 30(7): 775–802.

3 Schmidt D, Steffen B. Program analysis as model checking of abstract interpretations. International Static Analysis Symposium. Pisa, Italy. 1998. 351–380.

4 Cadar C, Dunbar D, Engler D. KLEE: Unassisted and




© 中国科学院软件研究所 http://www.c-s-a.org.cn

automatic generation of high-coverage tests for complex systems programs. Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation. San Diego, California, USA. 2008. 209–224.

5 Kremenek T. Finding Software Bugs with the Clang Static Analyzer. California: Apple Inc, 2008.

6 Xu ZX, Kremenek T, Zhang J. A memory model for static analysis of C programs. International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Heraklion, Crete, Greece. 2010. 535–548.

7 Khedker U, Sanyal A, Sathe B. Data Flow Analysis: Theory and Oractice. Boca Raton, FL: CRC Press, 2009.

8 Xu ZB, Zhang J, Xu ZX. Melton: A practical and precise memory leak detection tool for C programs. Frontiers of Computer Science, 2015, 9(1): 34–54. [doi: 10.1007/s11704-014-3460-8]

9 Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Francisco, California, USA. 1995. 49–61.

10 Liu T, Huuck R. Case study: Static security analysis of the android goldfish kernel. International Symposium on Formal Methods. Oslo, Norway. 2015. 589–592.

11 Muntean P, Rahman M, Ibing A, *et al*. SMT-constrained symbolic execution engine for integer overflow detection in C code. Proc. of 2015 Information Security for South Africa (ISSA). Johannesburg, Azania. 2015. 1–8.

12 Wang TL, Wei T, Lin ZQ, *et al*. IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. Proc. of the Network and Distributed System Security Symposium. San Diego, California, USA. 2009.

13 Cadar C, Godefroid P, Khurshid S, *et al*. Symbolic execution for software testing in practice: Preliminary assessment. Proc. of the 33rd International Conference on Software Engineering. Honolulu, HI, USA. 2011. 1066–1071.

14 Sui YL, Yu D, Xue JL. Static memory leak detection using full-sparse value-flow analysis. Proc. of the 2012 International Symposium on Software Testing and Analysis. Minneapolis, MN, USA. 2012. 254–264.

15 Li L, Cifuentes C, Keynes N. Practical and effective symbolic analysis for buffer overflow detection. Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Santa Fe, New Mexico, USA. 2010. 317–326.

16 Sui YL, Xue JL. SVF: Interprocedural static value-flow analysis in LLVM. Proc. of the 25th International Conference on Compiler Construction. Barcelona, Spain. 2016. 265–266.

17 Enck W, Gilbert P, Han S, *et al*. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. on Computer Systems (TOCS), 2014, 32(2): 5.

18 Arzt S, Rastofher S, Fritz C, *et al*. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM SIGPLAN Notices, 2014, 49(6): 259–269. [doi: 10.1145/2666356]

19 Horváth G, Pataki N. Clang matchers for verified usage of the C++ standard template library. Annales Mathematicae et Informaticae, 2015, 44: 99–109.

20 Android 的全系统符号执行工具-Android_S2E. http://www.infoq.com/cn/presentations/whole-system-symbol-Implementation-tools-android-s2e. [2017-01-05].

21 Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. Proc. International Symposium on Code Generation and Optimization. San Jose, CA, USA. 2004. 75–86.

22 Spreitzenbarth M, Schreck T, Echtler F, *et al*. Mobile-sandbox: Combining static and dynamic analysis with machine-learning techniques. International Journal of Information Security, 2015, 14(2): 141–153. [doi: 10.1007/s10207-014-0250-0]

23 Zhang JX, Li ZJ, Zheng XC. PathWalker: A dynamic symbolic execution tool based on LLVM byte code instrumentation. International Symposium on Dependable Software Engineering: Theories, Tools, and Applications. Nanjing, China. 2015. 227–242.

24 Software Assurance Reference Dataset. https://samate.nist.gov/SRD/testsuite.php. [2016-11-10].



© 中国科学院软件研究所 http://www.c-s-a.org.cn