第 37 卷第 3 期
2020 年 3 月

计算机应用与软件
Computer Applications and Software

Vol. 37 No. 3
Mar. 2020

基于关键路径测试的安全补丁存在性检测

文 琪 江喆越 张 源
（复旦大学软件学院 上海 201203）

摘 要 漏洞的修复对于应用软件的安全至关重要。为了能够及时地修复所有已知漏洞，安全防护人员需要准确地检测一个安全补丁是否被应用。提出一个基于关键路径的语义层面的漏洞补丁存在性检测工具 PatchChecker，通过找寻一条在漏洞修复前后发生改变的路径，分析其语义特征，生成能代表漏洞补丁的签名信息；利用这一签名信息，在目标程序中找出对应路径进行比较，判断漏洞补丁的应用情况。PatchChecker 通过聚焦于单一路径，在提升对细节变化检测能力的同时，避免了未知代码修改带来的干扰。实验表明，PatchChecker 能够以较高的准确率检测漏洞补丁是否被应用。

关键词 漏洞 补丁 模糊测试 模拟执行

中图分类号 TP309.02 **文献标志码** A DOI：10.3969/j.issn.1000-386x.2020.03.001

SECURITY PATCH EXISTENCE DETECTION BASED ON CRITICAL PATH TESTING

Wen Qi Jiang Zheyue Zhang Yuan
(*Software School, Fudan University, Shanghai 201203, China*)

Abstract It is very important for the security of application software to fix the vulnerability. In order to fix all known vulnerabilities in time, security researchers need to accurately detect whether a security patch has been applied or not. This paper proposes a PatchChecker for vulnerability patch existence detection tool based on the critical path at the semantic level. Through finding a path that changed before and after the vulnerability repair, and analyzing its semantic characteristics, it would generate signature information that could represent the vulnerability patch. Then, we used this signature information to find out the corresponding path in the target program for comparison, and judged the application of vulnerability patches. By focusing on a single path, PatchChecker improved the ability to detect detail changes while avoiding interference caused by unknown code modifications. The experiments show that PatchChecker can detect whether the vulnerability patch is applied with high accuracy.

Keywords Vulnerability Patch Fuzz testing Simulation execution

0 引 言

近年来，新发现的安全漏洞数量快速增长[1]，给用户的安全带来了巨大的隐患。为了保障应用软件的安全，开发者需要及时对漏洞进行修复。而在开源日渐成为一种趋势的今天[3]，代码重用或者引用越来越多[4-5,9]，一个漏洞影响到的软件不再局限于其自身，还包括了许多引用到这些代码的软件，所有这些受影响的软件都需要对这个安全漏洞进行修复。当漏洞影响到的软件范围广泛、软件维护者分散时，如何保证所有受影响的软件都能及时地修复漏洞成为了一个巨大的挑战。也正是由于大量的软件并没有做到及时修复所有已知漏洞，使得无论是对于安全防护人员还是恶意攻击者而言，能够准确地检测安全漏洞是否被修复，都有着无法忽视的意义。

收稿日期：2019－05－07。国家自然科学基金项目（U1636204，61602123，U1836210，U1736208，U1836213，61602121）；国家重点基础研究发展计划项目（2015CB358800）；上海市青年科技英才扬帆计划项目（16YF1400800）。**文琪**，硕士生，主研领域：系统安全。**江喆越**，博士生。**张源**，副教授。

为了避免出现歧义,我们首先对漏洞补丁存在性检测的目标和范围给出一个明确的定义。漏洞补丁存在性检测的目标是,对于某一个开源库或软件的漏洞补丁,检查一个给定的目标程序是否被应用了该补丁,其中目标程序可能对原函数做了定制化等检测者未知的修改。同时,我们假设漏洞补丁的所有信息对检测者来说是已知的,且待检测目标程序必须含有该漏洞影响的函数,即我们不会去询问诸如“iOS12 的内核中是否应用了安卓内核上的某漏洞补丁”这样没有任何实际价值的问题。我们研究的目标专注于如何高效且准确地检测漏洞补丁的存在性,不会过多的关注漏洞修复与否会对程序造成多大的危害,即无论是一个可以造成远程代码执行的漏洞还是一个仅仅只能造成 DoS 攻击(denial-of-service attack)[2] 的漏洞,对我们的检测程序来说是没有区别的。

对于漏洞补丁存在性检测这样一个问题而言,检测结果的准确性十分重要,较低的准确率使得我们无法对工具的结果置信,只能对待测程序重新进行人工检查。而现有工作主要是利用函数相似度比较来找寻有漏洞的函数,尽管它们提到这些技术也可以用来判断一个函数中的漏洞是否被修复,但实际它们会以牺牲准确性为代价优先保证效率。FIBER[18] 作为以漏洞补丁存在性测试为目标的工具,虽然在准确性上有很大提升,但由于其签名包含代码结构信息,仍难以在检测的准确性和稳定性之间取得一个很好的平衡。

针对现有工具在漏洞补丁检测准确性上的缺陷,本文设计并实现了一个工具,弥补现有工具的不足,通过将关注点从整个函数转移到函数的局部,尽量摆脱函数补丁之外的代码改动带来的影响,在保证方法适用性的前提条件下,准确识别出一个函数是否应用了特定的漏洞补丁。

本文的主要贡献有:

- 指出漏洞补丁存在性检测技术的重要性以及当前相关领域研究匮乏的现状。
- 设计并实现了漏洞补丁存在性检测工具 PatchChecker,通过聚焦于函数的单条路径,利用漏洞补丁的语义特征,克服了之前工作在大范围测试中准确性低、难以在漏报率和误报率中取得平衡的问题,真正实现了高效、稳定、准确的检测效果。
- 对 PatchChecker 进行了全面的实验和评估,验证了其检测效果。

1 相关工作

目前的相关研究中,以漏洞补丁存在性检测为目标的工作寥寥无几,绝大多数的工作都是围绕函数相似度比较展开,指出可以利用函数相似度比较进行漏洞补丁存在性检测。

1.1 函数相似度比较

在源代码层面,CP-Miner[7] 利用数据挖掘技术检测出了众多由简单复制粘贴导致的漏洞。CCFinder[6] 将输入的源码信息转义成特征序列,改进了之前行对行的检测工作中所存在的缺陷。为了解决搜索空间和效率的问题,Deckard[8] 将源码转为树结构,通过子树相似度来寻找相似代码。ReDeBug[4] 给出了一个基于语法的独特设计,虽然找到的代码克隆相对较少,但检测速度高、范围广,并减少了误报率。VulPecker[9] 构建了一个包含不同漏洞及相应特征的集合,针对不同漏洞选取相应的相似性算法。

在二进制程序层面,相关工作无法依赖于变量名、变量类型等基础信息,于是往往会选择基于代码的结构[10-12] 来实现。比如 BinDiff[13] 依赖于控制流图的同构来分析判断,而 BinSlayer[14] 则进一步将问题转换成二分图匹配问题。discovRE[10] 则是提取基本代码块中的数值特征信息,利用梯度下降计算代码块间的距离,最后通过解决最大公共子图同构的问题来计算图之间的距离。改进方案如 Genius[11] 和 Gemin[12] 则从控制流图中提取特征并转换成多维向量进行检测。而在基于语义的研究工作方面,文献[15]利用基本代码块的 I/O 作为特征进行匹配,文献[16-17]则利用符号执行和定理证明器对具有相同语义的基本代码块进行了形式上的证明。

由于漏洞补丁普遍对程序本身的改动十分有限,一般漏洞的修复往往通过几行代码的修改就能实现,这些改动比程序在版本迭代过程中带来的功能性改动要小得多。在这种情况下,这些基于整个函数的相似度比较的方法,很可能会被这些功能性改动影响到程序判断的结果,无法达到一个较高的准确率。

1.2 漏洞补丁存在性检测

FIBER 是当前唯一专门用于漏洞补丁存在性检测的工具,其可用性和准确性相比于函数相似度比较的方式都得到了一个质的提升。FIBER 首先会在漏洞补丁中选取最为合适的修改位置作为代表,依据这些选取出来的修改,在样本二进制程序中生成一个包含尽量多的源代码信息的签名,最后利用这个生成的签名在待测二进制程序中进行匹配。FIBER 是一个半语义化的工具,其生成的签名同时包含了漏洞补丁的部分语义信息和部分结构信息。

FIBER 将其关注点完全放在特定的几行被漏洞补

丁改动的代码上,同时为了保证签名的唯一性,额外增加了一些与漏洞补丁本身无关的代码作为上下文进入签名之中。但这样的处理方式,在保证唯一性的同时,也使得签名变得更加不稳定,如果这些与漏洞补丁无关的代码在功能迭代或定制化的过程中发生了改变,FIBER 的检测很可能因此而产生错误。

2 系统架构

如图 1 所示,本文将 PatchChecker 的工作流程划分为三个模块:预处理模块、输入生成模块和检测模块。

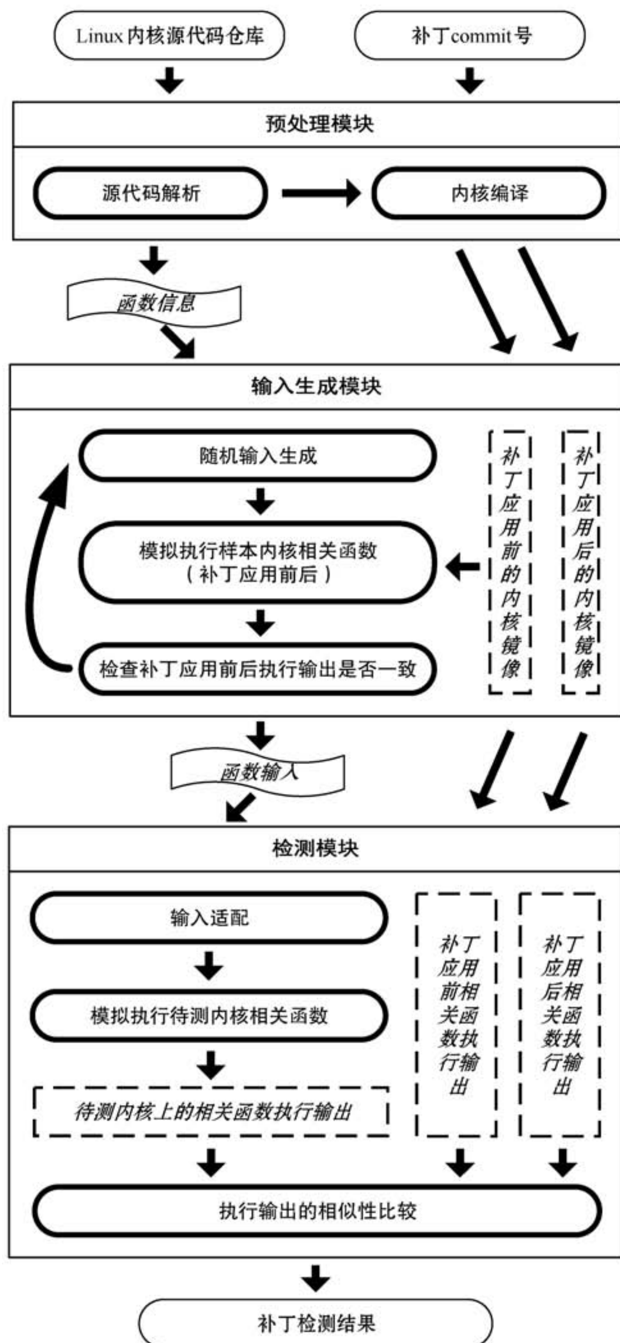

图 1 PatchChecker 的整体模块划分

2.1 预处理模块

预处理模块是 PatchChecker 执行开始的部分,除了基本的漏洞补丁文件解析外,其最主要的工作是编译生成后面模块需要用的内核镜像样本。

PatchChecker 会从 Linux 内核的源代码仓库中提取出漏洞修复前后的两份除了漏洞补丁外完全相同的源代码,采用完全相同的编译器与编译选项,基于内核的默认配置进行编译。为了使后续的分析能够正常进行,这里需要确认漏洞影响到的函数在生成的内核镜像样本中存在。对于不存在的情况,主要是两种原因导致,一是漏洞影响到的函数所在的内核子模块默认未开启,编译时这些代码直接被忽视,我们需要根据内核的 Makefile 文件,找到相应的控制选项并开启后重新编译;二是漏洞补丁修改的函数被编译器进行了内联处理,这种情况只能尝试在降低编译优化等级的情况下重新编译,但即使如此,函数仍可能被内联。

通过多次的迭代重新编译,我们能对大部分情况生成可用的内核镜像样本。对个别函数始终无法找到的情况,如果这个漏洞补丁影响到的其他函数中有能在内核样本中找到的,我们可以认为这个内核样本是可用的,因为一个漏洞的修复一般来说是原子操作,不存在一个函数被修复,而另一个函数没有被修复的情况。如果漏洞影响到的所有函数都始终无法找到,那么 PatchChecker 暂时无法对此漏洞进行检测。

2.2 输入生成模块

输入生成模块的设计目标是希望找到一组能够代表漏洞补丁语义的输入。对一个函数而言,一组输入对应着一条特定的执行路径,路径上的信息代表着这次执行带来的影响。以 CVE-2016-7117 为例,图 2 是其官方补丁的代码(因篇幅原因省略了对注释的修改),可以发现其对代码的修改难以用简单的特征准确代表,但仔细观察补丁内容,当代码的约束条件为 err != 0 && datagrams != 0 && err != -EAGAIN 时,赋值操作 ock -> sk -> sk_{err} = -err 和函数调用 fput_light(sock -> file, fput_needed)的执行顺序在漏洞修复前后发生了改变。而这个漏洞的本质正是在调用 fput_light 之后,sock 指针指向的结构体可能已经被 free,因此触发了 UAF(Use After Free)漏洞。而经过官方修补后,这个赋值操作先于 fput_light,因此赋值时 sock 指针指向的结构体不可能已经被 free 了。所以,在这

个例子中,用这条关键路径上这两个操作的顺序作为签名来代表这个漏洞补丁的语义非常准确。

```
-out_put:
-        fput_light(sock->file, fput_needed);
-
         if (err == 0)
-                return datagrams;
+                goto out_put;

-        if (datagrams != 0) {
+        if (datagrams == 0) {
+                datagrams = err;
+                goto out_put;
+        }
+
+        if (err != -EAGAIN) {
-                if (err != -EAGAIN) {
-                        sock->sk->sk_err = -err;
-                }
-
-                return datagrams;
+                sock->sk->sk_err = -err;
         }
+out_put:
+        fput_light(sock->file, fput_needed);

-        return err;
+        return datagrams;
```

图 2 漏洞 CVE-2016-7117 官方补丁

为了找到这样能够代表漏洞语义信息的输入,PatchChecker 采用了以模糊测试为主、静态分析为辅的方法。模糊测试技术往往被用于漏洞发现,利用自动或半自动生成的随机数据去触发漏洞。在这个场景中,PatchChecker 的目标相比于触发漏洞要容易许多,往往只要执行到了漏洞补丁改动的代码部分,路径上的信息就很可能在漏洞修复前后的样本中表现出不一致,足以反映这个漏洞补丁的语义。

2.3 检测模块

检测模块是 PatchChecker 工作流程中的最后一环。在该模块中,对于给定的漏洞补丁,PatchChecker 将在参考内核镜像样本中生成的输入交给待测内核执行,获取其输出,通过将得到的输出分别与该输入在漏洞补丁应用前后的内核镜像上运行得到的输出进行对比,给出判断结果。如果发现待测内核镜像上运行得到的输出与两个样本上运行得到的输出的相似度一致,那么 PatchChecker 将无法判断待测目标的修补情况。一种常见的例子是待测内核增加或修改了部分关键的判断条件,导致生成的输入在待测内核上运行时,走到了一条与漏洞补丁修改的代码无关的路径上。对于这种情况,我们可以生成多种不同的满足要求的输入,只要有一组输入执行时能走到期望的路径上,就可以作出正确的判断。

3 设计与实现

3.1 执行引擎

执行引擎是 PatchChecker 的核心,在输入生成模块和检测模块中,执行引擎都扮演着举足轻重的角色,它负责将给定的程序输入转换成其在指定的内核上执行后产生的输出。由于在本文场景中,所有的执行过程都是单独执行一个函数,在未设置好上下文的情况下,通过插桩的手段去实际运行单个函数极易出现异常。所以我们选择使用模拟执行的方式来运行,这样可以更为简单地建立一个可用的运行环境,并对可能出现的异常进行恰当的处理。同时,考虑到执行引擎的运行效率也是衡量工具的关键因素,我们选择使用具体的数值进行模拟执行,以避免符号化执行可能引起的效率问题。

综合考虑,我们选取了 Unicorn Engine[19] 作为执行引擎的核心。Unicorn Engine 基于 QEMU[20] 实现,是一个轻量级、跨平台、跨架构的 CPU 模拟器框架,其能够同时支持 Arm、Arm64(Armv8)、M68K、Mips、Sparc 和 X86(包括 X86_64),这为 PatchChecker 的跨架构支持提供了基础保障。Unicorn Engine 由纯 C 代码实现,并依靠实时编译技术提供了较高的运行效率。

图 3 简单描绘了执行引擎的工作流程,执行引擎首先会将整个内核镜像加载到内核地址空间中,然后按照输入的要求,初始化其余的内存空间和寄存器。在初始化过程中,由于是利用实值进行模拟执行,而交给执行引擎的输入是带有部分符号信息的,故执行引擎首先需要进行去符号化。由于输入中的符号,实际是由内存空间的划分导致的,执行引擎首先会选择一块未使用的空间,接着按照输入中列出的内存区域的编号顺序,依次为每块内存区域切分一片固定大小的内存,例如 1 MB。当内存区域分配好后,只要将输入中的所有内存区域编号和偏移的二元组用该区域地址加偏移的结果替换即可消除符号。鉴于内存的分配是按照输入中内存空间的编号依次有序进行的,故只要输入内容不变,去符号化的结果也会保持不变,这确保了程序模拟执行输出的稳定,有利于之后对输出的比较。输入转换完后,执行引擎需要利用 Unicorn Engine 的钩子机制,对所有的内存操作进行挂

钩,以记录发生过改变的内存,减少不必要的噪声和分析开销。

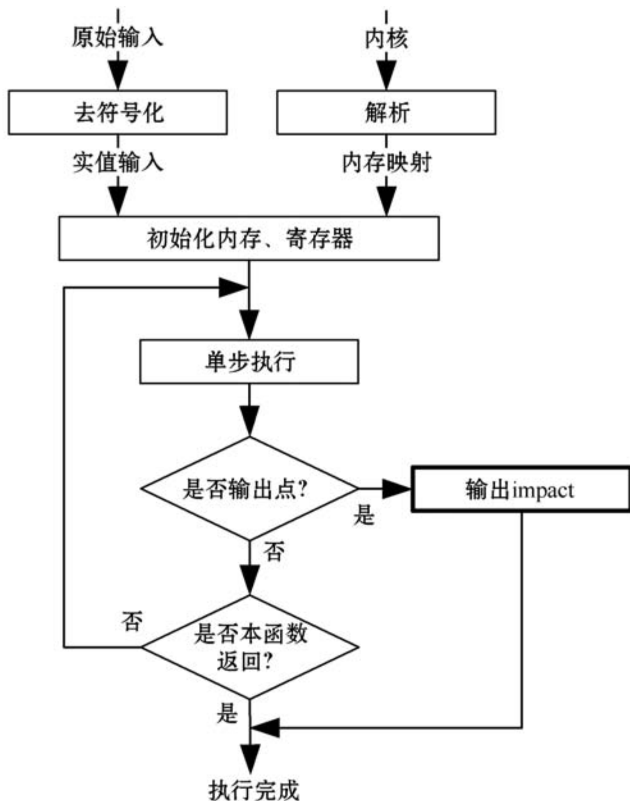

图 3　执行引擎的工作流程

所有初始状态设置完成后,执行引擎将从目标函数的开头开始单步执行。对于每一条指令,执行引擎会判断其是否是一个输出点,对于当前 PatchChecker 的实现,我们只选取了所有的函数调用及函数返回作为输出点。对于所有的输出点,执行引擎将记录下程序执行到该位置时,相比于初始状态发生改变的内存地址及其值、当前调用函数的地址或名称。针对函数调用,由于函数原型解析的复杂性,暂时只会记录第一个参数。对于函数返回,则会记录下相应的函数返回值。在一个输出点记录下来的所有这些信息称作一个 impact,当执行结束时,执行引擎按顺序记录下了一系列的 impact,即得到了一个 impact 的序列,这个序列可以视为执行引擎针对本次输入执行得到的输出。

3.2　模糊测试引擎

由于本文目标在于找到一组能够在漏洞修复前后的两个内核样本中运行产生不同效果的带符号的输入,现有的模糊测试工具难以满足这一需求,因此我们实现了一个简单的模糊测试引擎。图 4 描绘了这个模糊测试引擎的工作流程。

首先,根据预处理模块解析出的函数原型,我们对

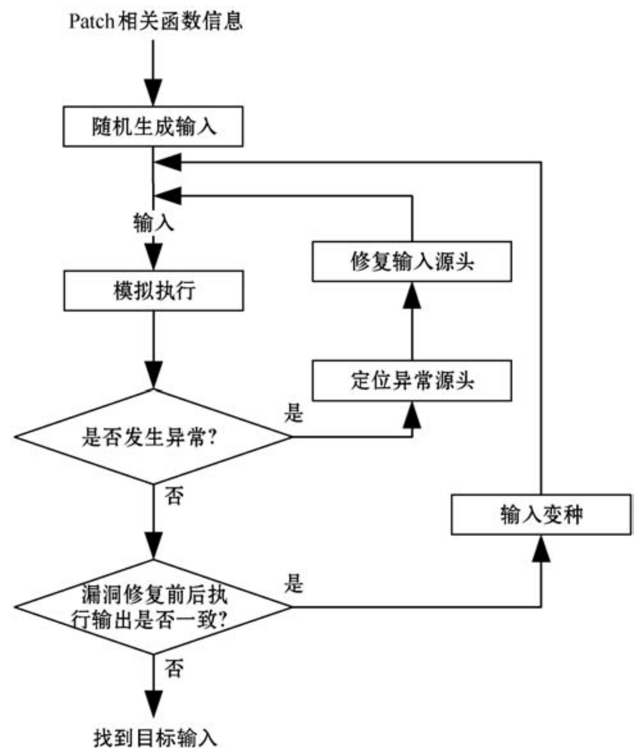

图 4　模糊测试的工作流程

函数参数进行随机的赋值。为了提高模糊测试的效率,这里我们对随机赋值的可选范围进行了限制,其中基础的数值包括 0、0xffffffff、2^n(n 为任意值),而指针则包括任意一块已经存在内存区域的基地址或者新分配的一块内存区域的基地址。通过该限制条件,我们在缩小随机范围的情况下,尽可能覆盖了包括但不限于空值、长度、符号位、指针在内的各种数据类型,使我们有能力遍历整个函数中足够多的分支。

由于初始随机产生的输入没有经过任何校验,故极有可能是非法输入。当这样一个非法输入交由执行引擎执行时,很可能会在执行中途因为各种异常而无法继续,尤其是内存访问异常最为常见。当模糊测试引擎捕获到这样的异常时,将采取符号执行或数据流分析的方式,确定导致这个异常的输入数据源头,直接对相应的输入点进行矫正。然后将修改后的输入重新交由执行引擎执行,如果执行的过程中又出现了新的异常,将采用相同的方式继续尝试修复输入,通过这样不断地迭代,便能得到可以完成整个函数运行的合法输入。

对于修复后的合法输入,将对其在漏洞修复前后的两个内核样本中执行产生的输出(impact 序列)进行对比。当这两个输出完全一致时,该输入是不满足需求的,将以其为种子进行变种。每次变种,我们都会在当前种子输入的函数参数、当前种子输入在样本内核上运行时访问到的内存空间、外部函数调用的返回值中随机选取一个作为本次的修改点,在赋值范围限制

内对其重新进行随机赋值。多次迭代变种之后,会找到满足要求的输入,即能在漏洞修复前后的两个内核样本中执行时产生不同的输出。

虽然这个模糊测试引擎的实现仍在初步阶段,且对输入变种的范围进行了较大的限制,但由于需要找到的输入条件并不苛刻,所以仍可以达到较好的效果。在之后的实验中,这个简单的模糊测试引擎已能全自动地完成大部分漏洞补丁的输入生成工作。

4 实验设计

本文选择 ARM32 作为实验中可执行程序的架构,在实验中采用的机器 CPU 是 64 核的 Intel® Xeon® CPU E7-4830,主频为 2.13 GHz,内存为 64 GB。

4.1 输入生成模块

从 CVE Details[21] 上 2013 年 – 2016 年的 Linux 内核漏洞中,选择了 12 个漏洞作为测试集合,在选取时尽量覆盖了不同类型、不同 CVSS[22] 分数、不同影响力的漏洞进行测试,以验证 PatchChecker 的适用性。

表 1 展示了在选出的 12 个 CVE 上,生成满足要求的输入所需的时间,其中小括号内的代表输入生成失败及失败用时。可以发现,12 个 CVE 对应的 20 个函数中,只有两个函数没有能够成功生成满足我们要求的输入。由于这两个函数所在 CVE 均有多个影响到的函数,而其他函数成功生成了满足要求的输入,因此即使其生成输入失败,也不会影响 PatchChecker 对其所在 CVE 的检测。

表 1 PatchChecker 输入生成运行时间

CVE 号	函数名	生成用时/s
CVE-2013-7446	unix_create1	138
	unix_dgram_poll	114
	unix_dgram_sendmsg	3 946
	unix_release_sock	820
CVE-2014-9529	key_gc_unused_keys	235
CVE-2015-2686	sys_recvfrom	102
	sys_sendto	(198)
CVE-2015-3288	handle_pte_fault	152
CVE-2015-3636	ping_unhash	17
	ping_v4_unhash	20
CVE-2015-5706	path_openat	1 417
CVE-2016-10200	l2tp_ip_bind	46
CVE-2016-10229	udp_recvmsg	12
	udpv6_recvmsg	13
CVE-2016-4470	key_reject_and_link	387
CVE-2016-7117	__sys_recvmmsg	48
CVE-2016-7910	disk_seqf_stop	13
CVE-2016-9120	ion_free	3
	ion_handle_get_by_id	2
	ion_ioctl	(121)

对于运行效率,由于函数复杂度不同,不同函数的输入生成用时差异巨大。但由于输入的生成可以离线完成,故这些生成用时均在可以接受范围内。

4.2 检测模块

对于检测模块,使用与输入生成测试中相同的 CVE 漏洞进行测试,函数输入使用输入生成测试过程得到的结果,对于生成失败的两个函数,通过人工介入,分别生成了一组满足要求的输入。而对于待测内核,则是在 Linux 内核代码仓库中选取了 100 个不同的提交,每个提交均采用 4 种不同的编译选项进行编译,得到 400 个待测内核,这些内核的漏洞补丁实际修复情况可以直接根据代码提交记录准确得知。提交的选取方式大致如下:首先根据选出的漏洞,找到其影响的文件,然后在内核代码仓库中,找出所有对这些被影响到的文件有改动的提交,从这些提交中,在保证时间跨度的前提下随机选取。这样的选取方式使得选出的提交更容易对漏洞影响的代码有功能上的改变,可以在一定程度上测试 PatchChecker 在不同版本代码中的稳定性。

表 2 展示了 PatchChecker 对 12 个漏洞的检测情况。由于 PatchChecker 设计的目的是为了能够通过自动化的检测为用户提供多一层的安全检查,防止用户暴露在已知漏洞的威胁之下,因此最需要关注的还是错误的比例。我们希望 PatchChecker 能够做到不产生误判,因为每一个误判都会降低这个工具的可信度。根据表 2 中的数据,在大部分的 CVE 上,PatchChecker 实现了很好的效果,只有 CVE-2013-7446 的 unix_create1 和 CVE-2015-5706 的 path_openat 错误率相对较高。对于 CVE-2013-7446 而言,由于其有多个函数可以用来进行检测,因此综合这些函数的判定结果后,还是可以实现很低的错误率。

表 2　PatchChecker 在待测 CVE 各个函数上的测试结果

CVE 号	函数名	错误率/%
CVE-2013-7446	unix_create1	18
	unix_dgram_poll	0
	unix_dgram_sendmsg	3
	unix_release_sock	0
CVE-2014-9529	key_gc_unused_keys	0
CVE-2015-2686	sys_recvfrom	0
	sys_sendto	0
CVE-2015-3288	handle_pte_fault	0
CVE-2015-3636	ping_unhash	0
	ping_v4_unhash	0
CVE-2015-5706	path_openat	65.5
CVE-2016-10200	l2tp_ip_bind	0
CVE-2016-10229	udp_recvmsg	4.5
	udpv6_recvmsg	4.5
CVE-2016-4470	key_reject_and_link	0
CVE-2016-7117	__sys_recvmmsg	3
CVE-2016-7910	disk_seqf_stop	0
CVE-2016-9120	ion_free	0
	ion_handle_get_by_id	0
	ion_ioctl	0

5　结　语

本文对漏洞补丁存在性检测这样一个新兴的问题进行了深入的研究,为了能够解决现有工具在效率、准确率等各方面的不足和所面临的问题,设计并实现了一个准确高效的全自动漏洞补丁存在性检测工具 PatchChecker。PatchChecker 聚焦于函数的单条路径,提取漏洞补丁的语义信息进行检查。通过对 12 个真实漏洞在 400 个 Linux 内核镜像上进行测试,验证了 PatchChecker 能够给出一个准确的检测结果。

参 考 文 献

[1] CVE: Vulnerabilities by year[EB/OL]. [2018-08-01]. https://www.cvedetails.com/browse-by-date.php.

[2] McDowell M. Understanding denial-of-service attacks[EB]. National Cyber Alert System,Cyber Security Tip ST04-015. 2004,2004.

[3] Github Annual Report[EB/OL].[2018-08-01]. https://octoverse.github.com.

[4] Jang J, Agrawal A, Brumley D. ReDeBug: finding unpatched code clones in entire os distributions[C]//2012 IEEE Symposium on Security and Privacy. IEEE, 2012: 48-62.

[5] Kim S, Woo S, Lee H, et al. VUDDY: A scalable approach for vulnerable code clone discovery[C]//2017 IEEE Symposium on Security and Privacy(SP). IEEE,2017:595-614.

[6] Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code[J]. IEEE Transactions on Software Engineering, 2002, 28(7): 654-670.

[7] Li Z, Lu S, Myagmar S, et al. CP-Miner: Finding copy-paste and related bugs in large-scale software code[J]. IEEE Transactions on software Engineering, 2006, 32(3): 176-192.

[8] Jiang L, Misherghi G, Su Z, et al. Deckard: Scalable and accurate tree-based detection of code clones[C]//Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007: 96-105.

[9] Li Z, Zou D, Xu S, et al. VulPecker: an automated vulnerability detection system based on code similarity analysis [C]//Proceedings of the 32nd Annual Conference on Computer Security Applications. ACM, 2016: 201-213.

[10] Khoo W M, Mycroft A, Anderson R. Rendezvous: A search engine for binary code[C]//Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013: 329-338.

[11] Feng Q, Zhou R, Xu C, et al. Scalable graph-based bug search for firmware images[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016: 480-491.

[12] Xu X, Liu C, Feng Q, et al. Neural network-based graph embedding for cross-platform binary code similarity detection [C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017: 363-376.

[13] BinDiff[EB/OL]. [2018-08-01]. https://www.zynamics.com/bindiff.html.

[14] Bourquin M, King A, Robbins E. Binslayer: accurate comparison of binary executables[C]//Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop. ACM, 2013: 4.

[15] Pewny J, Garmany B, Gawlik R, et al. Cross-architecture bug search in binary executables[C]//2015 IEEE Symposium on Security and Privacy. IEEE, 2015: 709-724.

[16] Gao D, Reiter M K, Song D. Binhunt: Automatically finding semantic differences in binary programs[C]//International Conference on Information and Communications Security. Springer, Berlin, Heidelberg, 2008: 238-255.

(下转第 12 页)

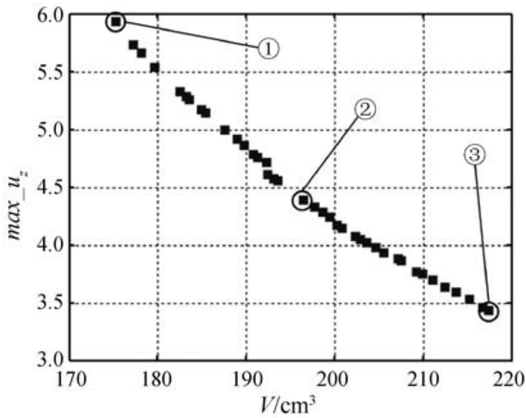

图 9 MAV 机身多目标设计 Pareto 解集

由于多目标设计优化的结果为 Pareto 解集[8],设计人员可以根据实际情况和需求在 Pareto 解集中选择设计方案[9]。表 7 列出了图 9 中标注的三个典型设计点处的设计变量和目标值,点 1 和点 3 分别表示注重最小化体积和最小化最大变形,点 2 代表权衡最小化体积和最小化最大变形。

表 7 三个典型设计点处的设计变量和目标值

点	设计变量	体积	z 方向最大变形
1	[47.11 85.03 41.30 41.30 252.00 6.27]	175.26	5.94
2	[48.87 85.10 41.89 40.02 250.52 7.12]	196.49	4.39
3	[48.62 85.20 41.33 40.00 250.00 8.00]	217.36	3.43

以本文使用的遗传算法为例,需要几千次的建模和仿真计算才能得到该多目标设计优化问题的 Pareto 解集。采用本文提出的设计优化框架,设计人员只需要进行一次建模和仿真,其他的重复工作可以全部由计算机通过集成三种软件完成,显然计算机采用参数化的方式进行建模和仿真,比人工进行效率更高。

3 结 语

本文充分利用三种软件的优势,提出一种 SolidWorks-MATLAB-ANSYS 集成框架,实现了参数化建模/仿真和优化计算,由计算机代替设计人员完成大量的重复建模、仿真工作,有效提高了设计效率。本文的方法被成功应用于一种微型飞行器机身的结构设计优化中。

此外,进化算法(如遗传算法、粒子群算法等)因具有高鲁棒性和广泛适用性,被广泛应用于全局优化中。但其需要大量的函数调用,即便采用参数化建模/仿真和优化计算,也需要大量的时间,尤其对于复杂产品的设计优化,其计算成本难以接受。代理模型技术是解决这一问题的有效手段,下一步工作将研究将代理模型技术与本文提出的 SolidWorks-MATLAB-ANSYS 集成框架相结合,进一步提高结构设计优化的效率。

参 考 文 献

[1] Park H S, Dang X P. Structural optimization based on CAD-CAE integration and metamodeling techniques[J]. Computer-Aided Design, 2010, 42(10): 889 - 902.

[2] 宋宏伟,刘浩. 基于 MATLAB 与 ANSYS 的结构优化设计[J]. 大连民族大学学报,2011,13(3):284 - 287.

[3] 方芳,黄松和,林刚. 基于 MatLab 和 SolidWorks 的凸轮轮廓设计及性能分析[J]. 矿山机械,2010(6):39 - 42.

[4] 常凯. 一种渐开线齿轮啮合特性分析的参数化求解方法研究[J]. 机械工程师,2017(8):94 - 96.

[5] 马东辉,赵东. 基于 ANSYS 和 MATLAB 的结构优化设计[J]. 制造业自动化,2013(19):106 - 108.

[6] 郑帅,柴晓艳,刘锡军,等. 基于 SolidWorks 和 Ansys Workbench 的钢管输送机构关键部件的优化设计[J]. 起重运输机械,2016(6):7 - 11.

[7] Nguyen J, Park S, Rosen D. Heuristic optimization method for cellular structure design of light weight components[J]. International Journal of Precision Engineering and Manufacturing, 2013, 14(6): 1071 - 1078.

[8] Shan S. An efficient pareto set identification approach for multiobjective optimization on BlackBox functions[J]. Journal of Mechanical Design, 2005, 127(5):279 - 291.

[9] Deb K, Pratap A, Agarwal S, et al. A fast and elitist multiobjective genetic algorithm: NSGA-II[J]. IEEE Transactions on Evolutionary Computation, 2002, 6(2):182 - 197.

[10] 舒乐时,周奇,蒋平,等. 基于序贯 Kriging 模型的潜器型线优化设计[J]. 船舶工程,2016(9):43 - 46.

(上接第 7 页)

[17] Ming J, Pan M, Gao D. iBinHunt: Binary hunting with inter-procedural control flow[C]//International Conference on Information Security and Cryptology. Springer, Berlin, Heidelberg, 2012: 92 - 109.

[18] Zhang H, Qian Z. Precise and accurate patch presence test for binaries[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 887 - 902.

[19] Unicorn Engine[EB/OL]. [2018 - 08 - 01]. https://www.unicorn-engine.org.

[20] Bartholomew D. Qemu: a multihost, multitarget emulator[J]. Linux Journal, 2006, 2006(145): 3.

[21] CveDetails[EB/OL]. [2018 - 08 - 01]. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html.

[22] CVSS[EB/OL]. [2018 - 08 - 01]. https://www.first.org/cvss/specification-document.