

基于模型检验的缓存侧信道形式化验证与攻击检测

张士轩¹, 王海霞², 邱鹏飞³, 吕勇强², 尹凌峰², 王鸿鹏^{1,4}, 汪东升^{2,5}

¹ 哈尔滨工业大学(深圳) 计算机科学与技术学院 深圳 中国 518071

² 清华大学 北京 中国 100084

³ 可信分布式计算与服务教育部重点实验室(北京邮电大学) 北京 中国 100867

⁴ 鹏城实验室 深圳 中国 518000

⁵ 中关村实验室 北京 中国 100194

摘要 缓存侧信道攻击利用对共享缓存的访问延迟差异泄露私密数据, 具有隐蔽性高、难以防御的特点, 对计算机系统的安全性威胁越来越大。此外, 瞬态执行类攻击的发现也证明了缓存可以作为隐蔽信道传递瞬态窗口中的信息。因此, 需要对处理器上的缓存设计进行全面的验证和安全性分析, 以评估其是否容易受到潜在的侧信道攻击。现有的关于缓存的评估和验证工作只考虑了最基本的缓存模型, 而忽视了缓存的非包含属性和缓存的一致性协议等细粒度特征。随着大量的利用缓存细粒度特征的侧信道攻击方式被发现, 凸显了传统方法对于缓存侧信道的评估和验证的不足。本文提出一种基于模型检验技术的形式化验证框架, 该框架可系统地验证缓存设计并自动检测其中的侧信道攻击。为描述缓存的行为模型, 本文提出了一种基于有限状态机的缓存微架构建模方法。它详细地描述了对缓存相关操作指令和缓存特征进行抽象及约简的方法, 使得模型的状态空间显著减小。本文还根据延迟差分比较的思想定义了免受侧信道攻击的缓存需要满足的全局安全属性, 违反该安全规范的每条指令输入序列都是一个潜在的漏洞; 此外, 本文提出了一个基于模型检验的反例搜索算法, 它能够穷尽地搜索缓存的状态机模型, 并找到所有违反安全约束的反例。这使得验证框架具备批量验证并发现侧信道攻击的能力, 并且让验证过程更加高效。本文使用所提出的验证框架对当前处理器中应用广泛的非包含缓存模型和考虑缓存一致性的多核缓存模型进行了验证, 验证得到的反例能够覆盖现有的攻击方式, 并发现 9 种新的攻击方式。新发现的攻击方式可以让攻击者绕过一些侧信道防御措施, 也可以为受害者提供当前缓存设计中的脆弱点, 以便进行针对性的防御。我们在真实处理器上进行了测试, 结果证明了本文发现新攻击的有效性。

关键词 硬件安全; 形式化验证; 缓存侧信道攻击; 模型检验

中图分类号 TP311 **DOI号** 10.19363/J.cnki.cn10-1380/tn.2025.09.02

Formal Verification of Cache Side-channels and Attack Detection Based on Model Checking

ZHANG Shixuan¹, WANG Haixia², QIU Pengfei³, LYU Yongqiang², YIN Lingfeng², WANG Hongpeng^{1,4},
WANG Dongsheng^{2,5}

¹ School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen 518071, China

² Tsinghua University, Beijing 100084, China

³ Key Laboratory of Trustworthy Distributed Computing and Service(BUPT), Ministry of Education, Beijing 100867, China

⁴ Peng Cheng Laboratory, Shenzhen 518000, China

⁵ Zhongguancun National Laboratory, Beijing 100194, China

Abstract Cache side-channel attacks exploit differences in access delays to shared caches, revealing sensitive data with high stealth and resilience against defense mechanisms. Additionally, the discovery of transient execution attacks demonstrates that caches can serve as covert channels for transmitting information within transient windows. Therefore, comprehensive verification and security analysis of cache designs on processors are necessary to assess their susceptibility to potential side-channel attacks. Existing evaluations and verification efforts concerning caches predominantly focus on basic cache models, neglecting finer-grained features such as non-inclusive properties and cache coherence protocols. As numerous side-channel attack techniques exploiting cache's finer-grained features have been unveiled, it underscores the inadequacy of traditional methods for evaluating and verifying cache side-channels. This paper introduces a formal verification framework based on model-checking techniques, systematically validating cache designs and automatically detecting

通讯作者: 王鸿鹏, 博士, 教授, Email: wanghp@hit.edu.cn

本课题得到国家重点研发计划项目(No. 2021YFB3100902); 国家自然科学基金项目(No. 62072263); 广东省安全智能新技术重点实验室基础研究项目(No. 2022B1212010005); 深圳市基础研究专项(No. JCYC20210324132212030)资助。

收稿日期: 2023-12-11; 修改日期: 2024-04-10; 定稿日期: 2025-08-21

potential side-channel attacks. To describe the behavioral model of the cache, we propose a finite-state machine-based modeling approach for cache microarchitecture. This approach intricately outlines methods for abstracting and simplifying cache-related operation instructions and features, significantly reducing the model's state space. Moreover, following the idea of delay difference comparison, this paper defines the global safety properties that caches must satisfy to be immune to side-channel attacks. Violations of these safety specifications by any instruction input sequence indicate a potential vulnerability. Additionally, we present a model-checking-based counterexample search algorithm that exhaustively explores the cache's state machine model, uncovering all instances that breach safety constraints. This empowers the verification framework with the capability to batch verify and discover side-channel attacks efficiently. The proposed verification framework is applied to widely-used non-inclusive cache models on current processors and multi-core cache models considering cache coherence. The obtained counterexamples cover existing attack methods and reveal nine new attack methods. The newly discovered attack methods can enable attackers to bypass some side-channel defense measures and provide victims with insights into vulnerabilities in the current cache design for targeted defense. Real processor testing validates the effectiveness of discovering new attacks presented in this paper.

Key words hardware security; formal verification; cache side-channel attacks; model checking

1 引言

在现代处理器的设计中, 高速缓存对于提高处理器的运行速度有重要作用, 因此也成为不可或缺的微架构组件。然而, 由于缓存命中或者缺失等行为带来的显著时间差异, 其可以被利用以发动基于时间的侧信道攻击。这类侧信道攻击与传统的利用软件系统漏洞的攻击不同, 它利用的是硬件微架构的延迟特征, 更难以被防御和修复。缓存侧信道攻击不需要对硬件进行侵入式探测, 只需要利用系统提供的访存、刷除、计时等非特权指令即可完成攻击, 这更增加了攻击的隐蔽性。由于缓存是进程间和核心间的共享资源, 恶意进程可以通过故意制造争用的方式泄露信息。因此缓存侧信道可以跨越进程或虚拟机边界, 攻击加密算法^[1-5], 或破解地址空间布局随机化 (Address Space Layout Randomization, ASLR)^[6-7]等。随着 Spectre^[8]和 Meltdown^[9]等瞬态执行类漏洞^[10-13]的发现, 缓存侧信道攻击被证明还可以作为 Spectre 和 Meltdown 攻击流程中的一部分, 将瞬态窗口中的秘密信息泄露出来。这使得缓存侧信道攻击再次成为安全研究者关注的热点。

近年来, 缓存侧信道攻击越来越朝着攻击层次多样化和攻击方式精细化的方向发展。例如: Liu 等人提出的基于驱逐的跨核攻击方法^[14], 使得攻击层次从 L1 级缓存拓展到了末级缓存 (Last Level Cache, LLC)。攻击方式精细化指的是: 越来越多的缓存微架构内部的细粒度特征被侧信道攻击所利用, 例如: Yao 等人利用缓存一致性协议实现了跨核的侧信道攻击^[15]; Guo 等人提出了同时基于预取指令和缓存一致性协议的侧信道攻击方案^[16]; Lipp 等人提出的 ARMageddon 实现了对于 ARM 处理器非包含缓存结构的侧信道攻击^[17]。

现有针对缓存安全性的评估工作大多使用分析

测试的方法, 其中 Deng 等人提出的“三步模型”^[18]将缓存侧信道的攻击过程抽象为三个步骤, 并使用该方法对多个安全缓存设计进行了分析; CacheFx^[19]则构建了一个仿真测试框架, 并引入了熵的概念, 以评估缓存设计抵御侧信道攻击的能力。但这种基于模型分析或测试的方法无法覆盖所有可能的状态和执行路径, 因此不能对缓存设计进行系统和全面的分析。此外, 基于形式化验证的框架 CacheGuard^[20]虽然能够对缓存设计进行全面的验证, 但仅能适用于普通缓存, 而不具备对于包含细粒度特征的缓存的建模和验证能力。

本文提出了一个用于验证缓存非包含属性和缓存一致性的框架, 用于发现缓存微架构设计中所有可能的侧信道攻击。本文采用指令和模型抽象的方法对缓存微架构设计进行建模, 并首次提出了缓存的细粒度特征抽象建模方法, 增加了模型对于微架构状态的表达能力。本文的验证方案和安全规范基于延迟差分比较的思想, 使得该验证框架具备了缓存侧信道攻击的特征检测能力。我们使用计算树逻辑 (Computation Tree Logic, CTL)^[21]来描述安全属性。在使用者构建了缓存微架构模型后, 该框架可以根据定义的安全属性自动进行验证。如果不满足安全属性, 则会借助反例搜索算法自动输出所有反例。反例由抽象的攻击序列组成, 使用者可以通过检查反例, 以判断该缓存微架构是否容易受到攻击。总之, 本文的主要贡献如下。

(1) 提出了一个微架构层级的形式化验证框架, 用来验证和检测非包含缓存以及缓存一致性等微架构特征是否会受到侧信道攻击。

(2) 提出了一种针对缓存微架构和相关指令的抽象建模方法, 使得该框架具备针对缓存细粒度特征建模的能力。

(3) 提出了一个基于差分延迟比较思想的验证

方案和安全规范,使验证框架具备对缓存侧信道攻击的检测能力。之后提出了基于模型检验的反例枚举算法,利用该算法可以在一次验证中自动地找到给定微架构设计中的所有缓存侧信道攻击。

(4) 利用本文提出的框架对 Intel Skylake-X 架构的缓存进行了抽象建模和形式化验证,共发现了 9 种新的侧信道攻击。论文选取验证得到的 SpectrePrefetch 攻击序列在真实处理器上实现,并作为案例进行论述,证明我们发现攻击的有效性。

2 背景介绍

2.1 缓存侧信道攻击

我们根据攻击者和受害者是否共享内存地址空间将缓存侧信道攻击分为两大类:基于重用的攻击和基于争用的攻击。基于重用的攻击的代表是: Flush+Reload^[22]、Flush+Flush^[23]和 Evict+Reload^[24],该类攻击需要攻击者和受害者共享地址空间,受害者访问共享数据时会在缓存中留下痕迹,攻击者通过探测缓存行来推断秘密信息。基于争用的攻击的代表是 Prime+Probe^[1],它依靠攻击者的地址空间和受害者的地址空间在缓存中发生争用从而导致互相驱逐,因此攻击者和受害者不需要共享内存地址空间。当某一方的数据被驱逐出缓存后可以通过计时的方式探测到这种变化。该攻击的流程分为 3 步。

(1) 首先攻击者使用预先准备的驱逐集填充受害者可能会使用的 cache set,此时受害者的秘密数据会被驱逐出缓存。

(2) 等待受害者的访问,受害者的秘密数据会再次加载进入缓存,同时会驱逐对应 cache set 的驱逐集中的数据。

(3) 攻击者再次访问第(1)步中用到的驱逐集并分别计时,如果某个 cache set 的驱逐集访问时间更长,说明该 cache set 的驱逐集中的数据在第 2 步时被受害者驱逐出缓存,即该 cache set 就是受害者秘密数据所映射到的 cache set。

2.1.1 非包含缓存结构上的侧信道

目前的多级缓存在包含属性上的设计主要有 3 种:包含缓存(Inclusive Cache)、非包含缓存(Non-inclusive Cache)和独占缓存(Exclusive Cache)。包含缓存的设计是:下级缓存中的数据包括所有上级缓存中的数据,即如果一个数据不在下级缓存中,那么它也一定不存在上级缓存中。非包含缓存的设计指的是:下级缓存中的数据可能包括也可能不包括上级缓存的数据,即一个数据可以仅存在于下级缓存,也可以仅存在于上级缓存,或者同时存在于两

者中。独占缓存的设计指的是下级缓存的数据和上级缓存的数据无交集,即一个数据只能存在于上级缓存或者下级缓存这两者之一。当前处理器中的主流是非包含缓存设计,Intel 公司的处理器曾经主要采用包含缓存设计,但近年来也逐渐转向采用非包含设计的缓存。

非包含缓存比包含缓存更难受侧信道攻击的影响,下面用一个常见的跨核 Prime+Probe 攻击为例说明。图 1 展示了一个二级缓存的结构,包括两个核心的私有 cache set 和 LLC set,其中 LLC 为攻击者核心和受害者核心共享。在 Prime+Probe 攻击中的第一步,攻击者核心访问 LLC set 中的驱逐集并将受害者的敏感数据驱逐。如果是包含缓存的情况,攻击者访问 LLC 中对应 cache set 的驱逐集可以将敏感数据从 LLC 中驱逐,而且根据包含缓存的性质,敏感数据在私有缓存组(Private Cache Set)中的数据副本会随着 LLC 中对应数据被驱逐而被反向失效。这样当受害者再次访问私密数据的时候,私密数据会被加载回缓存中,此时受害者在 Probe 阶段再次访问驱逐集,通过时间差异可以判断受害者访问的敏感地址属于哪个 cache set。

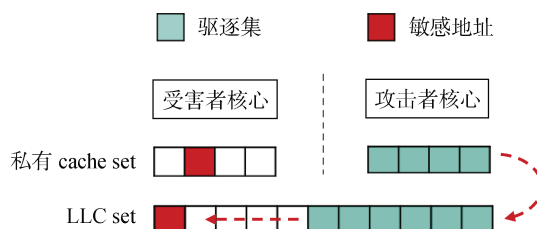


图 1 跨核 Prime+Probe 攻击示例

Figure 1 Example of cross-core Prime+Probe attack

但在非包含缓存中,当攻击者核心在 Prime 阶段访问驱逐集后,敏感数据可以从 LLC 中被驱逐到内存,但是由于非包含的特性,敏感数据在受害者核心私有缓存中的数据副本并不会被反向失效。那么第二步受害者访问秘密数据时,就不会从内存中读取数据,而是可以直接从私有缓存中读取数据,这就导致其不会对 LLC 中的 cache set 产生任何影响。因此在攻击者的 Probe 过程中,不会产生时间差异,也就无法攻击成功。因此,本文将重点选择非包含缓存结构建模并验证其安全性。

2.1.2 缓存一致性协议上的侧信道

在多核缓存中,缓存一致性协议也可以被利用以发动跨核的侧信道攻击,因此我们有必要对考虑缓存一致性协议的多核缓存设计的安全性进行形式化验证。目前处理器维护缓存一致性主要有:监听式

的方案和目录式的方案, 其中目录式方案由于其性能更好, 成为现在处理器设计的主流。目录式缓存一致性的设计主要是用 4 个状态来表示当前缓存中某数据所处的状态, 即修改(Modified, M)、独占(Exclusive, E)、共享(Shared, S)和失效(Invalid, I), 这就是最常用的 MESI 协议。此外, 还有一些处理器使用 MOESI 或者 MESIF 缓存一致性协议, 但它们实际上是 MESI 协议的拓展版。

利用缓存一致性协议的跨核攻击原理主要有两种。第一种为受害者的操作修改了和攻击者共享缓存地址的一致性状态, 从而产生了时间差异被攻击者探测到。例如: 在受害者私有核心中如果对共享地址的数据进行了修改, 那么根据缓存一致性协议会将其他核心的数据副本失效掉, 这样如果攻击者再次访问该数据就会发现它私有缓存中的数据变为了失效状态, 从而观测到访问延迟增加。然而作为缓存侧信道的共享地址, 一般并没有写入的权限。现有研究发现有两种方法可以实现与“写”操作类似的效果, 从而将其他核心的数据副本失效。第一种方法是使用 *prefetchw* 类指令^[16]。*prefetchw* 指令的作用是针对某个要修改的数据提前将它取入缓存。在 Intel 的缓存实现中, 为了该数据的写入做准备并实现效率的最大化, 系统不会校验该数据的读写权限就会直接把其他核心的数据副本失效, 从而节省写入的时间。这种优化不会在架构层面产生错误但会在微架构层面产生和“写”操作类似的效果, 从而被侧信道攻击所利用。

在瞬态执行窗口中执行写操作也可以将其他核心的数据副本失效, 但是在瞬态窗口中的写操作因为读写权限的保护, 并不会在架构层面上真

的写入成功。之后, 在处理器发现预测执行错误并进行状态回滚的时候, 这些执行的痕迹就留在了微架构中。该原理可以被利用作为缓存侧信道的攻击。

第二种利用缓存一致性协议的跨核攻击原理是“静默升级”机制^[25]。图 2 展示的是拥有三个核心和私有缓存以及一个共享 LLC 的缓存结构。其中子图(a)展示的是当一个核心要读取的数据不在其私有缓存中, 而且该数据的一致性状态为 E 时, 处理器会从该数据所属的私有核心缓存中而不是 LLC 中获取数据, 之后将数据传递给请求发起的核心。这是因为处理器中存在一种“静默升级”机制, 即如果某数据只在一个私有核心的缓存中存在, 那么该核心在对该数据做出修改时上级缓存不会向下级缓存发送同步信号, 而是先选择“静默”。而在另一个核心请求该数据时, 由于“静默升级”机制的存在, 处理器会认为 LLC 中的数据是陈旧的数据, 因此会去数据所属私有核心查看该数据是否被修改并将数据从私有的 L2 级缓存传递给请求核心的私有缓存。子图(b)展示的情况是当一个核心要读取数据不在其私有缓存中, 而且一致性状态为 S 时。由于 S 状态说明至少有两个核心保存该数据副本, 如果有写入操作, 那么必然有一个核心的数据副本失效变成只有一个核心保存数据副本, 此时的缓存一致性状态变化一定会同步到 LLC。因此处理器可以认为 S 状态的数据一定是干净的数据, 在 LLC 中的数据副本也一定是有效的数据, 因此其他核心请求数据的时候会从 LLC 转发到私有缓存。由于数据从私有 L2 → 私有 L2 和 LLC → 私有 L2 这两条传输途径之间存在时间差异, “静默升级”机制可以被用来作为侧信道。

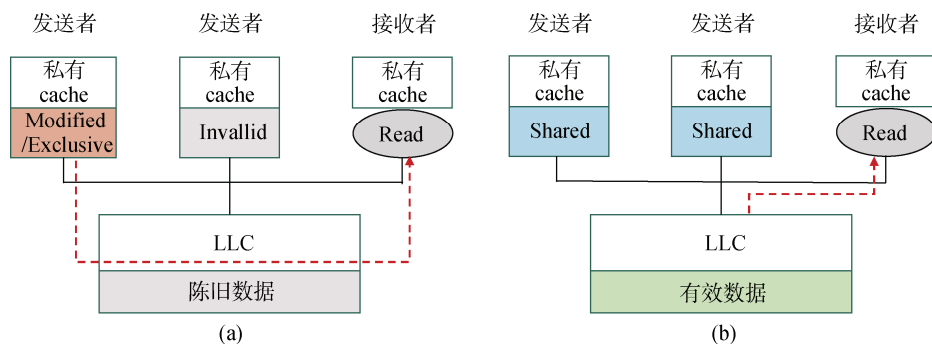


图 2 “静默升级”机制

Figure 2 “Silent upgrade” mechanism

由于非包含缓存结构和考虑缓存一致性的多核缓存结构的复杂性, 本文的研究工作将选择这两种结构设计作为验证对象。

2.2 模型检验

模型检验是一种用于有限状态并发系统的形式化验证技术, 相比于另一种常用的形式化验证方法:

定理证明,其主要优点是:不需要使用者手工撰写证明过程,而由模型检验工具自动完成验证,并且可以输出反例以帮助找到导致系统验证失败的原因。现已广泛应用于软件、电路和通信协议等领域的验证。模型检验被验证的系统需要建模成为一个有限状态机并使用 Kripke 结构表示^[26-28],即 $M = (S, S_0, R, L)$ 。其中 S 是有限状态集合, $S_0 \subseteq S$ 是初始状态的集合, $R \subseteq S \times S$ 表示状态变迁关系,我们令 AP 表示原子命题的集合,那么 $L: S \rightarrow 2^{AP}$ 表示标记函数,它标记了在该状态下为真的原子命题的集合。Kripke 结构可以展开为计算树的形式,树中包含了系统的所有状态,其中根节点为初始状态,树的分支可以表示系统所有的状态转换。

针对系统的规范或者安全约束主要由 CTL*逻辑描述。CTL*公式由两类符号组成,第一类是表示路径量词的符号: A (对于所有的路径), E (至少存在一条路径)。第二类为时序运算符,主要包括: X (次态)表示从某状态开始的下一个状态开始满足约束, F (将来)表示路径中将来的某个状态会满足约束, G (全局)表示约束在路径上的每个状态都满足。路径量词和时序运算符组合起来构成了 CTL*公式。如果用 f 表示 CTL*公式,那么模型检验的目标就是找到 S 中满足公式 f 的所有状态集合 $\{s \in S \mid M, s \models f\}$ 。CTL*逻辑包含两个子逻辑:基于分支时间的计算树逻辑和基于线性时间的线性时序逻辑(Linear Temporal Logic)。本文主要使用 CTL 公式定义安全规范。

3 威胁模型

我们假设攻击者的目标是通过相关内存操作的时间差异来窃取受害者的秘密信息。攻击者和受害者共享缓存资源,这是产生缓存侧信道的根本原因。受害者的程序中存在这样一些代码,它们会根据秘密数据的不同而访问不同的内存地址。攻击者通过测量某些操作的延迟差异来获得受害者所访问的内存地址并以此推测受害者的秘密信息。我们假设攻击者处在非特权级下,攻击者可以执行访存、刷除、预取以及在瞬态窗口中写入等非特权级下具有执行权限的操作。攻击者需要知道受害者的一些代码信息(例如:受害者所采用的加密算法),以获得秘密数据和内存地址之间的关系。攻击者可以测量他自己的操作延迟,也可以调用受害者的代码并测量其执行时间。此外,我们假设攻击者可以完全操控木马进程以辅助攻击(本文木马的概念是指软件木马,并非指硬件电路构成的木马)。攻击者、木马进程和受害

者分布在不同的物理核心上。

4 缓存行为模型验证框架

4.1 验证框架概述

框架总体上的流程主要分为 3 个阶段,如图 3 所示。

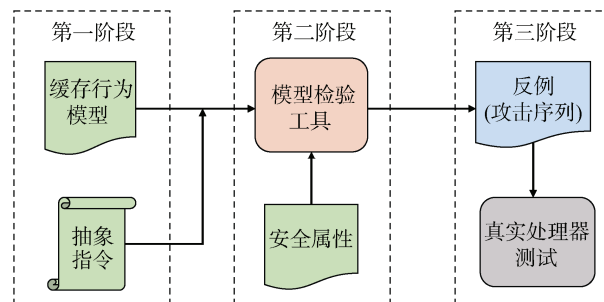


图 3 形式化验证框架

Figure 3 Formal verification framework

第一阶段:建模缓存行为模型。我们使用状态机模型来建模缓存的行为,并使用模型检验中常用的 Kripke 结构来表示有限状态机。由于真实缓存的状态空间极大,无法直接建模并输入到模型检验工具进行验证。因此我们定义了一套针对缓存行为模型的抽象和建模方法,其中包括缓存相关指令的建模和缓存微架构的建模规则。这一部分具体在 3.2 节介绍。

第二阶段:定义安全规范作为模型检验器的输入。我们使用 CTL 公式来表示 Kripke 结构需要满足的约束规范。安全规范的定义关键要找到缓存侧信道攻击的特征,违反安全规范的反例就可以视为一个潜在的安全威胁。该部分的内容主要在 3.3 节介绍。

之后,将缓存模型和安全规范输入模型检验器进行验证。由于现有的模型检验工具只能输出一个反例,而无法找到系统中所有可能的侧信道攻击方法。因此我们对模型检验工具进行了一些拓展,实现了基于模型检验的反例搜索算法,增强了传统模型检验工具的能力,使其能够找到系统中所有可能的缓存侧信道攻击模式。这部分内容具体在 3.4 节阐述。

第三阶段:我们对模型检验输出的反例进行人工的筛选和分析,并对其中某些攻击序列在真实处理器上进行实现从而验证该攻击的有效性。

4.2 缓存建模方法

为了对某特定架构缓存的侧信道攻击进行形式化验证,我们观察并提取缓存侧信道攻击通用的模式,以确定如何对缓存建模以及需要在缓存形式化模型中包含哪些特征。

如基础知识部分所介绍的, 一轮完整的侧信道攻击往往可以被抽象为三个阶段的操作。首先第一阶段: 攻击者通过访存、刷除、预取等操作来设置缓存的初始状态。第二阶段: 调用或者等待受害者程序的执行。第三阶段: 攻击者通过执行一些指令并计时, 通过观察其时间差异来判断第二阶段中受害者访问了哪些地址, 或者受害者访问了哪个缓存集(cache set), 通过这些获得的信息就可以推断出受害者中的秘密数据。我们可以给出如下的断言: 每次攻击流程只有最后一步需要进行时间测量, 因为前面的几步在时间上都是没有差异的。这个结论可以由反证法证明: 如果在前面几步中时间也有差异, 那么截止到有时间差异的那一步, 其本身就可以构成侧信道。因此我们可以得到:

定义 1. 缓存模型的输入可以抽象表示为一串指令操作序列 L :

$$L = [\text{op}_1 \rightarrow \text{op}_2 \rightarrow \dots \rightarrow \text{op}_n] \quad (1)$$

该操作指令序列由 n 条抽象的缓存操作指令组成。对缓存侧信道进行形式化验证的目标就是输出这样的抽象操作序列形式的结果。实现一个缓存侧信道攻击往往至少需要 3 步抽象的指令操作, 某些攻击也可能需要 3 步以上的指令操作才能成功, 因此 $n \geq 3$ 。其中每个操作指令 op_n 的抽象建模方式在 4.2.1 小节介绍。

4.2.1 缓存相关指令建模

缓存相关的操作指令的抽象建模主要包括指令、地址空间、操作对象等几个部分。本文方法参考并拓展了“三步模型”^[18]论文中的方法。

首先需要明确指令的执行对象, 即本条指令是由攻击者执行的还是由受害者执行的。由攻击者执行的指令用大写字母 A 来表示, 由受害者执行的指令用大写字母 V 表示。其中某些攻击可能涉及 3 个核心, 因此我们设定一个木马核心用大写字母 T 表示, 它可以和攻击者协同操作, 从而窃取受害者的信息。

之后我们在大写字母的上角标位置定义该指令的操作类型。上角标位置有三种情况, 分别是为: 空, inv , w 。

(1) 如果上角标位置为空, 则该指令为访存指令(例如: x86 下的 mov 指令或者指针的解引用)。

(2) 如果上角标位置的字符为 inv , 则说明该指令为“冲刷”指令, 它的作用是可以将该地址的数据在缓存中失效并写回内存(在 x86 架构中可以用 clflush 指令实现)。

(3) 上角标位置为 w 时, 如果是受害者执行的指

令(例如: V^w), 则说明此时受害者执行的是瞬态写操作。虽然一般的隐蔽信道中, 所借助的地址受害者和攻击者都不具备写权限, 但是在瞬态执行窗口中依然可以进行“写”操作, 该写操作在流水线后端并不会提交, 也并没有真正地在该位置写入数据。因此产生的效果只会在微架构层中留下痕迹, 而在架构层上并不可见。然而留存在微架构中的痕迹依然可以被用来推断受害者的私密信息。

(4) 如果上角标 w 搭配的是攻击者执行的指令(例如: A^w), 我们就将其抽象为“预取”指令(在 x86 架构中可以用 prefetchw 指令实现)。“预取”指令也可以对只有读权限没有写权限的数据执行, 然而 CPU 在预取之后会默认当作用户要对该数据进行修改, 从而根据缓存一致性协议提前将其他核心中的数据失效掉。虽然此时数据并没有被修改, 但对其他核心产生的微架构层面的影响依然可能被侧信道攻击所利用。

上角标 w 主要对应的是基于缓存一致性的攻击类型, 在后面的缓存微架构形式化模型构建章节和形式化验证结果与分析的章节还会具体阐述。

最后, 需要对指令的作用地址进行抽象建模, 因为真实处理器的地址空间是一个非常大的值, 我们不可能也没有必要对这么大的地址空间进行建模验证。因此我们将地址抽象后分为几个类别: a , b , d , u , NIB, 它们都被写在符号表示的下角标的位置处来指示该指令所作用的地址。这些地址抽象类型的定义如表 1 所示。

根据已经定义的一条抽象指令的三个要素, 即指令执行的对象, 指令的操作类型和指令的操作地址, 我们举例说明该抽象指令的语义。

(1) V_u : 受害者访问秘密地址。

(2) V_u^w : 受害者在瞬态执行窗口尝试对私密地址进行写操作。

(3) A_a^{inv} : 攻击者在 a 地址执行刷除操作。

(4) A_a^w : 攻击者在 a 地址执行预取操作。

(5) A_d : 攻击者访问 d 类地址的集合, 相当于填充驱逐集。

(6) Invany : 将所有地址从缓存中刷除。

4.2.2 缓存行为模型构建

我们采用有限状态机模型对缓存的特征进行建模, 首先以普通单级缓存为例展示缓存行为模型的构建方法。

对于普通单级缓存, 由于大多数缓存侧信道攻击最终都是通过计时结果来判断私密数据地址所处

表 1 指令操作地址抽象类型的定义
Table 1 Definition of the abstract type of the operation address

地址符号	描述
u	u 表示未知的由受害者访问的秘密地址, 例如在加密算法或者瞬态窗口中访问的地址, 例如: 受害者访问 u 地址的操作可以用 V_u 表示。攻击者的任务就是通过一组操作和计时来推断 u 的值。由于是秘密地址, 因此攻击者不能直接访问 u
a	a 是由攻击者或者受害者访问的某个确定的地址。该地址可能等于也可能不等于 u , 因此 a 和 u 是共享内存地址空间的
b	b 是和 a 映射到相同 cache set 的一个地址集合。 b 类地址一般不用来作为操作指令的地址, 而是用来判断 u 地址是否属于 b 的集合
d	d 是和 a, b 地址映射到相同 cache set 的一个地址集合, 但是和受害者访问的地址 u 以及 a 并不共享地址空间, 访问 d 类地址并不能直接命中 u 。 d 集合的地址通常作为驱逐集来使用
NIB	NIB 表示那些和地址 a 不在同一个 cache set 的地址集合。它一般不作为指令访问的地址, 而是用来指示 u 所处的位置

的 cache set 或者它的虚拟地址来推断秘密数据, 因此我们将整个缓存微架构抽象为两个缓存行(cache line)。一个用来表示和私密数据地址映射到了相同的 cache set, 另一个表示和私密数据地址映射到了不同的 cache set, 这样就足以覆盖缓存侧信道攻击的全部情况。第一个缓存行是地址 a 所在的缓存行, 该缓存行可取的值包括 a 、 b 、 d 、以及空状态(使用 f 来表示空状态)。我们定义变量 C_1 来表示第一个缓存行所处的状态, 且 $C_1 \in \{a, b, d, f\}$ 。第二个建模的缓存行表示和地址 a 映射到不同 cache set 的那些地址, 用变量 C_2 来表示该缓存行所处的状态, 且 $C_2 \in \{NIB, f\}$ 。

$$\frac{\forall C_1 C_2 \text{ Addr Obj. } (C_1 = \text{Addr}) \wedge (\text{Op} = \text{rd} \vee \text{Op} = \text{wr})}{C_1 = \text{Addr}, \text{Latency} = \text{fast}} \quad (2)$$

$$\frac{\forall C_1 C_2. (\text{Addr} = \text{NIB}) \wedge (\text{Obj} = V) \wedge (\text{Op} = \text{rd} \vee \text{Op} = \text{wr})}{C_2 = \text{NIB}, \text{Latency} = x} \quad (3)$$

$$\frac{\forall C_1 C_2 \text{ Addr Obj. } (C_1 = \text{Addr}) \wedge (\text{Op} = \text{inv})}{C_1 = f, \text{Latency} = \text{slow}} \quad (4)$$

$$\frac{\forall C_1 C_2 \text{ Addr Obj. } (C_1 \neq \text{Addr}) \wedge (\text{Op} = \text{rd} \vee \text{Op} = \text{wr})}{C_1 = \text{Addr}, \text{Latency} = \text{slow}} \quad (5)$$

$$\frac{\forall C_1 C_2 \text{ Addr Obj. } (C_1 \neq \text{Addr}) \wedge (\text{Op} = \text{inv})}{\text{Latency} = \text{fast}} \quad (6)$$

$$\frac{\forall C_1 C_2 \text{ Addr Obj. } (\text{Op} = \text{Invany})}{\text{Latency} = x} \quad (7)$$

$$\frac{\forall C_1 C_2 \text{ Addr Obj. } (C_2 = \text{Addr} = \text{NIB}) \wedge (\text{Op} = \text{rd} \vee \text{Op} = \text{wr})}{\text{Latency} = \text{fast}} \quad (8)$$

此外, 缓存时间侧信道主要通过时间延迟来泄露私密数据, 因此我们需要对模型输出的延迟进行建模。在这里我们不考虑每条指令执行所需的具体时间或者时钟周期, 而是直接将延迟抽象为“快”和“慢”两种情况, 因为大多数缓存时间侧信道攻击也是通过区分延迟“快”或者“慢”这两种情况来推测私密数据。例如: 在访存指令中, 如果该地址存在于缓存中, 则缓存命中, 延迟为“快”。如果该地址不在缓存中, 则缓存缺失需要从内存中加载数据, 延迟为“慢”。此外还存在一些延迟不确定的情况, 用字母 x 表示, 它主要代表本模型不考虑延迟的一些情况。我们使用变量 Latency 表示缓存状态机对于输入某条指令后所输出的延迟, 且 $\text{Latency} \in \{\text{fast}, \text{slow}, x\}$ 。

为了方便表示, 我们将抽象指令的三个要素: 指令执行的对象, 指令的操作类型和指令的操作地址拆开并分别定义。首先指令执行的对象使用变量 Obj 表示, 且 $\text{Obj} \in \{V, A, T\}$, 即执行指令的对象只可能是受害者、攻击者和木马这三种情况。指令的操作类型用变量 Op 表示, 且根据前面的定义 $\text{Op} \in \{\text{rd}, \text{wr}, \text{inv}, \text{Invany}\}$ 。指令的操作地址使用变量 Addr 表示, 且根据前文定义的抽象地址类型 $\text{Addr} \in \{a, b, d, u\}$ 。下面我们开始定义缓存模型的操作语义, 其表示形式如下, 横线上方为当前状态和状态机输入, 横线下为状态机接受输入后的下一个状态和输出延迟。

定义 2. (缓存操作指令的语义)

普通缓存操作指令的语义由公式(2)~公式(8)定义。

其中公式(2)的语义是当缓存行 C_1 的当前状态等于 Addr 输入, 而且操作类型 Op 为读或者写的时候, 输出的延迟 Latency 为 fast, 且 C_1 的状态会变成和 Addr 相同。

公式(3)的语义是当输入指令的操作地址为 NIB 时, 映射到 a 之外缓存行, 可以不考虑这种延迟情况。

公式(4)的语义是当 C_1 的当前状态等于 Addr 输入, 而且操作类型 Op 为刷除的时候, 输出的延迟 Latency 为 slow, 因为当执行刷除指令时, 相比于数据不在缓存中的情况, 数据在缓存中需要引发额外的操作, 因此执行得更慢。且 C_1 的状态会变成 f (空状态)。

公式(5)的语义是当缓存行 C_1 的当前状态不等于 Addr 输入, 而且操作类型为读或写时, 说明此时为

缓存未命中。因此延迟更高, 而且 C_1 的状态会变成和 Addr 相同。

公式(6)的语义为: 当缓存行 C_1 的当前状态不等于 Addr 输入, 而且操作类型为刷除时, 说明此时要刷除的数据并不在缓存中, 因此缓存行 C_1 保持原状, 且 Latency 为 fast。

$$\frac{((\text{pri}A_1 = \text{Addr} \wedge \text{Obj} = A) \vee (\text{pri}V_1 = \text{Addr} \wedge \text{Obj} = V)) \wedge (\text{Op} = \text{rd} \vee \text{Op} = \text{wr})}{\text{Latency} = \text{fast}} \quad (9)$$

$$\frac{((\text{pri}A_1 = \text{Addr} \vee \text{llc}_1 = \text{Addr}) \wedge \text{Obj} = A) \wedge (\text{Op} = \text{inv})}{\text{pri}A_1 = f, \text{llc}_1 = f, \text{Latency} = \text{slow}} \quad (10)$$

$$\frac{((\text{Addr} \neq \text{pri}V_1 \wedge \text{Addr} \neq \text{llc}_1) \wedge \text{Obj} = V) \wedge (\text{Op} = \text{rd} \vee \text{Op} = \text{wr})}{\text{pri}V_1 = \text{Addr}, \text{llc}_1 = \text{Addr}, \text{Latency} = \text{slow}} \quad (11)$$

$$\frac{(\text{Addr} = d \wedge \text{Obj} = A \wedge \text{llc}_1 \neq d) \wedge (\text{Op} = \text{rd} \vee \text{Op} = \text{wr})}{\text{llc}_1 = d, \text{pri}A_1 = d, \text{Latency} = \text{slow}} \quad (12)$$

$$\frac{(\text{Addr} = d \wedge \text{Obj} = A \wedge \text{llc}_1 = d) \wedge (\text{Op} = \text{rd} \vee \text{Op} = \text{wr})}{\text{pri}A_1 = d, \text{Latency} = \text{fast}} \quad (13)$$

$$\frac{(\text{Addr} = d \wedge \text{Obj} = V) \wedge (\text{Op} = \text{rd})}{\text{pri}V_1 = d} \quad (14)$$

上述是一些普通缓存的基础语义规则, 后文具体要验证的多级非包含缓存和考虑缓存一致性的缓存模型还会在该语义的基础上考虑它们特有的微架构特征并加入一些新的语义。在普通缓存中我们只建模了 2 个 cache set, 但是在多级缓存模型中, 2 个 cache set 不能表示所有可能的情况。我们建模了 6 个 cache set, 包括受害者的私有缓存分别有 2 个 cache set, 即: $\text{pri}V_1 \in \{a, b, d, f\}$ 和 $\text{pri}V_2 \in \{\text{NIB}, f\}$ 。攻击者的私有缓存建模了 2 个 cache set, 即: $\text{pri}A_1 \in \{a, b, d, f\}$ 和 $\text{pri}A_2 \in \{\text{NIB}, f\}$ 。此外也建模 2 个 cache set 用来表示 LLC 中的情况, 即: $\text{llc}_1 \in \{a, b, d, f\}$ 和 $\text{llc}_2 \in \{\text{NIB}, f\}$ 。下面我们来定义非包含多级缓存模型的操作语义和状态转换。

定义 3. (非包含多级缓存操作指令的语义)

非包含多级缓存的操作语义如公式(9)~公式(14)定义。

其中公式(9)的语义说明当攻击者或受害者在本地核心的私有缓存命中数据时, 延迟都为“快”。

公式(10)的语义表示攻击者执行刷除操作, 且目标地址在缓存中时, 此时延迟为“慢”。由于受害者一

般不会执行刷除操作, 因此建模语义不考虑这种情况。

公式(8)的语义为: 缓存行 C_2 的状态如果和输入的 Addr 相同且都等于 NIB, 那么此时读写数据也认为是缓存命中, 且 Latency 为 fast。

公式(11)的语义表示当受害者的访问没有命中私有缓存或 LLC 时, 此时延迟结果为“慢”而且私有缓存和 LLC 的状态将会变成和输入地址相同。同理, 攻击者操作的命中情况和刷除情况可以参考前面的规则撰写, 在此不再赘述。

公式(12)的语义规则表示当攻击者访问 d 类地址时对应真实世界中的情况是攻击者状态驱逐集的过程, 而且攻击者可以访问她私有缓存和 LLC 中的驱逐集。而且我们主要考虑跨核攻击的情况, 当 LLC 对应 cache set 中的状态不是 d 时, 此时状态驱逐集的延迟为“慢”。

公式(13)的语义规则为上一个规则的相反情况, 即在访问驱逐集过程中, 如果 LLC 中的 cache set 已经是对应驱逐集的地址, 那么访问时间为“快”。

公式(14)表示的直接语义是受害者访问驱逐集的状态变化。但实际上受害者并不会主动访问驱逐集, 因此该语义在实际攻击中对应的情况是: 程序运行中的不可避免的操作系统活动会访问一些地址, 从而会将受害者私有缓存中的数据驱逐出去。这些“系统活动”实际上实现的效果等价于在受害者核心访问驱逐集, 但它并不会对 LLC 中的状态产生影响, 因为系统活动所访问的地址量很少, 它只可能将数据从较小的私有缓存中驱逐, 但是 LLC 容量很大, 因此对于 LLC 中数据的驱逐效果可以忽略。而且由于“系统活动”的不确定性, 我们也无法测得其延迟, 因此延迟情况无须建模。本条规则所建模的“系统活动”可以对私有缓存中的数据进行驱逐, 这也是我们在非包含缓存中的跨核攻击能够成功的关键。

$$\frac{\text{Obj} = A \wedge \text{Op} = \text{rd} \wedge \text{Addr} = a \wedge C_{A1} = I \wedge (C_{V1} = E \vee C_{T1} = E)}{C_{A1} = S, C_{V1} = S, \text{Latency} = \text{slow}} \quad (15)$$

$$\frac{\text{Obj} = A \wedge \text{Op} = \text{rd} \wedge \text{Addr} = a \wedge C_{A1} = I \wedge (C_{V1} = S \wedge C_{T1} = S)}{C_{A1} = S, \text{Latency} = \text{mid}} \quad (16)$$

$$\frac{\text{Obj} = A \wedge \text{Op} = \text{wr} \wedge \text{Addr} = a \wedge (C_{V1} = S \vee C_{T1} = S \vee C_{V1} = E \vee C_{T1} = E)}{C_{A1} = E, C_{V1} = I, C_{T1} = I, \text{Latency} = \text{slow}} \quad (17)$$

$$\frac{\text{Obj} = A \wedge \text{Op} = \text{wr} \wedge \text{Addr} = a \wedge (C_{V1} = I \wedge C_{T1} = I \wedge C_{A1} = E)}{C_{A1} = E, \text{Latency} = \text{fast}} \quad (18)$$

下面我们将多核缓存一致性的特征拓展到普通缓存上, 并建立形式化模型。因为只有共享地址空间的数据才需要维护一致性, 因此利用缓存一致性的攻击必然是共享内存地址空间的。所以输入指令的地址 $\text{Addr} \in \{a, u, \text{NIB}\}$, 这里主要区分受害者访问的地址 u 等于 a 或者不等于 a 。指令的操作类型包括: $\text{Op} \in \{\text{rd}, \text{wr}, \text{inv}, \text{Invany}\}$ 。这里的 wr 并不表示真正对数据进行修改, 而是指在瞬态窗口中的写操作或者预取操作。在实际的缓存侧信道攻击中, 不管是攻击者还是受害者都不具备对共享地址修改的权限, 因为如果她们可以对共享地址进行修改, 那么将可以直接修改数据传输信息, 而不再需要“侧信道”了。虽然两者都不能在架构层面对共享地址的数据进行修改, 但是可以在微架构层面实现和写操作类似的可以影响微架构状态的效果。

此外由于利用一致性的攻击可能涉及 3 个核心, 因此 $\text{Obj} \in \{A, V, T\}$, 即内存操作的发出核心可以是攻击者、受害者或者木马。我们将这 3 个核心中的缓存状态抽象为 6 个缓存行, 每个核心有 2 个缓存行, 分别表示地址 a 所在的缓存行和地址 a 之外的缓存行, 即: $C_{V1} \in \{S, E, I\}$ 、 $C_{V2} \in \{\text{NIB}, f\}$ 、 $C_{A1} \in \{S, E, I\}$ 、 $C_{A2} \in \{\text{NIB}, f\}$ 、 $C_{T1} \in \{S, E, I\}$ 和 $C_{T2} \in \{\text{NIB}, f\}$ 。在 a 所在的缓存行中, 因为不考虑和受害者不共享内存地址, 所以该缓存行中的数据只有可能是 a , 因此我们直接用数据 a 的一致性状态: S, E, I , 来表示该缓存行中的数据, 而且由于攻击者不可能修改该数据, 因此省略了 M 状态。对于延迟情况, 多核一致性缓存系统中也会更加复杂, 我们定义: $\text{Latency} \in \{\text{fast}, \text{mid}, \text{slow}, x\}$, 其中 fast 对应着数据访问或者写入时直接在本私有核心命中, mid 指的是在 LLC 中命中, slow 指的是在其他私有核心命中, x 表示不确定的状态。接下来将上述的微架构特性描述建立形式化的缓存行为规范和操作语义。

定义 4. (考虑缓存一致性的多核缓存操作指令的语义)

操作语义由公式(15)~公式(18)定义。

公式(15)定义的规则表示: 当一个核心要读取的数据不在其私有缓存中, 而且该数据的一致性状态为 E 时, 处理器会从该数据所属的私有核心缓存中而不是 LLC 中获取数据, 之后将数据传递给请求发

起的核心, 此时的延迟为 slow 。这是因为处理器中存在一种“静默升级”机制, 即如果某数据只在一个私有核心的缓存中存在, 那么该核心在对该数据做出修改时上级缓存不会向下级缓存发送同步信号, 而是先选择“静默”。而在另一个核心请求该数据时, 由于“静默升级”机制的存在, 处理器会认为 LLC 中的数据是陈旧的数据, 因此会去数据所属私有核心查看该数据是否被修改并将数据从私有的 L2 级缓存传递给请求核心的私有缓存。

公式(16)定义的规则表示: 当一个核心要读取数据不在其私有缓存中, 而且一致性状态为 S 时。由于 S 状态说明至少有两个核心保存该数据副本, 如果有写入操作, 那么必然有一个核心的数据副本失效变成只有一个核心保存数据副本, 此时的缓存一致性状态变化一定会同步到 LLC。因此处理器可以认为 S 状态的数据一定是干净的数据, 在 LLC 中的数据副本也一定是有效的数据, 因此其他核心请求数据的时候会从 LLC 转发到私有缓存。对应图 2(b)的情况, 即当其他核心状态为 S 时, 从 LLC 中读取数据的情况。上述 2 条语义描述了处理器中的“静默升级”机制。

公式(17)定义的规则表示当在攻击者核心执行“预取”操作时, 如果其他核心数据是有效的, 那么需要发送请求到其他核心将他们的数据失效, 因此延迟情况为 slow 。

公式(18)定义的规则表示当其他核心的数据是无效状态时, 由于此时执行“预取”不需要再发送信号将他们的数据失效掉, 因此执行速度较快, 延迟为 fast 。

本定义仅考虑 $\text{Obj} = A$ 的情况, 即攻击者为操作对象, 其他核心对应的操作与攻击者核心类似, 在此不再赘述。

4.3 安全属性定义

根据前文对于缓存指令和行为模型的抽象建模方法的定义可知, 地址 u 为受害者所操作的私密地址, 且受害者可以在该地址上进行读或者瞬态写操作(例如 V_u 和 V_u^w)。因此我们定义: 缓存侧信道攻击的目标就是通过一组内存操作序列推断出秘密地址 u 的值或者所属范围。根据我们对地址的抽象结论, 地址 u 的取值只可能有 3 种情况: a, b, NIB 。因此我

们将 u 的这 3 种取值情况构建出 3 个子模型, 这 3 个子模型会根据输入指令地址的不同, 分别输出对应 3 种情况的延迟。如果攻击者在最后一步的指令操作后得到的 3 个子模型延迟完全相同, 则说明攻击者不能分辨出秘密地址的位置, 否则如果 3 个子模型输出的延迟存在差异, 则攻击者可以分辨出秘密地址的位置。具体情况如图 4 所示, 子图(a)展示的情况是当 u 等于 a 时延迟为快, 当 u 映射到 b 或者 NIB 时, 延迟为慢。此时可以根据延迟准确地区分出秘密地址 u 的具体值(类似于 Flush+Reload 类攻击所具备的分辨能力)。子图(b)所展示的情况是当 u 等于 a 或

者 u 映射到 b 时, 计时结果为快, 当 u 映射到 NIB 时计时结果为慢。因为 a 和 b 地址是映射到同一个 cache set 的, 当攻击者以 cache set 为单位进行探测时就会找到某个 cache set 比其他的访问延迟都要快, 因此攻击者可以区分出 u 映射到具体哪个 cache set(类似于 Prime+Probe 攻击所能实现的分辨能力)。子图(c)所展示的情况是 3 种情况的延迟完全相同, 此时攻击者无法区分秘密地址 u 属于哪一类地址, 因此攻击无法成功。此外, 若输出为 x , 则说明这种情况为模型考虑之外的不定态, 因此必然无法构成攻击。

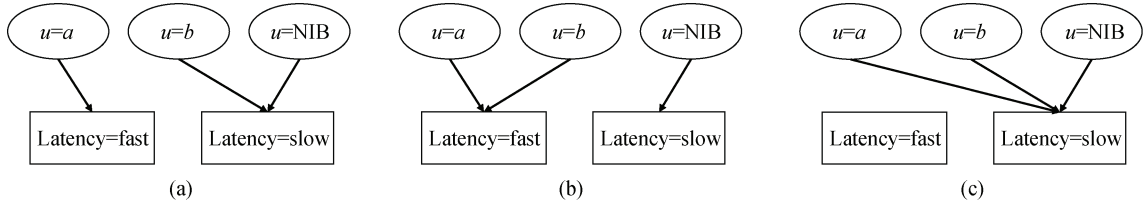


图 4 子模型输出延迟的情况
Figure 4 Output delay in submodule

根据上述分析, 我们将安全规范定义如下:

定义 5. (能构成攻击的安全属性).

$$((L_a = L_b) \wedge (L_a \neq L_{\text{NIB}})) \vee ((L_a \neq L_b) \wedge (L_a \neq L_{\text{NIB}})) \Rightarrow \text{Attack} = \text{true} \quad (19)$$

其中, L_a , L_b 和 L_{NIB} 分别表示当 u 等于 a , b 和 NIB 时缓存子模型所输出的延迟, Attack 变量表示攻击能否成功。上述合取范式的第一个子句和第二个子句代表了不同的攻击分辨率, 其中第一个子句表示攻击者仅能分辨受害者访问了哪个 cache set, 但不能确定具体访问的地址。第二个子句表示攻击者能够分辨出受害者访问了哪个地址。

定义 6(不构成攻击的安全属性).

$$(L_a = L_b = L_{\text{NIB}}) \vee (L_a = x) \vee (L_b = x) \vee (L_{\text{NIB}} = x) \Rightarrow \text{Attack} = \text{false} \quad (20)$$

即当无法区分三个子模型输出的延迟, 或者延迟中包含 x 状态时, 无法构成攻击。

对于最终输入形式化验证工具的全局安全规范, 我们需使用 CTL 逻辑定义:

定义 7(CTL 逻辑表示的全局安全属性).

$$AG \neg (\text{Attack} = \text{true}) \quad (21)$$

其中, A 和 G 为 CTL 逻辑中的符号, A 为描述路径的量词, 指的是对于计算树中的所有路径。 G 为时序描述符, 指的是对于某路径上的每个状态都满足某项性质或约束。因此该全局安全规范表述的语义是: 对于计算树中所有路径上的所有状态都不能

构成攻击。

4.4 反例搜索算法

当缓存模型不满足约束时, 模型检验工具会输出反例。然而现有的模型检验工具普遍只能输出一个反例。但实际上, 一个缓存设计上所能实现的侧信道攻击方式不止一种, 而我们的目标是找到某个缓存设计中所有潜在的侧信道攻击方式。如果模型检验工具每次只能输出一个结果, 那么即使是多次运行, 它也有很大的概率找到现有已知的攻击方式, 而无法找到新的攻击类型。我们提出了一种新的路径搜索算法, 使之可以在一次验证中输出所有违反约束的反例, 该算法在 NuSMV 工具的基础上实现。该算法的流程如图 5 所示。

NuSMV 的源码实现了一个专门的函数 check_AG_only 来检查那些只包含 AG 规则的 CTL 逻辑公式, 这恰好满足本文所定义的 CTL 约束的验证需要。我们主要在该函数的基础上进行拓展, 使其具备输出所有反例的能力。该算法的前 3 步和源代码的实现相同, 第 4 步开始为我们主要实现的部分。第 4 步调用系统接口计算从初始状态开始到达违反约束状态的最短路径 d , 并从最短路径开始进行前向的广度优先搜索。第 6 步为预先计算后面搜索过程需要用到的状态列表, 列表的作用是记录 $i \in [0, d]$ 中每个距离为 i 下面的状态集合, 该集合中的状态在经过 $k-i$ 次转换之后必然可以到达违反约束的状态。第 7~10

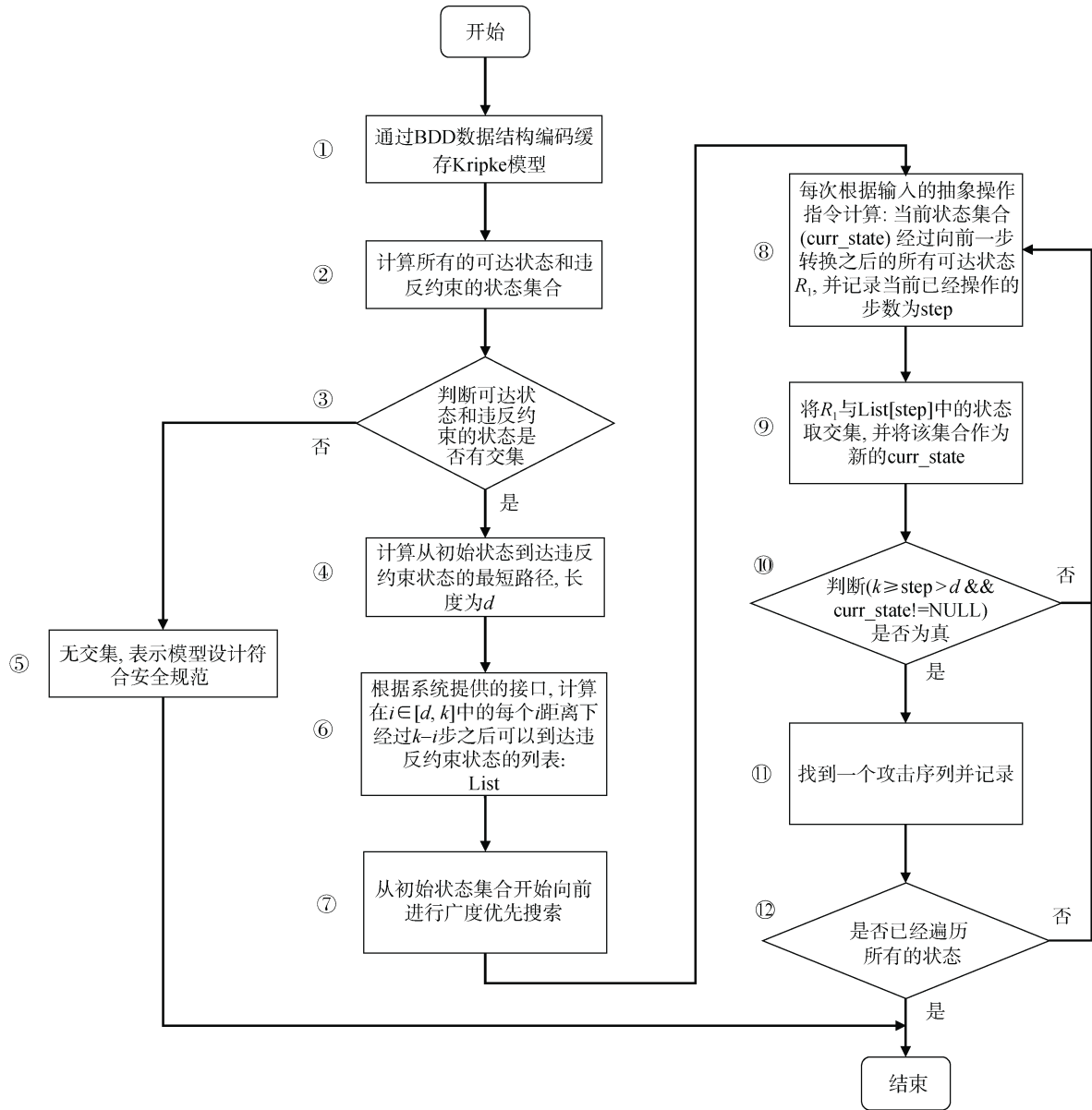


图5 反例搜索算法流程图

Figure 5 Flowchart of the counterexample search algorithm

步为通过一个队列实现针对状态的广度优先搜索算法, 第7步是将在距离 d 处的状态推入队列。第8步相当于将所有状态根据输入的抽象指令执行一步状态转换操作。第9步是将该层的状态集合和对应步数的状态列表 $List$ 取交集, 得到的集合就必然是通过在限定步数内的转换可以到达违反约束状态的集合。将该新得到的集合作为 $curr_state$ 重新推入队列。第10步为判断是否找到了反例, 如果当前状态转移的步数在 d 和 k 的范围内, 而且 $curr_state$ 不为空, 说明已经找到了一条到达违反约束状态的攻击序列, 在第11步将其记录下来。之后第12步判断广度优先搜索是否完成, 即状态转换的步数是否超过了最远可达距离。如果还未完成遍历系统中的所有状态,

则回到第8步继续进行搜索, 直到遍历了系统中的所有状态后结束。

5 形式化验证结果及分析

我们在所建立的缓存模型上运行模型检验工具, 其中违反定义7的指令序列被认为是反例。

5.1 普通非包含缓存模型验证结果

表2中所列取的是针对普通非包含缓存的验证结果。表中的第一列是该攻击的类型, 第二列为抽象表示的攻击序列, 第三列显示该攻击类型为现已存在的攻击还是我们所发现的新攻击。从表中可以看出我们的验证结果可以包括现有的跨核 Flush+ Reload 和跨核的 Flush+Flush 攻击, 而且也包括仅依靠

表 2 普通非包含缓存的攻击序列

Table 2 Attack sequence of normal non-inclusive cache

攻击类型	攻击序列	是否为新类型的攻击
Flush+Reload	$A_a^{inv} \rightarrow V_u \rightarrow A_a$	
	$Invany \rightarrow V_u \rightarrow A_a$	
Flush+Flush	$A_a^{inv} \rightarrow V_u \rightarrow A_a^{inv}$	现有攻击
Bernstein attack	$V_u \rightarrow V_a \rightarrow V_u$	
Prime+Prime+Probe	$A_d \rightarrow V_d \rightarrow V_u \rightarrow A_d$	新攻击
Evict+Evict+Reload	$V_d \rightarrow A_d \rightarrow V_u \rightarrow A_d$	
Evict+Evict+Flush	$V_d \rightarrow A_d \rightarrow V_u \rightarrow A_a^{inv}$	
Evict+Evict+Time	$V_u \rightarrow V_d \rightarrow A_d \rightarrow V_u$	

内部干涉发动的 Bernstein 攻击。同时, 验证结果也表明 Prime+Probe 类攻击在非包含缓存中只能在同核心实现, 但是并不能实现跨核心的攻击。此外, 我们也发现了一些更复杂的需要至少四步操作才能完成攻击的类型。

下面我们选择其中 3 个典型的案例进行分析。

(1) Prime+Prime+Probe 攻击: 该类攻击在真实世界的实现最早出现于 ARM 处理器。首先攻击者访问驱逐集地址, 将 LLC 中某个 set 中的敏感数据驱逐出去。因为在非包含缓存中, LLC 中的敏感数据被驱逐不会导致受害者私有缓存中的敏感数据失效。所以此时空闲一段时间, 等待受害者核心中的其他系统活动访问私有缓存并将秘密数据驱逐。攻击者重复多次访问驱逐集, 此时敏感数据将不会存在于缓存中。等待受害者访问秘密地址 u 之后, 攻击者再次访问驱逐集。如果受害者访问了 u , 则秘密数据会存在于 LLC 中, 此时再次访问驱逐集时间变慢。后面的 Evict+Evict+Reload 和 Evict+Evict+Flush 类的攻击也是相似的原理。

(2) Evict+Evict+Flush 攻击: 该攻击的第一步为利用受害者核心内的系统活动将探测地址 a 从受害者私有缓存中驱逐到 LLC 中。之后第二步攻击者访问对应 cache set 的驱逐集, 将该探测地址驱逐出缓存。在实际攻击中, 前两步需要交替进行多次。第三步等待受害者的操作。如果受害者访问了目标地址 u , 那么探测地址 a 就会被加载回缓存中, 否则地址 a 不在缓存中。之后在第四步攻击者调用 clflush 指令刷除探测地址 a 时, 如果 a 地址在缓存中, 则该刷除过程的延时更长。反之若目标地址不在缓存中, 则该刷除过程的延时更短。

(3) Evict+Evict+Time 攻击: 首先第一步执行受害者的进程, 此时机密数据存在于私有缓存和 LLC

当中。之后等待受害者核心的系统活动将秘密数据驱逐, 同时攻击者核心重复访问驱逐集即可将机密数据驱逐出缓存。最后再次执行受害者进程, 如果 u 和 d 所在的 set 相等, 则访问时间更慢。反之, 攻击者的驱逐集访问不会将机密数据从 LLC 中驱逐, 下次受害者执行会更快。

上述的结果都证明了即使是非包含缓存也依然可能会受到基于争用的侧信道攻击, 这也说明了那些基于非包含属性的安全缓存设计^[29]在某些情况下并不是绝对安全的。

5.2 缓存一致性模型的验证结果

表 3 是我们针对考虑缓存一致性的多级缓存的验证结果。

下面我们将选择其中 3 个典型案例, 具体阐述所发现的新类型攻击的操作过程。

表 3 利用缓存一致性的攻击序列

Table 3 Attack sequence of exploiting cache coherence

涉及核心数	攻击类型	是否为新类型的攻击
双核	$A_a^{inv} \rightarrow A_a \rightarrow V_u \rightarrow A_a^w$	新攻击
	$A_a^{inv} \rightarrow A_a \rightarrow V_u^w \rightarrow A_a$	新攻击
	$A_a^{inv} \rightarrow A_a \rightarrow V_u^w \rightarrow A_a^w$	
	$A_a \rightarrow V_u^w \rightarrow A_a$	
	$A_a^{inv} \rightarrow V_u \rightarrow A_a^{inv}$	
	$A_a^{inv} \rightarrow V_u \rightarrow A_a^w$	新攻击
	$A_a^{inv} \rightarrow V_u^w \rightarrow A_a^{inv}$	
	$A_a^{inv} \rightarrow V_u^w \rightarrow A_a^w$	
	$A_a^w \rightarrow V_u \rightarrow A_a^w$	
	$A_a^w \rightarrow V_u^w \rightarrow A_a$	
三核	$T_a^w \rightarrow V_u \rightarrow A_a$	新攻击
	$T_a^w \rightarrow V_u^w \rightarrow T_a \rightarrow A_a$	
	$A_a \rightarrow V_u^w \rightarrow T_a \rightarrow A_a$	新攻击
	$A_a^{inv} \rightarrow V_u^w \rightarrow T_a \rightarrow A_a$	
	$A_a^{inv} \rightarrow T_a \rightarrow V_u \rightarrow A_a$	新攻击
	$A_a^w \rightarrow V_u^w \rightarrow T_a \rightarrow A_a$	

(1) $A_a^{inv} \rightarrow T_a \rightarrow V_u \rightarrow A_a$: 它的攻击流程为首先将目标数据在处理器中的所有副本全部刷除, 此时目标数据在处理器的缓存中都是无效的状态。第二步, 使用木马核心访问该目标地址。第三步为等待受害者进行访问。此时如果受害者访问的地址 u 等于

地址 a , 则受害者和木马核心的一致性状态为 **Shared**。如果没有访问, 则受害者核心的状态为 **Invalid**, 木马核心状态为 **Exclusive**。最后由攻击者核心探测目标地址, 如果是 **Shared** 状态, 则从 LLC 中提取数据。如果地址是 **Exclusive** 状态, 则需要从木马所在核心的私有缓存中跨核提取数据, 这两者会产生明显可分辨的时间差异。该攻击所利用的原理是前文介绍的 CPU 所存在的“静默升级”机制。

(2) $A_a^{\text{inv}} \rightarrow V_u^w \rightarrow T_a \rightarrow A_a$ 。该攻击所利用的原理和上述的类似。即: 第一步, 将目标数据在缓存中的所有副本都刷除。第二步, 等待受害者进行瞬态执行窗口中的写操作, 这一步操作完成之后会将目标地址加载到受害者的私有缓存中, 同时将其其他核心中的目标数据全部失效。第三步, 木马核心访问目标地址。此时如果受害者执行了瞬态执行写的操作, 则当前共享数据的一致性状态为 **Shared**。如果受害者没有执行瞬态写操作则当前共享数据是 **Exclusive** 状态。第四步, 攻击者核心访问该数据, 通过计时判断该数据是从 LLC 中访问还是跨核的私有缓存中访问, 从而推断出受害者是否访问了目标地址。该攻击主要用于在瞬态窗口中将私密信息传递出来。

(3) $A_a^{\text{inv}} \rightarrow A_a \rightarrow V_u^w \rightarrow A_a^w$ 。该攻击流程为: 第一步, 将目标数据在缓存中的所有副本都刷除。第二步, 攻击者核心访问目标地址, 此时该地址仅存在于攻击者私有核心中。第三步, 等待受害者核心在瞬态执行窗口中对目标地址进行修改。如果受害者修改了目标地址, 则攻击者核心的数据就会失效。若没有修改目标地址中的数据, 此时攻击者核心中的数据是有效的, 但是受害者核心中的数据是失效的。第四步, 攻击者执行“预取”指令并计时。如果受害者没有修改目标数据, 则预取的时间会更快, 否则预取的时间更慢。这样就可以构成一个侧信道。

(4) $T_a^w \rightarrow V_u^w \rightarrow T_a \rightarrow A_a$: 该攻击同时利用了预取机制和“静默升级”机制。攻击的具体步骤为: 第一步, 由木马核心对探测地址 a 执行预取操作, 此时其他所有核心的数据副本都被失效, 该数据仅存在于木马私有核心中。第二步, 等待受害者在瞬态窗口中的写入操作或者受害者不执行任何操作。如果受害者没有执行写入操作, 则 a 地址依然仅存在于木马核心中。如果受害者执行了写入操作, 则 a 地址仅存在于受害者核心中, 而木马核心的 a 地址失效。第三步, 操控木马核心再次访问地址 a 。根据第二步的执行结果, 地址 a 有两种状态: 仅存在于木马核心或者同时存在于木马核心和受害者核心。因此, 在第四

步攻击者访问地址 a 时, 由于“静默升级”机制的存在, 这两种状态会产生不同的延迟。

5.3 验证结果的讨论

本文所提出的验证框架对包含细粒度特征的缓存微架构进行了建模, 因此能够发现多种利用缓存细粒度特征的侧信道攻击。它的优势在于部分新发现的攻击序列能够在攻击场景受限的情况下完成攻击。由于随着缓存侧信道威胁日益严重, 用户可能会部署一些防御措施。例如: 禁止普通权限的用户使用 `clflush` 指令或限制攻击者和受害者进程间的页面共享。对于前者, 本文发现的 $T_a^w \rightarrow V_u^w \rightarrow T_a \rightarrow A_a$ 攻击序列能够绕过这种防御方式。而本文发现的 **Evict+Evict+Time** 攻击序列能够绕过后者的防御。本文发现的多种攻击序列能够给攻击者提供更多的攻击选择, 也能够帮助受害者检验防御措施是否完善。

此外, 本文发现的利用缓存细粒度特征的侧信道攻击并不会增加其具体实现的难度。攻击者只需要将抽象指令替换为具体指令并附加一些必要的基础代码(如计时、内存屏障、进程间同步等)即可实现攻击。将抽象指令替换为具体指令的过程也并不复杂, 大多数抽象指令类型只需要一条具体指令即可实现, 例如: 访存、预取、刷除等。涉及构造驱逐集的抽象指令(如 A_d)的实现略微复杂, 但目前其他的研究工作中已经提出了成熟的驱逐集构造方法^[30]。

此外, 木马进程的实现和攻击者进程类似, 它们都由纯软件构成, 我们可以根据抽象指令序列(符号 T 相关的指令表示木马进程的操作)来构造木马进程。在攻击过程中, 需要攻击者实现进程的调度, 以保证攻击者、受害者和木马进程按照抽象攻击序列的顺序来执行。

5.4 与现有形式化验证框架的对比

表 4 展示了不同的验证或者测试工作所发现缓存侧信道的种数, 在单层缓存侧信道攻击方面, 本文和 **CacheGuard** 都能够覆盖 24 种侧信道攻击, 而“三步模型”则缺失了 3 种。**CheckMate** 没有考虑单层缓存模型, 因此不予对比。在细粒度缓存侧信道方面, 我们主要考察缓存的三个微架构特征: 非包含、缓存一致性和预取。其中, “三步模型”和 **CheckMate** 分别能够发现 2 种和 1 种基于缓存一致性的攻击, 这里需要注意的是: 虽然“三步模型”在论文中声称有更多种攻击, 但这些攻击依赖于受害者的内部干涉, 因此攻击局限性较大, 我们暂不统计这种情况。

在细粒度缓存侧信道验证中, 对非包含属性缓存的验证得到 7 种攻击序列(如表 2 中所展示), 利用

表 4 不同验证框架检测侧信道能力的对比

Table 4 Comparison of side-channel discovery capabilities of different verification frameworks

验证或测试框架	单层缓存 侧信道	细粒度缓存侧信道		预取
		非包含缓存	缓存一致性	
三步模型 ^[18]	21	0	2	0
CacheGuard ^[20]	24	0	0	0
CheckMate ^[31]	/	0	1	0
本文	24	7	17	10

缓存一致性攻击的侧信道为 17 种(如表 3 中所展示)。利用预取机制的缓存侧信道数量为 10 种, 它们是包含在利用缓存一致性攻击的侧信道中的。这是因为预取指令会改变数据的一致性状态, 它的执行时间也会受到一致性状态的影响, 因此利用预取属性的侧信道攻击必然也是利用缓存一致性的。由此可见, 由于本文形式化模型包含了缓存的细粒度特征, 因此所发现的攻击类型更加全面。

5.5 在真实处理器上的测试结果

为证明在抽象后的形式化模型上验证得到的攻击序列的有效性, 我们在真实的处理器上进行测试。由于本文验证得到的攻击序列所利用的处理器特征具有相同性, 因此我们对验证结果的其中一个案例: $A_a^{inv} \rightarrow A_a \rightarrow V_u^w \rightarrow A_u^w$ 进行实机测试并分析。根据该类型漏洞一般的命名规范, 我们将该侧信道变种命名为 SpectrePrefetch。实验测试的伪代码如表 5 所示。为了方便展示, 本 PoC 的实现为同进程间的跨核攻击, 但是通过引入进程间同步的一些技术, 也可以拓展为跨进程跨核的攻击。

我们将上述的代码在实际处理器上实现并测试, 该机器型号为 Intel Core i7-7800X 微架构为 Skylake-X, 操作系统为 Ubuntu18.04。该攻击结合了 Spectre 类攻击的原理, 能够将瞬态执行窗口中读取到的私密数据跨核心传输。因此在实验中需要将瞬态执行部分的代码, 即 `victim_function`, 绑定到和主进程不同的物理核上。这样在瞬态执行窗口的写操作才可以将其他物理核心的数据副本失效, 之后再通过比较预取指令的执行时间来判断受害者访问的私密数据的值。

在伪代码中的第 9 行定义了一个数组 `array2`, 该数组为受害者和攻击者共享内存地址的数组, 用来作为隐蔽信道传输瞬态执行窗口中泄露出的私密数据。在第 10 行定义的 `array1` 是用来索引私密数据的数组, 在索引值小于 16 时的访问为合法访问, 当索引值大于等于 16 时的访问为非法访问。

在代码的 21~23 行为毒化分支预测器的过程,

通过多次的合法访问会让分支预测器认为下一次的访问依然是合法的访问, 因此就会在分支判断还未结束的时候, 推测执行下面的语句。在第 26 行即为毒化结束后, 开始推测执行 `array1` 范围之外的数据, 通过将 `secret` 指针的地址和 `array1` 的首地址相减可以得到秘密数据相对于 `array1` 首地址的偏移量, 该偏移量可以用来索引秘密数据。

表 5 SpectrePrefetch 攻击的伪代码

Table 5 Pseudo code of SpectrePrefetch attack

Algorithm 1: Pseudo code of SpectrePrefetch attack	
0	function <code>victim_function(addr)</code> {
1	/* pin victim thread on other core that different from
2	main procedure. */
3	<code>sched_setaffinity(0, other_core);</code>
4	if (<code>addr < array1_size</code>)
5	<code>array2[array1[addr] * 512] = 1;</code>
6	}
7	
8	function <code>main()</code> {
9	int <code>array2[256 * 512];</code>
10	int <code>array1[16] = {1};</code>
11	int <code>junk;</code>
12	uint64_t <code>time, latency;</code>
13	char * <code>secret = "S";</code>
14	
15	/* prime the array2 */
16	for (int <code>i = 0; i < 256; i++</code>) {
17	<code>junk += array2[i * 512];</code>
18	}
19	
20	/* begin to poison the branch predictor. */
21	for (int <code>i = 0; i < 9; i++</code>) {
22	<code>victim_function(0);</code>
23	}
24	
25	/* speculative access to secret data. */
26	<code>victim_function(secret - (char*)array1);</code>
27	
28	/* determine the secret data just visited by timing the
29	prefetch operation of array2. */
30	for (int <code>i = 0; i < 256; i++</code>) {
31	<code>time = rdtscp();</code>
32	<code>__builtin_prefetch(array2[i * 512]);</code>
33	<code>latency = rdtscp() - time;</code>
34	if (<code>latency > THRESHOLD</code>) {
35	/* the current i was just visited that equal
36	to the secret 'S'. */
37	} else {
38	/* the current i has not been visited. */
39	}
40	}
41	return 0;
	}

在代码的 16~18 行和 30~39 行为隐蔽信道的实现部分, 通过缓存隐蔽信道可以将瞬态窗口读到的秘密数据传递到架构层从而被攻击者可见。其中 16~18 行的作用是遍历 `array2` 中的数据, 确保 `array2`

中对应地址的数据都被加载到缓存中。30~39 行的作用是再次通过 `prefetch` 遍历执行 `array2` 中对应地址的数据并通过 `rdtscp` 指令记录 `prefetch` 指令的执行周期。其中阈值 `THRESHOLD` 经过提前的标定校准设定为 80 个周期。如果执行时间小于 `THRESHOLD`, 则说明该地址的数据是有效的状态, 即该地址并没有被受害者访问。如果延迟长于 `THRESHOLD`, 则说明受害者在瞬态执行中访问了该地址, 并由于缓存一致性的作用将该地址的数据反向失效了。至此, 就可以通过访问时间的不同来确定私密数据的值。

我们将上述的 PoC 在实体机器上测试 100 轮并统计其测试结果, 我们观察到 `SpectrePrefetch` 漏洞可以以 21.4B/s 的速度泄露数据, 平均误码率约为 3.5%。该测试均在未对测试代码进行专门速度优化的情况下进行。同时我们还测试了在分支指令后面加入 `mfence` 或 `lfence` 之后的情况(即在代码的第 3 行后加入这两条指令), 结果表明加入了 `mfence` 或者 `lfence` 等序列化指令之后, 我们将无法读取私密数据。其中的原因是序列化指令防止了程序的推测执行, 这意味着条件判断后的指令必须等待判断结果被执行完毕后才能执行。值得注意的是: 添加序列化指令虽然能够防御上述攻击, 但基于缓存一致性使其他核心数据失效的这一原理依然可以用于其他的瞬态执行攻击或者侧信道攻击当中。

6 总结

本文介绍了一种对利用缓存非包含属性和缓存一致性特征的侧信道的形式化建模方案与验证框架, 详细阐述了对缓存微架构和相关指令的抽象建模方法。本框架通过延迟差分比较的验证方法来检测缓存侧信道, 并定义了相关安全约束。之后, 通过在现有模型检验工具 `NuSMV` 的基础上拓展实现了反例搜索算法, 使其可以输出所有反例。使用该验证框架对 `Intel Skylake-X` 微架构的非包含缓存架构和考虑缓存一致性的多核缓存进行了验证, 验证输出的反例不仅可以覆盖现有的攻击方式, 还找到了九种新攻击方式。最后在真实处理器上对其中一个典型案例 `SpectrePrefetch` 进行了测试, 结果证明了我们所发现新攻击的有效性。

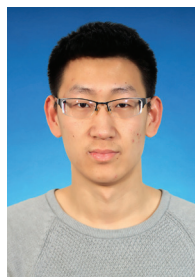
本文所提出的验证框架具备可迁移性和通用性, 它不仅可以用来挖掘现有缓存结构中潜在的侧信道攻击, 而且处理器微架构设计者可以用本文提出的方法在整个设计流程的早期阶段对模型进行原型验证, 从而根据验证输出的结果对微架构设计在侧信道方面的安全性进行全面的评估。未来考虑将本文

的验证框架推广到处理器中所有的微架构组件, 例如: 分支预测器、TLB 以及流水线上的各部件, 为解决目前处理器中普遍存在的硬件安全漏洞提供一种新的解决方案。

参考文献

- [1] Osvik D A, Shamir A, Tromer E. Cache Attacks and Countermeasures: The Case of AES[C]. *Topics in Cryptology – CT-RSA 2006*, 2006: 1-20.
- [2] Yarom Y, Genkin D, Heninger N. CacheBleed: A Timing Attack on OpenSSL Constant-Time RSA[J]. *Journal of Cryptographic Engineering*, 2017, 7(2): 99-112.
- [3] Dai M, Paccagnella R, Gomez-Garcia M, et al. Don't Mesh Around: {Side-Channel} Attacks and Mitigations on Mesh Interconnects[C]. *31st USENIX Security Symposium*, 2022: 2857-2874.
- [4] Paccagnella R, Luo L, Fletcher C W. Lord of the ring (s): Side channel attacks on the {CPU}{On-Chip} ring interconnect are practical[C]. *30th USENIX Security Symposium*, 2021: 645-662.
- [5] Wan J P, Bi Y X, Zhou Z, et al. MeshUp: Stateless Cache Side-Channel Attack on CPU Mesh[C]. *2022 IEEE Symposium on Security and Privacy*, 2022. DOI:10.1109/sp46214.2022.9833794.
- [6] Gruss D, Maurice C, Fogh A, et al. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR[C]. *The 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016. DOI:10.1145/2976749.2978356.
- [7] Lipp M, Hadžić V, Schwarz M, et al. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors[C]. *The 15th ACM Asia Conference on Computer and Communications Security*, 2020. DOI:10.1145/3320269.3384746.
- [8] Kocher P, Horn J, Fogh A, et al. Spectre Attacks: Exploiting Speculative Execution[C]. *2019 IEEE Symposium on Security and Privacy*, 2019. DOI:10.1109/sp.2019.00002.
- [9] Lipp M, Schwarz M, Gruss D, et al. Meltdown: Reading Kernel Memory from User Space[C/OL]. *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018: 973-990. <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>.
- [10] van Schaik S, Milburn A, Osterlund S, et al. RIDL: Rogue In-Flight Data Load[C]. *2019 IEEE Symposium on Security and Privacy*, 2019. DOI:10.1109/sp.2019.00087.
- [11] Canella C, Van Bulck J, Schwarz M, et al. A systematic evaluation of transient execution attacks and defenses[C]. *28th USENIX Security Symposium (USENIX Security 19)*. 2019: 249-266.
- [12] Ragab H, Milburn A, Razavi K, et al. CrossTalk: Speculative Data Leaks across Cores Are Real[C]. *2021 IEEE Symposium on Security and Privacy*, 2021. DOI:10.1109/sp40001.2021.00020.
- [13] Schwarz M, Lipp M, Moghimi D, et al. ZombieLoad: Cross-Privilege-Boundary Data Sampling[C]. *The 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019. DOI:10.1145/3319535.3354252.
- [14] Liu F F, Yarom Y, Ge Q, et al. Last-Level Cache Side-Channel Attacks Are Practical[C]. *2015 IEEE Symposium on Security and Privacy*, 2015. DOI:10.1109/sp.2015.43.
- [15] Yao F, Doroslovacki M, Venkataramani G. Are Coherence Protocol

- States Vulnerable to Information Leakage? [C]. *2018 IEEE International Symposium on High Performance Computer Architecture*, 2018. DOI:10.1109/hpca.2018.00024.
- [16] Guo Y N, Zigerelli A, Zhang Y T, et al. Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks[C]. *2022 IEEE Symposium on Security and Privacy*, 2022. DOI:10.1109/sp46214.2022.9833692.
- [17] Lipp M, Gruss D, Spreitzer R, et al. {ARMageddon}: Cache Attacks on Mobile Devices[C]. *25th USENIX Security Symposium*, 2016: 549-564.
- [18] Deng S W, Xiong W J, Szefer J. Analysis of Secure Caches Using a Three-Step Model for Timing-Based Attacks[J]. *Journal of Hardware and Systems Security*, 2019, 3(4): 397-425.
- [19] Genkin D, Kosasih W, Liu F, et al. Cachefix: A framework for evaluating cache security[J]. *arXiv preprint arXiv:2201.11377*, 2022.
- [20] Xu Z H, Yin L F, Lyu Y Q, et al. CacheGuard: A Behavior Model Checker for Cache Timing Side-Channel Security: (Invited Paper)[C]. *2022 27th Asia and South Pacific Design Automation Conference*, 2022. DOI:10.1109/asp-dac52403.2022.9712560.
- [21] Clarke E M. Model Checking[M]. *Foundations of Software Technology and Theoretical Computer Science*, 1997: 54-56.
- [22] Yarom Y, Falkner K. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack[C]. *23rd {USENIX} Security Symposium*, 2014: 719-732.
- [23] Gruss D, Maurice C, Wagner K, et al. Flush+Flush: A Fast and Stealthy Cache Attack[C]. *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016: 279-299.
- [24] Gruss D, Spreitzer R, Mangard S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches[C]. *USENIX Security Symposium*, 2015: 897-912.
- [25] Miao C L, Bu K, Li M M, et al. SwiftDir: Secure Cache Coherence without Overprotection[C]. *2022 55th IEEE/ACM International Symposium on Microarchitecture*, 2022. DOI:10.1109/micro56248.2022.00052.
- [26] Burch J R, Clarke E M, McMillan K L, et al. Symbolic Model Checking: 1020 States and beyond[J]. *Information and Computation*, 1992, 98(2): 142-170.
- [27] Cimatti A, Clarke E, Giunchiglia E, et al. NuSMV 2: An Open-Source Tool for Symbolic Model Checking[C]. *Computer Aided Verification*, 2002: 359-364.
- [28] Clarke E, McMillan K, Campos S, et al. Symbolic Model Checking[M]. *Computer Aided Verification*, 1996: 419-422.
- [29] Kayaalp M, Khasawneh K N, Esfeden H A, et al. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks[C]. *The 54th Annual Design Automation Conference 2017*, 2017. DOI:10.1145/3061639.3062313.
- [30] Vila P, Kopf B, Morales J F. Theory and Practice of Finding Eviction Sets[C]. *2019 IEEE Symposium on Security and Privacy*, 2019. DOI:10.1109/sp.2019.00042.
- [31] Trippel C, Lustig D, Martonosi M. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests[C]. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018. DOI:10.1109/micro.2018.00081.



张士轩 于 2021 年在哈尔滨工业大学电子科学与技术系获得硕士学位。现在哈尔滨工业大学(深圳)电子信息专业攻读博士学位。主要研究领域为形式化验证、硬件漏洞挖掘。Email: sxzhang@stu.hit.edu.cn



王海霞 于 2004 年在中国科学院计算技术研究所计算机系统结构专业获得博士学位。现任清华大学副研究员。CCF 高级会员。主要研究领域为计算机系统和数据可靠性、算法加速和形式化验证。Email: hx-wang@tsinghua.edu.cn



邱朋飞 于 2020 年在清华大学计算机系获得博士学位。现任北京邮电大学副研究员。主要研究领域为计算机硬件安全。Email: qpf@bupt.edu.cn



吕勇强 于 2006 年在清华大学计算机系获得博士学位。现任清华大学副研究员。CCF 高级会员。主要研究领域为处理器漏洞挖掘、处理器安全验证与设计、硬件安全。Email: luyq@tsinghua.edu.cn



尹凌峰 于 2021 年在清华大学计算机系获得学士学位。现在清华大学计算机系攻读硕士学位。主要研究领域为安全硬件设计与验证。Email: ylf17@mails.tsinghua.edu.cn



王鸿鹏 于 2001 年 4 月于哈尔滨工业大学计算机学院获得博士学位。现任哈尔滨工业大学(深圳)计算机科学与技术学院教授。CCF 高级会员。主要研究领域为计算机系统结构、人工智能。Email: wanghp@hit.edu.cn



汪东升 于 1995 年在哈尔滨工业大学计算机科学与技术专业获博士学位。现任清华大学计算机系教授。主要研究领域为计算机体系结构、计算机系统安全与安全系统。Email: wds@tsinghua.edu.cn