

基于原子混淆的通用侧信道漏洞修补方法

邹德清^{1,2,3,4}, 张盼^{1,2,3,4}, 刘伟⁵, 陈维杰^{1,2,3,4}, 陆弈帆^{1,2,3,4}

(1. 大数据技术与系统国家地方联合工程研究中心, 湖北 武汉 430074; 2. 服务计算技术与系统教育部重点实验室, 湖北 武汉 430074;
3. 大数据安全湖北省工程研究中心, 湖北 武汉 430074; 4. 华中科技大学网络空间安全学院, 湖北 武汉 430074;
5. 北京京航计算通讯研究所, 100089 北京)

摘 要: 包含侧信道漏洞的代码在程序被执行时会表现出与输入有关的非功能性行为, 攻击者利用微架构的侧信道攻击可获取这些行为, 并通过分析行为与输入之间的关联模式恢复应用数据内容, 达到窃取用户机密数据的目的。软件层的侧信道漏洞修补方法带给程序的性能损耗较低, 并且因为无须修改硬件或系统, 可实现快速修补和大范围部署, 成为密码算法实现采用的主流策略。现有修补方案与程序的具体实现深度绑定, 需要人工介入, 存在实现难度大、不通用的问题。针对以上问题, 提出了一种结合动态混淆技术和硬件原子事务特性的通用侧信道漏洞修补方法。所提方法向侧信道漏洞代码中插入动态混淆访存操作, 以达到隐藏真实访存地址的目的, 并将漏洞代码和混淆访存封装为硬件原子事务, 保障被封装的代码在运行时连续执行而不会被中断, 避免攻击者利用细粒度的侧信道攻击区分真实的访存操作和混淆的访存操作。基于 LLVM 编译器实现了原型系统 SC-Patcher, 引入了包括安全跳板和原子事务聚合等在内的多种设计, 提高了方法的安全性和实用性。安全与性能测试结果表明, 使用所提方法完成侧信道漏洞修补的程序, 在几乎未增加额外性能开销的同时, 能够有效抵抗侧信道攻击, 防范攻击者从漏洞处还原有效的机密数据内容。

关键词: 侧信道防御; 漏洞修补; 原子事务; 混淆执行

中图分类号: TP393

文献标志码: A

DOI: 10.11959/j.issn.2096-109x.2022014

Universal patching method for side-channel vulnerabilities based on atomic obfuscation

ZOU Deqing^{1,2,3,4}, ZHANG Pan^{1,2,3,4}, LIU Wei⁵, CHEN Weijie^{1,2,3,4}, LU Yifan^{1,2,3,4}

1. National Engineering Research Center for Big Data Technology and System, Wuhan 430074, China

2. Services Computing Technology and System Lab, Wuhan 430074, China

3. Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Wuhan 430074, China

4. Huazhong University of Science and Technology, Wuhan 430074, China

5. Beijing Jinghang Research Institute of Computing and Communication, Beijing 100089, China

Abstract: Executing code containing side-channel vulnerabilities exhibits different non-functional behaviors related

收稿日期: 2021-10-26; 修回日期: 2021-11-22

通信作者: 刘伟, micliu126@126.com

基金项目: 国家自然科学基金 (62172168)

Foundation Item: The National Natural Science Foundation of China (62172168)

引用格式: 邹德清, 张盼, 刘伟, 等. 基于原子混淆的通用侧信道漏洞修补方法[J]. 网络与信息安全学报, 2022, 8(2): 100-111.

Citation Format: ZOU D Q, ZHANG P, LIU W, et al. Universal patching method for side-channel vulnerabilities based on atomic obfuscation[J]. Chinese Journal of Network and Information Security, 2022, 8(2): 100-111.

to inputs. Attackers can obtain these behaviors by leveraging micro architecture side-channel attacks and then analyze the pattern between the behaviors and the inputs to access sensitive data. Vulnerability repairing at the software layer brings low overheads to a program's execution. Besides, it does not require modifying hardware or system, which enables fast patching and widespread deployment. It becomes the mainstream strategy applied to the current cryptographic implementations. However, existing solutions are deeply bound to the program's implementation and requires manual intervention. This brings challenge to implement and is not versatile enough. A general patching method was proposed for side-channel vulnerabilities that combined dynamic obfuscated execution with hardware atomic transaction. To hide the real accesses of the side-channel vulnerabilities of a program, the proposed method inserted dynamic confusing accesses into the vulnerabilities. To avoid an attacker using fine-grained side-channel attack to distinguish the real access and the confusing access, both of them were encapsulated as transactions and they were guaranteed to be uninterrupted during the running period. In addition, a prototype system called SC-Patcher was implemented based on the LLVM compiler. Various optimization strategies were supported, including secure springboard and transaction aggregation, to further improve system security and performance. Experimental results show that the proposed method makes it impossible for an attacker to restore accurate sensitive data through side-channel attack, and it also brings almost no additional performance overhead to the program.

Keywords: side-channel defense, vulnerability repair, atomic transaction, obfuscated execution

0 引言

微架构侧信道在硬件架构中非常普遍，这些侧信道产生的一个主要原因是硬件组件被系统内的不同进程所共享，特别是硬件在设计中基于局部性原理引入的各种缓存组件，如缓存、页表、页表缓存和分支目标缓存等。在执行程序过程中，数据或代码是否被暂存到这些组件中会表现出比较明显的响应时间差异。由于这些组件被系统内的进程共享，攻击者可利用多种方式^[1-11]影响进程对缓存组件的使用，然后通过测量时间差披露受害程序的运行轨迹。

含有微架构侧信道漏洞（以下简称漏洞）的程序在运行时会表现出与机密输入有关的非功能性行为。例如，不同机密输入的变化会影响基于控制流泄露漏洞选择不同执行路径，或影响基于数据访问泄露的漏洞访问不同目标地址。攻击者可在获取程序运行轨迹后，还原程序行为，通过分析输入与漏洞代码行为间的关系，还原用户机密数据内容。

常见的侧信道防御策略包括异常检测和行为混淆，学术界提出了许多硬件层的^[12-15]和系统层的^[16-24]的解决方法。这些方法都存在性能或兼容性方面的限制，未得到广泛使用。开发人员在实际中倾向于采用细粒度的靶向防御，即在软件层

对代码实施侧信道漏洞检测^[25-30]，然后修补程序中的侧信道漏洞，这类方法通常只会给程序带来很小的性能损耗，并且因为无须修改硬件或系统，可实现侧信道漏洞的快速修补以及大范围部署，被主流的加密实现采用。

基于软件的漏洞修补可分为行为一致^[31]和噪声混淆^[2]。基于行为一致的策略实现难度较大，首先需要对代码进行准确分析，然后对路径平衡补全^[31]。这种策略在面对复杂程序时很难保障行为的完全一致，特别是循环嵌套。因此多数情况下，开发者需要手工完成修补，这就对开发者的专业素养要求很高，此外在程序代码量特别大时，漏洞分析和修补非常容易出现偏差。噪声混淆的策略虽然容易实现自动化，但简单的噪声混淆容易被攻击者通过预先构建好的统计学模型移除^[32]或者细粒度的侧信道攻击绕过^[2,33]，并不能达到好的防御效果。

本文提出了一种全自动的基于原子混淆的通用侧信道漏洞修补方法。该方法的核心思想是结合硬件原子事务技术和噪声混淆技术，在漏洞代码中增加用于混淆的访存操作，然后把漏洞代码和混淆操作代码封装为原子事务，并在程序运行时监控事务执行状态，保障原子事务内的代码在结束前连续执行未被中断，否则立即中止执行并回退。混淆访存使得攻击者在攻击过程中获取到

的轨迹除了真实目标访存外,还包含非常多的用于混淆的噪声访存;原子事务保障攻击者无法通过细粒度的侧信道攻击去辨别真实的访存和噪声访存。完成漏洞修补后的程序,攻击者即便能够搜集到程序运行轨迹,也只能观察到事务内一段连续执行的代码对硬件组件状态的改变,这些包含大量噪声的状态信息难以被用于还原用户的机密信息。

本文基于 LLVM 编译器实现了自动修补方法的原型系统 SC-Patcher,并在实现中采用了多种安全和性能方面的优化策略。首先,SC-Patcher 针对不同类型的漏洞设计了不同的混淆策略,在保障混淆的安全性的同时,尽可能减少额外访存指令数量以降低对程序性能的影响。其次,在封装原子事务时,SC-Patcher 采取事务聚合的优化操作将多个离散代码(包括漏洞代码、混淆操作和正常代码)封装到同一事务,在提高性能的同时增加事务内噪声的多样性。通过真实的侧信道攻击验证 SC-Patcher 修补后的密码算法的安全性,结果表明修补后的程序能够很好地抵御攻击者窃取机密数据的行为。通过 SC-Patcher 对修补后的 OpenSSL 加密算法进行性能测试,结果表明 SC-Patcher 的漏洞补丁给程序带来的性能损耗(3.11%)几乎可以忽略不计。与现有的侧信道防御方法 T-SGX^[24]的结果进行对比,SC-Patcher 在安全性和性能方面有显著提高。

本文贡献包括:

- 1) 提出了一种基于原子混淆的通用方案能够实现侧信道漏洞的自动修补;
- 2) 实现了原型系统 SC-Patcher,并给出了许多安全和性能方面的优化设计方案;
- 3) 使用 SC-Patcher 对真实的程序进行漏洞修补,对修补后的程序进行安全和性能测试,证明了修补方法的安全性和实用性。

1 背景知识

1.1 侧信道攻击

微架构的侧信道攻击指的是攻击者通过观察硬件组件的状态信息窃取目标程序的私密内容的行为。在侧信道攻击中,只有目标程序的机密输入 $\mathcal{S}$ 、程序运行时表现的行为 $\mathcal{P}$ (如分支选择或

者数据访问)和可被观察到的组件状态信息 $\mathcal{I}$ 之间存在关联关系: $\mathcal{I} \sim \mathcal{P} \sim \mathcal{S}$,攻击者才能通过使用观察到的状态信息 $\mathcal{I}$ 还原出机密输入 $\mathcal{S}$ 。其中 $\mathcal{I} \sim \mathcal{P}$ 表示状态信息 $\mathcal{I}$ 与程序行为 $\mathcal{P}$ 之间存在关联关系。系统会泄露程序行为的状态信息,如基于处理器的缓存^[3-6]、页表^[8-9]、页表缓存^[7]和分支目标缓存^[2]。可以通过产生各类中断来减少攻击时的噪声,提高还原信息的准确度^[8-9]。 $\mathcal{P} \sim \mathcal{S}$ 表示程序行为 $\mathcal{P}$ 和机密输入 $\mathcal{S}$ 之间存在关联关系,即机密输入的变化会导致程序的行为发生改变,而攻击者可以通过获取程序的行为推测出机密输入的内容。常见的关联关系有:分支选择依赖,即程序控制流选择的目标分支依赖机密输入;数据访问依赖,程序内数据访问的目的地址依赖机密输入。通常程序中存在分支选择依赖和数据访问依赖关系的代码被分别视为程序的基于控制依赖的侧信道漏洞和基于数据访问依赖的侧信道漏洞^[31,34]。

1.2 侧信道防御策略与检测技术

从部署层上对侧信道的防御策略进行分类,可分为硬件架构层、系统层和软件层的防御策略。

硬件架构层的防御策略,最直接的方法是分隔共享的资源,如处理器的缓存分隔^[11,13]和页表缓存分隔^[15]。另外一种方法是随机化资源的使用,混淆侧信道的观察^[12]。

系统层的侧信道防御策略包括:① 硬件组件分割,把组件的不同部分分配给不同的进程,如页染色^[16]或者页锁定^[17];② 刷新系统状态以降低信息的泄露风险^[18];③ 降低系统测量的准确值,如修改虚拟机的 rdtsc 指令,添加随机值^[19-21];④ 检测程序的异常行为,如通过计数器判断程序的运行是否过慢^[22],或者检测程序运行过程中产生的中断是否过多^[23-24]。

软件层的侧信道防御策略主要有 3 类:一致化、随机混淆和漏洞检测。一致化指的是,开发者通过修改程序或者算法,使得攻击者在机密输入变化的情况下搜集到的程序轨迹是一样的。随机混淆是在程序中添加噪声来隐藏真实的程序轨迹,一般是在与程序机密数据相关的访问中添加额外的混淆访存,或者对代码或者数据进行随机重排。漏洞检测主要分为静态和动态方法。静态

检测方法常用抽象表征去分析源码并测量泄露的上确界。CacheAudit^[25]通过马尔可夫链量化程序泄露信息的上限。还有很多工具用于验证程序是否有依赖机密数据的条件跳转和内存访问^[26-28]。动态检测方法是根据程序运行时的真实轨迹去判断侧信道漏洞。CacheD^[29]检查每条指令在不同的机密输入下是否会访问不同的缓存地址。Shin等^[30]运用 K-means 聚合算法去检查缓存行为和机密输入的关联关系。MicroWalk^[30]采用相互信息测量相同的关系。侧信道漏洞检测只负责发现和定位漏洞，而侧信道漏洞自动修补方面的工作相对较少，通常需要专家的人工介入，对漏洞进行分析与重新设计算法，专业性很强，门槛很高。文献[31]通过分析程序，平衡分支路径实现侧信道漏洞修补，该方法一方面并不支持复杂逻辑程序，另一方面攻击者仍可以通过细粒度的侧信道攻击分辨真实的轨迹和混淆的轨迹^[33]。

硬件层和系统层的防御策略很少被应用于真实的平台或者场景中，主要原因是这些方案存在性能或兼容性方面的问题，难以被部署到实际的生产环境中。相对而言，软件层的防御策略则更加实用，具体为：这类策略能够快速修复漏洞并完成大范围的部署；另外，软件层的漏洞修复大多不会给程序带来额外的性能开销。

1.3 事务内存

事务内存的设计目的是简化并行编程模型，降低多核处理器上获取互斥锁的开销，提高并行的线程对共享内存操作的效率。事务内存的特性包含原子性、一致性和隔离性。原子性指的是事务内的操作作为一个整体是不可被分割的，要么事务内的所有读写提交全部被系统接受，要么全部被系统抛弃；一致性指的是事务执行的前后，系统都必须处于一致性状态；隔离性指的是事务在提交之前，内部操作对象的状态对外界不可见，不会影响系统的状态。如果一个事务在没有冲突的情况下执行结束，其所有的读写会被提交到内存中，否则所有因为读写而保存的中间状态将被抛弃（不会被提交到真实的内存中），同时系统会将程序的控制权交给进入事务区前注册的一个异常处理句柄去决定是否重新执行事务区。在运行事务时，硬件需要把事务内所有读数据和写数据

的内容暂存到一个缓冲区中，然后在事务运行结束前，再把缓存的内容提交到内存中。Intel TSX (transactional synchronization extensions) 将 L1 缓存作为缓冲区。这种设计无须在硬件上增加新的事务缓存硬件，可直接复用现有的缓存一致性协议检测事务间的访问冲突，能够极大地简化硬件的设计。但是这种设计导致 TSX 需要对事务内的读写集施加严格的限制，即 L1 数据缓存必须能够容纳事务的所有内存的写操作，否则会导致事务的退出。

RTM (restricted transactional memory) 是 Intel TSX 的硬件事务内存解决方案提供的接口。CPU 的指令集中一共有 4 条与 RTM 相关的指令，其中 XBEGIN 用于初始化事务；XEND 用于结束事务的执行；XABORT 用于中断事务的执行；XTEST 用于测试代码是否在事务中执行。前 SC-Patcher 的实现采用 RTM 指令对代码进行原子事务封装，它同样适应于其他厂商的硬件事务内存解决方案，如 ARM 的 TME (transactional memory extension)，AMD 的 ASF (advanced synchronization facility)。

2 威胁假设模型

假设开发人员已通过各种手段，发现并标识出应用程序代码中的侧信道漏洞。本文方法的目标是修补程序中的这些侧信道漏洞，阻止攻击者利用这些侧信道漏洞窃取用户的私密信息。在普通场景中，攻击来自系统内非 root 权限的普通用户，不能直接访问被攻击程序的内容。在可信执行环境（如 Intel SGX）的场景中，攻击可以来自于恶意的操作系统，攻击者可以调度所有的进程，利用各种页级、缓存级、指令级的侧信道攻击，探测系统状态，但不能直接观察目标程序的内存。

不考虑类似能量分析^[32]、电磁辐射^[35]等物理级的侧信道攻击，这些攻击通常需要攻击者近距离地去搜集物理机的各类信号。瞬态执行攻击，如 Meltdown^[36]和 Spectre^[37]攻击及其变种^[38]不在防御范围内，这些攻击与程序中是否存在微架构侧信道漏洞并无关系，并且属于硬件漏洞。

3 漏洞修补方法

本节首先简单介绍修补方法的工作流程，然后详细介绍工作流程中涉及各个模块。

侧信道漏洞修补的工作流程如图 1 所示，含有编译和运行两个阶段，整个过程涉及多个模块的参与。编译阶段包含代码标识模块、混淆插桩模块和原子事务封装模块；运行阶段包含动态混淆模块和异常检测模块。在编译阶段，开发者提供程序源码和侧信道漏洞位置信息给一个修改后的编译器。代码标识模块处于编译器前端，基于漏洞位置信息标识应用代码，并将这些标识信息

作为元数据传递给编译器后端。混淆插桩模块和原子事务封装模块根据代码标识信息，以基本块为最小单元对漏洞代码做进一步处理：混淆插桩模块在编译时向漏洞代码插入额外的混淆访存指令，以达到隐藏程序真实运行轨迹的目的；原子事务封装模块封装漏洞代码和混淆代码为原子事务，保障代码在执行中不可被中断。在运行阶段，动态混淆模块与异常检测模块被加载到程序进程空间中，动态混淆模块在程序执行时不断变换混淆访存目标地址，增加混淆噪声的不确定性，异常检测模块则监控原子事务在执行过程中未被恶意中断，确保代码是以原子性运行的。

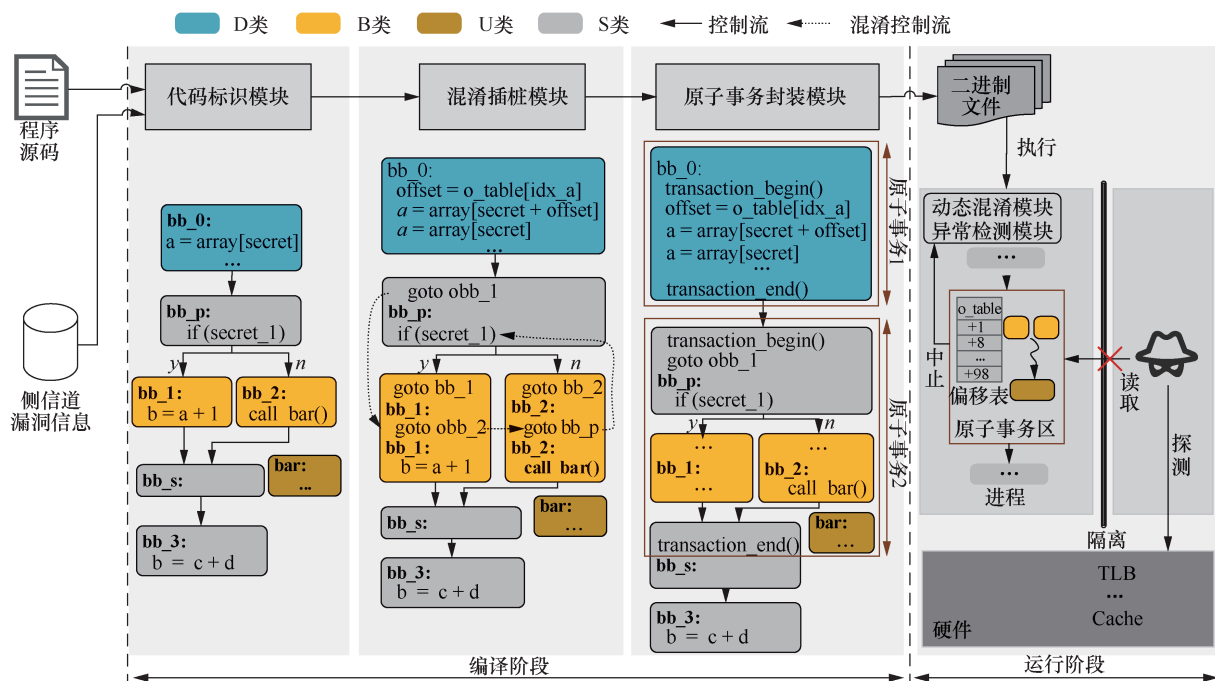


图 1 侧信道漏洞修补的工作流程
Figure 1 The workflow of patching side-channel vulnerabilities

图 1 展示了代码经各个模块处理后的变化。其中基本块 bb_0 使用机密数据 secret 作为数组内数据访问的索引，存在基于数据访问的侧信道漏洞；基本块 bb_p 使用机密数据 secret_1 作为分支判断条件，执行基本块 bb_1 或基本块 bb_2 会导致机密数据 secret_1 值的泄露，存在基于控制依赖的侧信道漏洞。

3.1 代码标识模块

代码标识模块根据用户提供的侧信道漏洞信息标识应用的源码。漏洞信息包括：文件

名、漏洞行和侧信道漏洞类型。表 1 展示了模块用于标识代码的 4 种类型，为了简化称呼，分别称这些代码为 D (data) 类代码、B (branch) 类代码、U (unsafe) 类代码和 S (safe) 类代码。需要特别注意的是 U 类代码，这类代码为 B 类或 U 类代码中被调用的函数。暴露这类代码的执行信息会导致调用函数执行信息的泄露。假如一段代码中同时包含数据访问依赖和控制选择依赖的漏洞，则被同时标记为 D 类和 B 类。

表1 SC-Patcher 中的代码类型标识
Table 1 Code Type Identification in SC-Patcher

类型	分类标识的原因
D	包含数据访问依赖的漏洞代码
B	包含控制选择依赖的漏洞代码
U	会导致其他类型的代码的访问信息泄露
S	安全代码, 不会产生机密数据泄露

图1采用不同颜色对D、B、U和S类代码进行区分,如基本块bb_0使用机密数据作为访问数组的索引值,存在数据访问依赖的侧信道漏洞,被标记为D类代码(蓝色基本块);分支在选择基本块bb_1和bb_2时依赖机密数据的值,因此包含控制选择依赖的侧信道漏洞,被标记为B类代码(黄色基本块);函数bar在基本块bb_2内被调用,bar的泄露会导致基本块bb_2的执行信息泄露,被标记为U类代码(棕黄色基本块)。

3.2 混淆插桩模块

混淆插桩模块负责向漏洞代码中插入混淆访存指令,达到在运行时隐藏漏洞代码的真实运行轨迹的目的。模块根据漏洞代码的类型不同,采取了不同混淆策略。

针对D类包含数据访问依赖漏洞的代码,混淆插桩模块在原始访存操作的前后增加多个用于混淆的内存读写操作。数量过多的混淆访存指令会给程序带来比较明显的性能损耗,而过少的数据混淆访存又无法提供充足的安全保障,为解决这一矛盾,该模块引入动态随机访存机制以增强混淆噪声,在提高安全性的同时降低了混淆访存的数量。图1中基本块bb_0的变化展示了混淆插桩模块增加数据访问混淆的效果:模块在array[secret]附近插入额外访存操作,访存目标地址基于混淆偏移表o_table对应表项的偏移值动态计算得到。

针对B类代码,即代码中包含控制选择依赖的侧信道漏洞,混淆插桩模块采取串联预访问的混淆策略:使用直接跳转指令,在执行真实代码前,穿插预先访问区域内的基本块,达到隐藏真实代码访问的目的。不同分支执行的指令数量存在不同,从而导致分支执行所耗费的时间存在差异。为降低因时间产生的信息泄露,混淆模块也在B类代码的不同分支中添加额外的访存指令,

平衡不同分支的指令数量,分支执行时间尽可能相似。如图1中混淆插桩模块对基本块bb_p、基本块bb_1、基本块bb_2的处理,展示了插桩代码如何实现混淆执行:控制流在进入基本块bb_p分支选择前,先跳转到基本bb_1,然后到基本块bb_2,最后执行条件选择进入真正的基本块执行,这样会强制加载其他基本块的内容到缓存中,从而达到混淆的目的。

3.3 原子事务封装模块

原子事务封装模块将漏洞代码和混淆代码封装为原子事务。针对不同分类标识代码,该模块所采用的封装策略存在差异。

对于只包含D类标识的代码,即存在数据访问依赖的侧信道漏洞的代码,原子事务封装模块将漏洞所在基本块封装为原子事务。图1中transaction_begin函数表示原子事务的开始,transaction_end函数表示原子事务的结束。

模块对于B类代码的处理,是期望避免代码的执行信息被泄露,但代码的前驱节点和代码对后继节点的分支选择都会暴露执行信息。因此原子事务封装模块会将基本块的前驱结点以及前驱结点的所有子基本块添加到原子区内,同时通过一个跳板代码完成状态切换并跳转到后继节点。图1的原子事务封装模块在基本块bb_p首指令处插入transaction_begin开始原子事务,并在基本块bb_s的入口处的transaction_end结束原子事务。这样基本块bb_p和bb_s间的基本块(包括函数bar的所有基本块)执行都处于同一原子事务区中。

当代码同时包含B类和D类标识时,原子事务封装模块采用B类代码的封装策略。U类代码的处理逻辑同B类代码,但在封装的实现细节上存在稍许差异。

3.4 动态混淆模块

动态混淆模块在程序运行时,通过随机改变混淆偏移表内的偏移值,实现混淆指令的动态地址访存,达到增强混淆噪声强度、提高安全性的目的。混淆访存目标地址是通过原目标地址值加上对应偏移表表项的偏移值计算所得。通常偏移表的表项值为-4 096~4 096的任意数,这个范围覆盖了缓存级到页级的地址。一方面能够适应多

种粒度的侧信道攻击；另一方面计算得到的混淆访存内容的地址都比较靠近目标访存，使得混淆地址和实际地址的相关性比较强，从逻辑上讲更具有迷惑性。动态混淆模块在记录原子事务执行次数的同时会随机生成一个阈值与所记录次数进行比较，当原子事务的执行次数大于阈值时，模块重置混淆偏移表，并使用随机值再次填充，最后清零事务执行次数并重新生成阈值后，继续程序执行。

3.5 异常检测模块

异常检测模块负责监控原子事务内代码的运行状态，判断事务在执行过程中是否存在被攻击的异常行为。若被封装的代码的原子性在执行过程中被破坏，硬件会立即中断事务的执行并回滚到事务初始状态，然后跳转到异常检测模块进行攻击判定。异常检测模块根据硬件交易事务退出时返回的状态码，决定是否停止代码的执行。运行过程中事务退出的原因有多种，其中包含许多正常的系统行为。例如，现代操作系统以固定时间间隔发送时钟中断，达到多任务分时轮转的目的，被封装为原子事务的代码在执行过程中很有可能因当前进程轮询的时间片结束，收到时钟中断信号，最终导致事务的中断。另外一种情况是，在代码执行过程中，目标执行代码、访存的代码页不在内存中，而产生缺页异常情况，导致交易事务区中的代码执行中断。事务内的代码在异常退出时，检测模块可读取硬件返回的退出状态。异常检测模块为每个异常设定可接受数量的阈值，然后统计导致事务退出的各个异常发生的次数，当次数超出阈值时，异常检测模块会判定程序存在被攻击的可能而立即停止代码的执行。

4 SC-Patcher 实现与优化细节

基于上述的侧信道漏洞修补方法，本文在 LLVM 11.0 的基础上实现了原型系统 SC-Patcher，其支持对大部分 C 和 C++ 的程序进行漏洞修补的处理。

原型系统的主要部分包括事务混淆插桩模块和原子事务封装模块，被整合在 LLVM 编译器的后端；代码标识模块作为 LLVM 编译器的一个前端插件，将每个导出函数的函数包装器注入

LLVM 中间表示中；动态混淆模块和异常检测模块作为静态链接库，在编译时被自动链接到程序中，整个原型系统的核心实现部分约 4 200 行代码。

本节将介绍 SC-Patcher 中原子事务分割、原子事务聚合、安全跳板、敏感函数的封装、寄存器优化的实现细节。

4.1 原子事务分割

SC-Patcher 需要分割存在缓存使用冲突的原子事务，这是因为在运行 Intel TSX 的事务时，原子事务内的所有写操作内容必须保留在 L1 数据缓存中，直到事务结束后才会真正地提交到内存中。任何导致暂存数据被驱逐出缓存的行为，都会导致事务执行的中止。因此若一个事务内的所有写操作不能暂存到 L1 数据缓存，这个事务将永远不会被成功执行。而 L1 数据缓存受硬件实现的影响，其大小有限，以 Skylake 处理器为例，L1 数据缓存为 8 路组相联，单个缓存行为 64 byte，缓存大小为 32 kB。对于单个缓存行大小的任意数据访问，处理器都会将该缓存行的所有内容加载到 8 路 L1 缓存组的任意一个槽。假如存在 8 个以上的不同缓存行的写操作内容映射到同一缓存组，会导致 L1 数据缓存驱逐事务的写内容，事务将立即执行失败。SC-Patcher 将缓存冲突作为事务切割的依据，尽可能降低缓存行的冲突，提高事务执行的成功率。

SC-Patcher 移植 T-SGX^[24] 内一个用于分析代码中的缓存使用量的静态分析组件，在编译时对漏洞和混淆代码进行缓存冲突分析。由于编译器在编译时不能获取程序在运行时的确切访存地址，因此该组件实现采取了保守的过近似分析算法以估算代码中最糟糕的缓存冲突情况：对于可获取准确地址的对象，分析组件，直接将其分配给对应的缓存组，并且相应路数加 1；对于访问对象的地址是未知寄存器以及固定偏移的组合（如 `rbp+16`），分析组件根据基指针分组计算，分别估计新增的缓存使用路数，并把这些路数累加到所有的缓存组；对于地址完全未知的对象，分析组件，将该对象分配给所有缓存组。事务区内缓存的使用是上述 3 种情况累加的上确值。组件会在分析结束后根据结果分割代码。

4.2 原子事务聚合

SC-Patcher 尝试在不产生缓存冲突的前提下, 将多个离散的原子事务聚合到同一原子事务中。该聚合处理提升了程序的安全性和执行效率: 从安全方面讲, 原子事务内执行的指令越多, 累积的硬件状态的改变越多, 攻击者攻击时的噪声越大, 攻击越困难; 从性能方面讲, 执行一个空的 Intel TSX 事务 (只执行 XBEGIN 和 XEND 指令), 会耗费大约 200 时钟周期, 因此减少频繁的事务创建和结束, 可提升程序性能。SC-Patcher 首先分析各原子事务内的缓存使用, 然后通过动态规划, 在不产生缓存冲突的情况下, 尽可能地合并事务。此外, 事务间通过安全代码完成控制流的转移, 因此 SC-Patcher 在进行缓存分析和事务封装时, 需要考虑这些安全代码。

4.3 安全跳板

SC-Patcher 使用安全跳板实现原子事务间的安全切换。事务的开始指令 (XBEGIN) 在执行前和结束指令 (XEND) 在执行结束并不在事务的保护中, 假如把原子事务封装的开始指令 (XBEGIN) 和结束指令 (XEND) 直接嵌在基本块中, 攻击者可以在程序运行时探测到两个相邻事务间非保护指令的执行, 而导致受保护的基本块的执行信息泄露。如图 2 所示, 控制流在基本块 `bb_leaky_1` 执行结束后跳到基本块 `bb_leaky_2`, 两个基本块被分割到不同事务, 如果将 XBEGIN 指令直接嵌在基本块 `bb_leaky_2` 的入口处, 攻击者通过不断地探测这条指令的地址, 可推测出基本块 `bb_leaky_2` 的执行状态。为避免信息泄露, SC-Patcher 在安全跳板中完成原子事务的状态切换: 基本块首先跳转到安全跳板中, 完成原子事务切换, 然后跳转到目标地址 (如图 2 中黑色箭头指向的程序控制流)。通过安全跳板的防护策略, 攻击者仅可以获取安全跳板内的公共指令地址的访问信息, 而原子事务内的代码的执行信息仍受到保护。

4.4 敏感函数的封装

SC-Patcher 设计了专门的指令片段用于进入和退出 U 类敏感函数。原因是无泄漏的 S 类代码需要在普通状态下调用 U 类敏感函数, 而受保护的 B 类代码需要在原子事务状态下进入 U 类敏感

函数。图 3 包含上述两种不同类型的函数调用, 其中被标记为 S 类的安全基本块 `bb_safe` 调用了 U 类敏感函数 `bar`, 包含漏洞的基本块 `bb_leaky` 为 B 类代码同样调用了函数 `bar`。

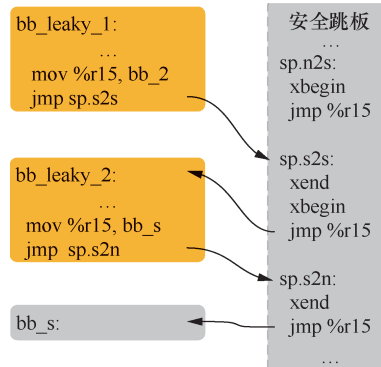


图 2 安全跳板
Figure 2 Secure springboard

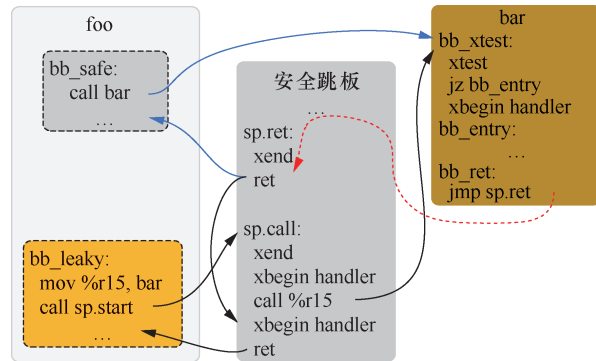


图 3 临界状态的控制流
Figure 3 Critical control flow

基本块 `bb_leaky` 在调用函数 `bar` 的开始和结尾处运行了两次原子事务的状态切换, 是因为静态分析函数间的缓存冲突几乎不太现实, 因此 SC-Patcher 通过事务切换消除潜在的缓存冲突。为避免攻击者通过探测原子事务封装指令的地址泄露被调用的函数执行信息, SC-Patcher 在处理受保护目标函数调用时 (如基本块 `bb_leaky` 调用函数 `bar`), 首先通过安全跳板内的一个中间片段 `sp.call` 完成原子事务状态切换, 然后跳转到目标函数 `bar` 内继续执行代码。此外, 函数 `bar` 在执行结束后, 首先通过一个直接跳转指令转到安全跳板中的指令片段 `sp.ret` 完成原子事务切换后, 然后回到基本块 `bb_leaky` 中。

基本块 `bb_safe` 在普通状态下调用函数 `bar`, 由于函数 `bar` 被封装为原子事务, 因此 CPU 在进

入 `bar` 函数前, 需要先开启事务, 否则函数会在执行中, 由于不处于事务状态下执行 `XEND` 指令而产生异常。函数 `bar` 返回时 (基本块 `bb_safe`), 需要退出原子事务状态恢复到普通模式后, 才可继续执行 `bb_safe`。因此, `SC-Patcher` 在每个 U 类敏感函数的入口函数前插入事务判断片段, 若发现 U 类函数不处于事务状态, 便立即开启一个事务。

4.5 寄存器优化

`SC-Patcher` 修改了编译器后端对寄存器的分配策略, 简化了安全跳板代码的实现。`Intel TSX` 会在事务异常中止时将中止状态码 (`abort status code`) 保存至 `rax` 寄存器中, 该操作会覆盖 `rax` 寄存器的原始值。一种解决方法是在弹簧跳板中插入相关指令, 在事务执行前保存 `rax` 的值, 在事务执行时再还原 `rax` 的值, 而这种设计需要分析哪些代码使用了 `rax` 寄存器, 并在跳转代码中插入额外的指令, 处理比较烦琐。`SC-Patcher` 通过修改编译器的寄存器分配策略, 在泄露的函数中设置 `rax` 为保留寄存器, 限制 `rax` 寄存器只被用于基本块内的传值, 而基本块间的数据流则不会使用 `rax` 寄存器。该优化操作省去了复杂的寄存器分析步骤, 简化了设计。

5 安全性分析与测试

本节对 `SC-Patcher` 实现的安全性进行分析, 并通过真实的侧信道攻击进行安全测试验证。

5.1 安全性分析

本文提出的侧信道漏洞修补方法的核心策略为混淆代码和原子事务。混淆代码是在编译时在侧信道漏洞的相关代码中插入额外的混淆访存指令, 用于隐藏漏洞代码的执行轨迹。本文方法针对控制流选择依赖的漏洞和数据访问依赖的漏洞分别采用了串联预访问和动态随机混淆访存的策略。对于攻击者而言, 在程序运行时, 硬件状态由漏洞的真实访存和混淆噪声访存共同作用, 收集到的状态信息包含大量噪声, 难以被攻击者用于还原程序的准确运行轨迹, 进而无法被用于恢复用户的私密数据。攻击者可以利用细粒度的侧信道攻击识别运行过程中的真实操作和混淆操作从而达到绕过混淆策略的目的, 但这类方法通常需要利用中断操作, 暂停程序的执行以降低噪声。

而本文方法基于硬件特性的原子事务把漏洞代码和混淆指令封装到原子事务中, 在运行时, 通过异常检测模块监控事务状态, 保障原子事务内的代码在执行未结束前不被中断, 从而阻断攻击者实施细粒度的侧信道攻击途径。通过双重安全保障, 本文实现了漏洞代码的修补, 保护程序敏感信息不被泄露。

原子事务的临界代码, 如事务的进入和退出, 会导致部分原子事务内执行的状态信息泄露, 因此 `SC-Patcher` 内所有与原子事务状态相关的切换操作均在安全跳板中完成后, 再跳转回目标地址。这种设计保障了临界代码的安全边际, 使得攻击者即便能够获取跳转代码的地址, 也不能通过这类公共地址推测出有用的执行信息。

`SC-Patcher` 对基于时间的侧信道攻击防御存在不足。尽管 `SC-Patcher` 在插入混淆指令时, 会根据不同分支所包含的指令数量, 尽可能地实现分支平衡。但对于复杂程序 (如循环和函数调用) 而言, 其很难使不同分支的执行时间完全一致。不过由于基于时间的攻击精度并不高, 特别在 `SC-Patcher` 保护的程序中, 攻击者只能测量各个原子事务内所包含的多段代码执行耗费的粗粒度的时间信息, 因此可推测到的内容极其有限。在后续工作中, `SC-Patcher` 可通过在安全跳板中增加包含随机访存次数的混淆函数, 并在执行过程中动态调用这些函数, 达到动态变换原子事务执行时间的目的。

5.2 安全性测试

本次测试的目标是评估攻击者能否通过 `Prime + Probe` 攻击窃取使用 `SC-Patcher` 修补后的 `AES` 程序中的加密密钥。

(1) 攻击工具

`Mastik`^[39] 是一个实现侧信道攻击和分析的工具集, 已提供包括 `Prime + Probe` 和 `Flush + Reload` 在内的多种侧信道攻击。

(2) 安全性测试

使用 `SC-Patcher` 对 `AES` 加密算法中关于 `S-box` 和 `T-box` 的访问操作进行修补。使用修补后的程序加密明文, 并在运行时, 利用 `Mastik` 进行 `Prime + Probe` 的攻击, 重复 10 000 次, 并分析预测密钥。实验中增加了同样使用事务交易特性

进行侧信道保护的 T-SGX^[24]作为对比测试对象。测试中修改了 T-SGX 的代码,从而支持在非 SGX 环境中执行。

(3) 测试原子事务的有效性

攻击测试中观察到的现象同 5.1 节的分析一致,当 Prime + Probe 攻击采样发生在原子事务内时,SC-Patcher 立即检测到攻击并中止程序执行,攻击将无法继续,因此恢复密钥失败。T-SGX 的交易事务同样能准确识别攻击。

(4) 测试动态混淆的有效性

实验假设攻击者改良了攻击的采样步骤,使得攻击可以成功绕过 SC-Patcher 和 T-SGX 事务区代码的检查(测试时通过在原子事务开始前和结束后添加钩子代码以达到绕过的目的)。实验中增加静态混淆的测试作为参照组,即在程序运行过程中保持动态混淆的偏移表表项内容不变。

表 2 展示了针对不同防御方法保护下的 AES 程序,利用 Prime + Probe 攻击所恢复的密钥比特位的准确率结果。攻击者在精准避开事务区边界的情况下,可以准确恢复 T-SGX 保护的程序的密钥内容。该结果直接说明单纯基于原子事务的防御策略并不能为程序提供完善的安全防护。在 SC-Patcher 的测试结果中,攻击者仍可以从静态混淆策略的防护中恢复 78.6%密钥内容,防护效果并不是非常理想;而在动态混淆的防护加持下,由于攻击者获取的状态信息中包含大量动态混淆访问操作,密钥还原的准确率仅达到 7.1%。因此从实验测试的分析结果可得,SC-Patcher 的动态混淆的修补策略是安全有效的。

表 2 还原 AES 密钥的准确率
Table 2 The accuracy of restoring AES keys

测试内容	准确率
原生 AES	100.00%
T-SGX	100.00%
SC-Patcher	静态混淆 78.60%
	动态混淆 7.10%

6 性能测试

本节测试原型系统 SC-Patcher 的侧信道漏洞修补给应用带来的额外性能开销。

(1) 实验环境

所有测试运行在 1 台 x86 架构的工作站上。硬件配置包括: Intel(R) Core(TM) i9-9900K CPU (代号 Coffee Lake, 8 核心 16 线程, 32 kB 的 L1 数据缓存, 32 kB 的 L1 代码缓存, 1 MB 的 L2 缓存, 16 MB 的 L3 缓存, 8 路组相联), 128 GB DDR4 内存(频率 3 000 MHz), 1 TB Samsung EVO 980 SSD。

(2) 测试程序与编译选项

测试的目标程序为 OpenSSL, 版本号为 1.1.0f。编译时均采用 -O3 等级的优化,并使用 -lto 开启链接时的优化选项。

(3) 漏洞检测

使用基于差分的漏洞检测工具 DATA^[34]检测 OpenSSL 中包含的 DES、AES 和 SHA-1 签名算法进行侧信道漏洞,然后通过人工审核筛选出共 29 个侧信道漏洞。

(4) 性能测试

SC-Patcher 编译测试程序的源码,并根据被提交的漏洞信息对生成的应用自动完成漏洞修补。输入随机生成的密钥,分别对 DES、AES 和 SHA-1 签名算法进行性能测试,重复执行每种加密算法 10 000 次,记录每次执行的平均时间。

表 3 展示了漏洞修补对加密算法执行效率的影响。从上述结果可得,漏洞修补给 OpenSSL 的 3 种算法带来了约 3.11%(几何平均)的性能损耗。Shinde 等^[8]的工作在人工优化后,仍然给程序执行带来几十倍性能开销,T-SGX^[24]将全部代码封装在事务区中,也带来了大约 50%的性能开销,而 SC-Patcher 的自动修补方法带给程序的性能损耗小于 5%,这个结果完全可以接受。因此从性能方面而言,本文漏洞修补方法非常实用。此外,漏洞修补使得编译后的二进制文件大小增加了 0.38%,改变很小。

表 3 加密算法的性能损耗
Table 3 The performance overhead of encryption

测试内容	算法			二进制文件大小
	DES	AES	SHA-1	
未漏洞修补	8.591 ms	6.360 ms	11.618 ms	3.138MB
漏洞修补后	8.719 ms	6.508 ms	12.261 ms	3.150MB
开销	1.42%	2.32%	5.53%	0.38%

7 结束语

本文提出了一种自动化的通用侧信道漏洞修补方法,创新性地结合混淆执行和硬件原子事务,在使用噪声混淆访存操作隐藏真实访存地址的同时,采用原子事务避免攻击者利用细粒度的侧信道攻击绕过混淆保护。安全分析与性能测试的结果表明,修补后的漏洞在不泄露用户机密数据内容的同时,带给程序的额外性能开销可以几乎忽略不计,测试结果均优于现有的工作。接下来的工作将拓展该方法的实现到 ARM 或 RISC-V 平台,并进一步提高兼容性和通用性。

参考文献:

- [1] ANDRYSCO M, KOHLBRENNER D, MOWERY K, et al. On subnormal floating point and abnormal timing[C]//Proceedings of IEEE Symposium on Security and Privacy(S&P'15). 2015: 623-639.
- [2] LEE S, SHIH MW, GERA P, et al. Inferring fine-grained control flow inside SGX enclaves with branch shadowing[C]//Proceedings of 26th USENIX Security Symposium (USENIX Security'17). 2017: 557-574.
- [3] LIU F, YAROM Y, GE Q, et al. Last-level cache side-channel attacks are practical[C]//Proceedings of IEEE Symposium on Security and Privacy(S&P'15). 2015: 605-622.
- [4] GRUSS D, SPREITZER R, MANGARD S. Cache template attacks: Automating attacks on inclusive last-level caches[C]//Proceedings of 24th USENIX Security Symposium (USENIX Security'15). 2015: 897-912.
- [5] GRUSS D, MAURICE C, WAGNER K, et al. Flush+ Flush: a fast and stealthy cache attack[C]//Proceedings of Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA' 16). 2016: 279-299.
- [6] OSVIK DA, SHAMIR A, TROMER E. Cache attacks and countermeasures: the case of AES[C]//Proceedings of Cryptographers' Track at the RSA Conference (CT-RSA'06). 2006: 1-20.
- [7] GRAS B, RAZAVI K, BOS H, GIUFFRIDA C, et al. Translation leak-aside buffer: defeating cache side-channel protections with TLB attacks[C]//Proceedings of 27th USENIX Security Symposium (USENIX Security'18). 2018: 955-972.
- [8] SHINDE S, CHUA ZL, NARAYANAN V, et al. Preventing page faults from telling your secrets[C]//Proceedings of 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'16), Xi'an, China. 2016: 317-328.
- [9] XU Y, CUI W, PEINADO M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems[C]//Proceedings of IEEE Symposium on Security and Privacy(S&P'15). 2015: 640-656.
- [10] PESSL P, GRUSS D, MAURICE C, et al. DRAMA: exploiting DRAM addressing for cross-cpu attacks[C]//Proceedings of 25th USENIX Security Symposium (USENIX Security'16). 2016: 565-581.
- [11] HE Z, LEE RB. How secure is your cache against side-channel attacks?[C]//Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17). 2017: 341-353.
- [12] ZHANG D, ASKAROV A, MYERS AC. Language-based control and mitigation of timing channels[C]//Proceedings of 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12). 2012: 99-110.
- [13] WANG Y, FERRAIUOLO A, ZHANG D, et al. SecDCP: secure dynamic cache partitioning for efficient timing channel protection[C]//Proceedings of 53rd ACM/EDAC/IEEE Design Automation Conference (DAC'16). 2016: 1-6.
- [14] KIRIANSKY V, LEBEDEV I, AMARASINGHE S, et al. DAWG: a defense against cache timing attacks in speculative execution processors[C]//Proceedings of 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18). 2018: 974-987.
- [15] DENG S, XIONG W, SZEFER J. Secure tlbs[C]//Proceedings of 46th Annual International Symposium on Computer Architecture (ISCA'19). 2019: 346-359.
- [16] SHI J, SONG X, CHEN H, et al. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring[C]//Proceedings of 41st International Conference on Dependable Systems and Networks Workshops (DSN-W'11). 2011: 194-199.
- [17] KIM T, PEINADO M, MAINAR-RUIZ G. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud[C]//Proceedings of 21st USENIX Security Symposium (USENIX Security'12). 2012: 189-204.
- [18] ZHANG Y, REITER MK. DüPpel: retrofitting commodity operating systems to mitigate cache side channels in the cloud[C]//Proceedings of 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS'13). 2013: 827-838.
- [19] VATTIKONDA BC, DAS S, SHACHAM H. Eliminating fine grained timers in xen[C]//Proceedings of 3rd ACM Workshop on Cloud Computing Security Workshop. 2011: 41-46.
- [20] LI P, GAO D, REITER MK. StopWatch: A Cloud Architecture for Timing Channel Mitigation[J]. ACM Transaction Information System Security, 2014(17): 1-28.
- [21] MARTIN R, DEMME J, SETHUMADHAVAN S. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks[C]//Proceedings of 39th Annual International Symposium on Computer Architecture (ISCA'12). 2012: 118-129.
- [22] ZHANG T, ZHANG Y, LEE R B. Cloudradar: a real-time side-channel attack detection system in clouds[C]//Proceedings of Research in Attacks, Intrusions, and Defenses (RAID'16). 2016: 118-140.
- [23] CHEN S, ZHANG X, REITER MK, et al. Detecting privileged side-channel attacks in shielded execution with Déjà Vu[C]//Proceedings of 2017 ACM on Asia Conference on Computer and Communications Security (AsiaCCS'17). 2017: 7-18.
- [24] SHIH MW, LEE S, KIM T, et al. T-SGX: eradicating controlled-channel attacks against enclave programs[C]//Proceedings of 25th USENIX Security Symposium (USENIX Security'16). 2016: 565-581.

- of 24th Network and Distributed System Security Symposium (NDSS'17). USA, 2017.
- [25] DOYCHEV G, KÖPF B, MAUBORGNE L, et al. CacheAudit: a tool for the static analysis of cache side channels[J]. ACM Transactions on Information and System Security. 2015(18): 1-32.
- [26] ALMEIDA JB, BARBOSA M, BARTHE G, et al. Verifying constant-time implementations[C]//Proceedings of 25th USENIX Security Symposium (USENIX Security'16). 2016: 53-70.
- [27] BARTHE G, BETARTE G, CAMPO J, et al. System-level non-interference for constant-time cryptography[C]//Proceedings of 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14). 2014: 1267-1279.
- [28] REPARAZ O, BALASCH J, VERBAUWHEDE I. Dude, is my code constant time[C]//Proceedings of the Conference on Design, Automation & Test in Europe (DATE'17). 2017: 1701-1706.
- [29] WANG S, WANG P, LIU X, et al. Cached: Identifying cache-based timing channels in production software[C]//Proceedings of 26th USENIX Security Symposium (USENIX Security'17). 2017: 235-252.
- [30] WICHELMANN J, MOGHIMI A, EISENBARTH T, et al. Micro-walk: a framework for finding side channels in binaries[C]//Proceedings of 34th Annual Computer Security Applications Conference (ACSAC'18). 2018: 161-173.
- [31] WU M, GUO S, SCHAUMONT P, et al. Eliminating timing side-channel leaks using program repair[C]//Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18). 2018: 15-26.
- [32] CORON J S. Resistance against differential power analysis for elliptic curve cryptosystems[C]//Proceedings of 1th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99). 1999: 292-302.
- [33] VAN B J, PIESENS F, STRACKX R. SGX-Step: A practical attack framework for precise enclave execution control[C]//Proceedings of 2nd Workshop on System Software for Trusted Execution (SysTEX'17). 2017: 1-6.
- [34] WEISER S, ZANKL A, SPREITZER R, et al. DATA-differential address trace analysis: finding address-based side-channels in binaries[C]//Proceedings of 27th USENIX Security Symposium (USENIX Security'18). 2018: 603-620.
- [35] GENKIN D, PIPMAN I, TROMER E. Get your hands off my laptop: Physical side-channelkey-extraction attacks on PCs[C]//Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems (CHES'14), Busan, South Korea. 2014: 242-260.
- [36] LIPP M, SCHWARZ M, GRUSS D, et al. Meltdown: reading kernel memory from user space[C]//Proceedings of 27th USENIX Security Symposium (USENIX Security'18). 2018: 973-990.
- [37] KOCHER P, HORN J, FOGH A, et al. Spectre attacks: exploiting speculative execution[C]//Proceedings of 2019 IEEE Symposium on Security and Privacy (S&P'19). 2019: 1-19.
- [38] VAN BULCK J, MINKIN M, WEISSE O, et al. Foreshadow: extracting the keys to the intel SGX kingdom with transient out-of-order execution[C]//Proceedings of 27th USENIX Security Symposium (USENIX Security'18). 2018: 991-1008.
- [39] Mastik: a micro-architectural side-channel toolkit[EB].

[作者简介]



邹德清（1975—），男，湖南长沙人，博士，华中科技大学教授、博士生导师，主要研究方向为虚拟化安全与云安全、网络攻防与漏洞检测、大数据安全、容错计算。



张盼（1991—），男，湖北仙桃人，华中科技大学博士生，主要研究方向为系统安全、虚拟化安全、编译器安全、模糊测试、侧信道防御。



刘伟（1980—），男，江苏扬州人，北京航计算通讯研究所研究员，主要研究方向为软件安全性设计与验证、信息安全、软件及FPGA工程化及测试。



陈维杰（1999—），男，江苏泰州人，华中科技大学硕士生，主要研究方向为系统安全，侧信道防御。



陆弈帆（1998—），男，江苏苏州人，华中科技大学硕士生，主要研究方向为系统安全、模糊测试、漏洞检测。