Arm 架构的分支预测器隐蔽信道研究

杨 毅[1], 吴凭飞[2], 邱朋飞[1], 王春露[1], 赵路坦[3], 张锋巍[4], 王 博[5],
吕勇强[6], 王海霞[6], 汪东升[2,7]

[1]北京邮电大学可信分布式计算与服务教育部重点实验室 北京 中国 100867
[2]清华大学计算机科学与技术系 北京 中国 100084
[3]中国科学院信息工程研究所 北京 中国 100093
[4]南方科技大学计算机科学与工程系 深圳 中国 518055
[5]飞腾信息技术有限公司 天津 中国 300459
[6]清华大学北京信息科学与技术国家研究中心 北京 中国 100084
[7]中关村实验室 北京 中国 100094

摘要 隐蔽信道是一种在不违背计算机当前安全策略的前提下, 在进程间传递信息的攻击方式。共两个进程参与到隐蔽信道的构建中: 木马进程和间谍进程, 具有高权限的木马进程通过隐蔽信道向低权限的间谍进程传递信息以完成攻击。隐蔽信道的传输介质种类很多, 如时间、功耗、温度等。在现代处理器中, 分支预测器作为重要的微架构组件, 有效提高了处理器的流水线效率, 但由于分支预测器在核内的多进程间共享, 使得其存在被用于构建隐蔽信道的风险。目前 Intel x86 架构已被发现存在基于分支预测器的隐蔽信道攻击, 但是 Arm 架构是否存在相似的攻击还没有得到充分的研究。本文中, 我们成功在 Arm 架构的实际硬件平台上构建了三种基于分支预测器的隐蔽信道。首先我们在 Arm 架构下设计并实现了类似于 x86 架构下的基于分支预测器的隐蔽信道 CC 和 RSC, 其次我们发现了一个新的基于分支预测组件 BTB 的隐蔽信道 BTBC。我们评估并分析了隐蔽信道参数对信道性能的影响及其成因, 并给出参数设置建议。在 Cortex-A53 及 Cortex-A72 两种核心上, 我们对三种隐蔽信道的信号特性、传输速率和误码率进行了测试和对比分析。实验表明在实际的 Arm 架构硬件平台下, BTBC 的传输信号边缘清晰, 震荡幅度小。在连续传输数据时表现出与 CC 和 RSC 近似的信道性能, 并且在两种核心上均可以低误码率进行数据传输, 其在 200bps 的传输速率下, 仅有 2%的误码率。最后我们还给出了对于此类隐蔽信道的防御措施。

关键词 Arm 架构; 分支预测器; 隐蔽信道
中图法分类号 TP309.1 DOI 号 10.19363/J.cnki.cn10-1380/tn.2025.01.01

Covert Channel of Branch Predictor on Arm Processor

YANG Yi[1], WU Pingfei[2], QIU Pengfei[1], WANG Chunlu[1], ZHAO Lutan[3], ZHANG Fengwei[4]
WANG Bo[5], LYU Yongqiang[6], WANG Haixia[6], WANG Dongsheng[2,7]

[1] Ministry of Education Key Laboratory of Trustworthy Distributed Computing and Service, Beijing University of Posts and Telecommunications, Beijing 100867, China
[2] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
[3] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[4] Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China
[5] Phytium Technology Co. Ltd., Tianjin 300459, China
[6] Beijing National Research Center for Information Science and Technology, Tsinghua University, Beijing 100084, China
[7] Zhongguancun Laboratory, Beijing 100094, China

Abstract The covert channel is an attack that transmits information between processes which is not allowed under the current security policy. Two processes are involved in constructing the covert channel: trojan and spy. The trojan process with high privilege transmits information to the spy process with low privilege through the covert channel to complete the entire attack. There are many types of transmission media for covert channels, such as time, power consumption, temperature, and so on. In modern processors, the branch predictor is an important microarchitecture component, which effectively improves pipeline efficiency. However, since the branch predictor is shared among multiple processes in single core, it has the potential risk of constructing covert channel. At present, the Intel x86 architecture has been found to have covert channel attacks based on branch predictors, but whether there are similar attacks on the Arm architecture has not been fully

通讯作者: 吕勇强, 博士, 副研究员, Email: luyq@tsinghua.edu.cn。
本课题得到国家重点研发计划(No. 2021YFB3100902)和国家自然科学基金(No. 62072263, No. 62102214)资助。
收稿日期: 2023-02-07; 修改日期: 2023-04-27; 定稿日期: 2024-11-19

studied. In this paper, we successfully build three branch predictor based covert channels on the actual hardware platform of the Arm architecture. First, we design and implement a branch predictor based covert channel CC and RSC similar to the x86 architecture under the Arm architecture, and second, we discover a new covert channel BTBC based on the branch prediction component BTB. We evaluate and analyze the impact of covert channel parameters on channel performance and its causes, and give recommendations for parameter settings. On the two cores of Cortex-A53 and Cortex-A72, we test and compare the signal characteristics, transmission rate and SER of three covert channels. The result shows that transmission signal of BTBC has clear edges and small oscillation amplitude. BTBC has similar channel performance to CC and RSC under the actual Arm architecture hardware platform, and can transmit data with a low SER on both cores. When the transmission capacity is 200bps, the SER is only 2%. Finally, we also give the defense measures against such covert channels.

Key words Arm architecture; branch predictor; covert channel

1 引言

现代处理器通过指令流水线技术提升处理器的指令执行并行度, 进而提升执行效率。但是由于在流水线执行时, 指令间可能存在结构冒险、数据冒险及控制冒险, 从而导致流水线阻塞[1]。因此, 现代处理器引入了多种微架构组件用于缓解上述冒险所带来的影响。其中一种典型的微架构组件为分支预测器, 其用于对分支指令的跳转方向及目的地址进行预测。

在现代处理器上, 一般一个核心拥有一个分支预测单元, 核上的不同权限等级的应用使用同一个分支预测单元。由于当前的处理器设计时更多地考虑分支预测器的高性能, 默认将分支预测器视作一个对上层透明的组件, 而没有考虑到这种跨进程的共用可能会在分支预测器上留下痕迹, 从而可能造成私密数据的泄露。一些研究者利用分支预测器跨进程、跨 SMT(Simultaneous multithreading, 同步多线程, 即在同一个物理核上运行多个逻辑核的技术)的特性在 Intel 的处理器上实现了隐蔽信道以及侧信道。Casen Hunger 等人[2]在 2015 年时提出了基于竞争的分支预测器隐蔽信道 CC(Contention-based Covert channel), 为首个基于分支预测器的隐蔽信道, 之后 Dmitry Evtyushkin 等人[3]于 2016 年时提出了基于残留状态的分支预测器隐蔽信道 RSC(Residual State-based Covert channel)。而基于分支预测器的侧信道提出时间则更早, Onur Aciicmez[4]在 2007 年时提出了第一个基于分支预测器中的 BTB 组件构造的侧信道, 其后还涌现出了如 branch shadowing[5], template attack[6]等基于分支预测器的侧信道。但是上述工作均在 Intel 上实现, 即 x86 架构, 而在嵌入式以及移动端使用更加广泛的 Arm 架构, 则少有隐蔽信道的研究, 因此针对 Arm 架构的分支预测器隐蔽信道的研究具有其重要意义。

在本文中, 我们首先对 Arm 架构的分支预测器进行分析, 并根据分支预测器组件模式历史表(Pattern History Table, PHT)的机制, 在 Arm 架构的硬件平台上进行跨平台的迁移, 并验证了两种存在于 x86 平台的隐蔽信道 CC 与 RSC。

除此之外, 我们在分析 Arm 架构的分支预测机制后, 发现与分支目标地址预测相关的分支目标地址缓存(Branch Target Address Cache, BTAC)在上下文切换时不进行刷新, 由此我们可以通过一个进程的执行流来保留或淘汰另一个进程的相关 BTAC 表项。我们利用该机制首次在 Arm 架构上发现了一种新的基于分支预测器 BTB 组件的隐蔽信道 BTBC(BTB-based Covert channel)。我们测试了大量参数设置, 评估并分析了不同参数设置对上述三个隐蔽信道的性能影响以及其成因, 给出参数设置建议。我们在对以上三种隐蔽信道进行信号振幅、信号边缘以及传输效率的评估后, 实验表明我们发现的 BTBC 拥有与 CC 和 RSC 近似的信道性能。

最后我们针对基于分支预测器的隐蔽信道给出了相应的防御措施建议。

本文的主要贡献有:

1) 验证了 Arm 架构的硬件平台上也存在 x86 平台上的 CC 与 RSC 隐蔽信道。

2) 首次在 Arm 架构发现了一种新的基于分支预测器 BTB 组件的隐蔽信道 BTBC。利用木马的不同行为淘汰或保留 BTB 表项, 以此控制 BTB 状态, 并构造隐蔽信道。

3) 评估并分析了隐蔽信道参数对信道性能的影响及其成因, 并给出参数设置建议。

4) 对这三种隐蔽信道的信号振幅、信号边缘以及传输效率进行了实验评估。实验表明, 我们发现的 BTBC 表现出与 CC 和 RSC 近似的信道性能。

5) 针对三种基于分支预测器的隐蔽信道给出了防御措施的建议。

本文后续组织如下: 第二章介绍分支预测器设计与隐蔽信道原理; 第三章描述基于分支预测器的三种隐蔽信道的基本原理与伪代码设计; 第四章包括三种隐蔽信道在 Arm 架构平台的实验设计, 参数

设置评估与分析, 信号特性、传输速率评估的结果与分析; 第五章给出了对于该类隐蔽信道的防御措施; 第六章总结全文工作并对后续工作进行展望。

2 背景与相关工作

2.1 分支预测器设计

在程序设计中, 分支指令是一种非常常见的指令, 据统计, 常规的代码中每 4～5 条指令就会有一条分支指令[7]。按照其是否附带跳转条件分为条件分支和无条件分支, 条件分支需要按照其跳转条件确定跳转与否, 而无条件分支则无论在任何情况下都直接跳转; 按照其分支目标地址是否直接硬编码于代码中, 将分支分为直接分支与间接分支, 直接分支在跳转时跳转至代码中指定的目标地址, 而间接分支跳转则通过在代码中指定寄存器, 并在实际执行时根据寄存器中的值决定跳转的目标地址[8]。

对于实现流水线的处理器, 可能会由于计算分支指令的跳转目标而导致流水线的阻塞, 这种情况被称为控制流冒险。为缓解由此导致的性能损失, 现代处理器使用分支预测器对分支指令的分支跳转方向或分支目标地址做预测[1]。处理器在解析分支指令后判断预测结果是否正确, 如果正确, 则继续向下执行, 不造成其他影响; 如果错误, 则处理器需要回滚以消除架构层面造成的影响。

分支预测器的预测正确率对于处理器的性能起到至关重要的作用。在真实处理器上的实验表明, 减少一半的分支错误预测可以使处理器性能提高 13%[9]。因此自 20 世纪 80 年代以来, 研究者们在提高分支预测器预测精度做出诸多努力, 提出了诸如 gshare[10], 两级自适应模型[11]等分支预测器设计。虽然当前分支预测器的设计日益复杂, 但是可以抽取出其中重要的部件, 以更好理解分支预测器的原理。

目前已知的分支预测器部件主要有: 分支历史缓冲区(Branch History Buffer, BHB)、模式历史表(Pattern History Table, PHT)、和分支目标缓冲区(Branch Target Buffer, BTB)等。

BHB 用于记录已执行分支的跳转历史, 并与分支地址通过逻辑运算后, 用于索引 PHT 或 BTB 表项。在每次分支执行后, BHB 都会使用分支跳转的信息更新。在典型的如两级自适应模型中, BHB 又称为 HR, 在分支指令被解析后, HR 通过左移的方式将跳转方向压入到最后一位, 完成分支历史的更新[11]。

PHT 用于预测分支跳转的方向, 通常它会有很多表项, 通过 BHB 以及分支地址进行索引。PHT 表项的内部则一般含有一个有限状态机, 用于预测该分支的跳转方向。典型的 2 位有限状态机如下图所示。T 代表跳转, F 代表不跳转。各状态内的 T 与 F 代表在当前状态下的预测方向。而当实际正确的分支跳转发生后会对状态机造成影响, 状态可能会不变, 也可能跳变到另一状态。

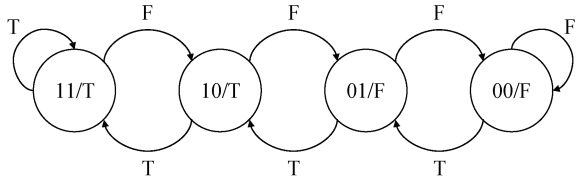


图 1 PHT 表项内的 2 位有限状态机
Figure 1 2-bit finite state machine in PHT entry

BTB 用于预测分支跳转的目标地址, 它的结构通常类似于 Cache。通过 BHB 以及分支地址进行索引, 并以分支地址的高地址做标签匹配。有研究者通过逆向工程发现, Intel 的 Haswell 微架构的 5～13 位用于索引, 14～29 位作为标签[12]。典型的 BTB 索引方式如图 2 所示。

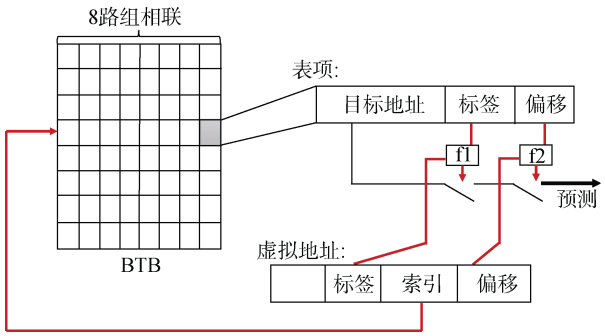


图 2 Haswell 微架构的 BTB 索引机制
Figure 2 BTB indexing mechanism in Haswell microarchitecture

为了高效地利用分支预测器, 分支预测器由核上各进程共同使用。此时若没有进行适当的隔离或刷新, 显然会导致潜在的安全隐患。以 Intel 为例, 在没有开启 STIBP 机制的情况下, 共享同一个物理核的两个逻辑核会共享物理核上的间接分支预测器, 而这样就会使得一个逻辑核上的线程可以控制另一个逻辑核上的线程的分支预测地址[13]。

而对于 Arm 架构的 Cortex-A53, 其包含有一个 256 表项的分支目标地址缓存(Branch Target Address Cache), 以及一个 3072 表项的全局的分支预测器(branch predictor)。分支目标地址缓存在上下文切换时不进行刷新[14], 而全局的分支预测器则在手册中没有明确给出是否刷新或隔离。这使得其可能存在与 Intel 上的 CC 与 RSC 类似的隐蔽信道, 并且利用

分支目标地址缓存也可能构建出新的隐蔽信道。

2.2 隐蔽信道

隐蔽信道是一种在系统的安全策略不允许的情况下进行数据传输的攻击。Butler Lampson[15]将这种信道定义为原本并不是为传输数据而使用的。由于它本身不使用合法的数据传输方式(例如读取和写入), 因此无法被操作系统的安全机制所检测或控制。它通常由高权限级的木马进程与低权限级的间谍进程组成。由于安全策略的原因, 高权限级的木马进程将无法向低权限级的间谍进程直接传递信息。一种可行方式是利用隐蔽信道来完成, 木马进程可以在一些共享的硬件资源上留下访问痕迹, 这种痕迹是可控并且可区分的, 木马进程这一动作相当于将私密信息编码进它在硬件资源的访问痕迹中。间谍进程通过探测硬件资源上的痕迹, 能够推理出木马进程对共享资源的访问方式, 并由此解码出木马进程想要传递的信息。

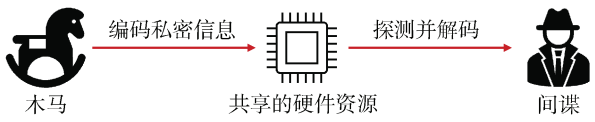


图 3 木马与间谍传递私密信息原理图

Figure 3 Principle diagram of trojan and spy transmitting private information

以基于分支预测器的隐蔽信道 RSC 为例, 木马进程会执行大量同方向的分支指令, 以此改变分支预测器之后的预测方向。假设木马执行了大量“不跳转”方向的分支指令, 之后分支预测器在对其他分支指令做预测时, 也同样会给出“不跳转”的方向预测。木马程序通过执行流来控制分支预测器状态的行为被称为“毒化”。在 RSC 中, 通常木马程序需要包含高达数万条的分支指令以保证分支预测器被完全的毒化。毒化完成后, 间谍进程同样执行大量同方向的分支指令。若间谍进程的分支指令实际跳转方向与毒化后的分支预测器的预测方向相同, 则其执行速度更快; 如果间谍进程的分支指令实际跳转方向与毒化后的分支预测器的预测方向不同, 则由于大量的分支预测错误, 导致处理器不断发生回滚, 执行速度变慢。在时间上产生了可观测的区别。同时在性能计数单元(Performance Monitor Unit)上也有可观测的指标, 如分支误预测事件。间谍程序根据所观测到的指标就可以推理出木马程序所执行的分支指令是“跳转”还是“不跳转”。如果木马程序的不同行为分别对应到私密数据的 0 和 1, 则间谍程序就此可以由此推理出木马程序所传递的私密数据。

3 基于分支预测器的隐蔽信道

3.1 威胁模型

假设系统上存在有两个进程, 分别为木马程序和间谍程序。木马程序拥有更高或与间谍程序相同的异常等级且具有对私密数据的访问能力。间谍程序的异常等级低于或等于木马程序, 且不具有对私密数据的访问能力。它们之间没有其他的信道可用于传递信息(如文件系统、网络等), 只能通过隐蔽信道来完成私密数据的传递。

我们考虑单一物理核上执行多个进程的应用场景。由于目前商用的 Arm 硬件平台上不存在类似于 Intel 的 SMT 技术, 所以这里不考虑两个进程运行于超线程的逻辑核的情况。在单个核上仅能有一个进程在运行, 由内核的调度器来进行处理器的时分复用。

本文实验使用性能计数事件作为观测指标。在 Arm 架构中, EL1 及以上的异常等级始终有对性能计数单元的访问权限, 而 EL0 则需要 EL1 及以上异常等级开启访问权限。由于本文实验全部于 EL0 完成, 因此对于 EL0 异常等级的间谍进程, 我们事先开启其访问性能计数单元的权限。而 EL0 得出的实验结论在 EL1 及以上异常等级同样成立。

3.2 基于争用的分支预测器隐蔽信道 CC

CC 由 Hdeunger 等人[2-3]于 2015 年提出, 又称为基于争用的分支预测器隐蔽信道(Contention-based covert channel)。其利用了木马进程和间谍进程对分支预测器的争用。木马程序根据想要转递 0 或 1, 执行不同的行为, 将分支预测器毒化为不同状态。当木马程序想要传递 1 时, 随机地执行大量分支, 其中一半为跳转, 一半为不跳转, 由此木马程序在处理器上造成了对分支预测器的随机争用。争用将导致间谍进程同样在执行大量分支时产生大量误预测, 同时导致执行间谍程序执行速度减缓。当木马程序想要传递 0 时, 则执行大量 nop 语句, 该语句不对分支预测器产生影响, 因此当间谍程序执行时, 其执行速度也不会受到木马程序的负面影响。间谍程序根据执行大量分支时是否产生大量误预测来判断木马程序的行为, 从而进一步推断木马程序所传递的私密数据。其原理如图 4 所示。

CC 隐蔽信道的伪代码如图 5 所示。木马程序根据所要传递的私密数据 n 执行不同代码。当 n 为 0 时, 执行 branches(); 当 n 为 1 时, 执行 nops()。间谍程序则周期性执行 branches(), 并在执行 branches() 前后分别采样分支误预测事件的计数值。如果计数

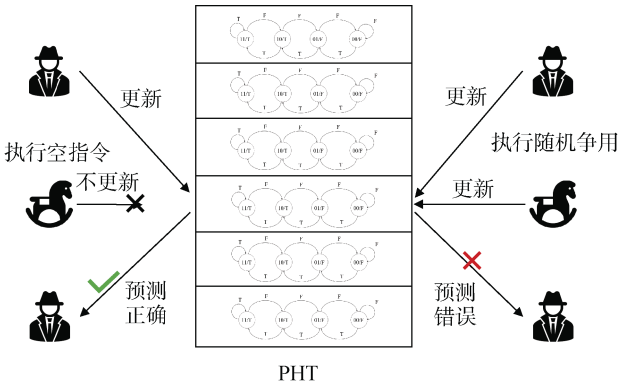


图 4　CC 原理图

Figure 4　Principle diagram of CC

```
Algorithm 1  Contention-based Covert channel
  trojan:
  if n == 0 then
    branches()
  else
    nops()
  end if


  spy:
  while true do
    usleep(5000)
    start=measurePMU(MISPRED)
    branches()
    end=measurePMU(MISPRED)
    if (end-start) is high then
      n = 0
    else {(end-start) is low}
      n = 1
    end if
  end while
```

图 5　CC 隐蔽信道伪代码

Figure 5　CC convert channel pseudocode

值之差较大, 则间谍程序推断木马程序传递的私密数据 n 为 0, 否则为 1。

伪代码中的 branches()和 nops()见图 6。从中可见, branches()内为一半的“跳转”和一半的“不跳转”。而在前文描述的分支方向预测组件 PHT 的原理中, 不难发现这种随机争用的本质是将 PHT 表项内容进行随机化的毒化。由于间谍进程在探测的过程中, 也会通过执行流对 PHT 表项造成影响, 因此如果木马程序在执行大量 nop 时, 其 PHT 将只受到间谍进程的跳转影响, 这类似于间谍程序在训练 PHT, 所以分支预测正确率提升, 同时执行速度加快; 而如果木马程序在进行随机化的毒化时, 间谍进程将使用未训练过的分支预测器进行分支预测, 所以分支误预测数上升, 同时执行速度减缓。

3.3　基于残留状态的分支预测器隐蔽信道 RSC

RSC 由 Dmitry Evtyushkin 等人[3]于 2016 年提出, 又称为基于残留状态的分支预测器隐蔽信道(Residual state-based covert channel)。由于前文提到执行流会影响 PHT, 而 PHT 状态会持续残留, 直到被替换或无效化。所以在 RSC 中, 间谍程序不再通过随机毒化的方式产生争用, 而是用同方向跳转进行毒化。其原理如图 7 所示。木马程序执行某一方向的分支语句毒化 PHT, 使得 PHT 之后的预测方向与毒化方向一致。当间谍程序的分支语句的跳转方向与预测方向一致时, 则预测成功, 分支误预测数低。反之预测错误, 分支误预测数高。

```
Algorithm 2  code blocks
  branches:
  mov w0, #0x1
  cmp w0, #0x0
  b.ne 2f
  nop
  cmp w0, #0x0
  b.eq 1f
  nop
  nop
  2:
  cmp w0, #0x0
  b.eq 1f
  nop
  nop
  1:
  cmp w0, #0x0
  b.ne 2f
  ...
  ...

  nops:
  nop
  nop
  nop
  ...
  ...
```

图 6　CC 隐蔽信道 branches()和 nops()伪代码

Figure 6　Pseudocode of CC covert channel branches() and nops()

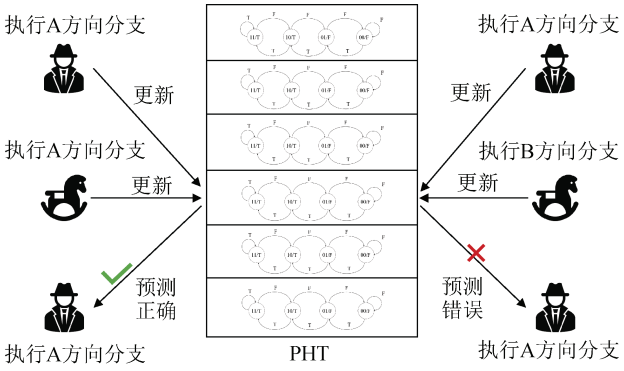


图 7　RSC 原理图

Figure 7　Principle diagram of RSC

RSC 隐蔽信道的伪代码如图 8 所示。木马程序根据所要传递的私密数据 n 执行不同代码。当 n 为 0 时, 执行 taken(); 当 n 为 1 时, 执行 nottaken()。间谍

程序则周期性执行 taken(), 并在执行 taken()前后分别采样分支误预测事件的计数值。如果计数值之差较小, 则间谍程序推断木马程序传递的私密数据 n 为 0, 否则为 1。

```
Algorithm 3  Residual State–based Covert channel
  trojan:
  if n == 0 then
    taken()
  else
    nottaken()
  end if


  spy:
  while true do
    sleep(5000)
    start=measurePMU(MISPRED)
    taken()
    end=measurePMU(MISPRED)
    if (end-start) is low then
      n = 0
    else {(end-start) is high}
      n = 1
    end if
  end while
```

图 8　RSC 隐蔽信道伪代码

Figure 8　RSC covert channel pseudocode

```
Algorithm4  code blocks
  taken:
  mov w0, #0x1
  cmpw0,#0x0
  b.ne 1f
  nop
  nop
  ...
  1:
  mov w0, #0x1
  cmp w0, #0x0
  b.ne 1f
  nop
  nop
  ...
  1:
  mov w0, #0x1
  cmp w0, #0x0
  b.ne 1f
  ...

  nottaken:
  mov w0, #0x1
  cmp w0, #0x0
  b.eq 1f
  nop
  nop
  ...
  1:
  mov w0, #0x1
  cmp w0, #0x0
  b.eq 1f
  nop
  nop
  ...
  1:
  mov w0, #0x1
  cmp w0, #0x0
  b.eq 1f
  ...
```

图 9　RSC 隐蔽信道 taken()和 nottaken()伪代码

Figure 9　Pseudocode of RSC covert channel taken() and nottaken()

伪代码中的 taken()与 nottaken()如图 9 所示, 木马程序在传递 0 时将执行大量“跳转”, 而在传递 1 时将执行大量的“不跳转”。这种同方向的分支跳转将使得 PHT 的预测为也变为毒化时的方向。间谍程序在执行时, 执行的指令其实际执行方向为“跳转”, 所以当木马程序传递 0 时, 分支预测器毒化为预测“跳转”, 间谍程序执行速度因此变快, 分支误预测数低; 而木马程序传递 1 时, 分支预测器毒化为预测“不跳转”, 间谍程序执行速度因此变慢, 分支误预测数高。

3.4　基于 BTB 的分支预测器隐蔽信道 BTBC

目前存在的基于分支预测器的隐蔽信道都是基于分支预测器中的 PHT 组件, 我们验证了其在 Arm 架构下的可行性, 并进行了分析。而由于分支预测器同时包含了其他的组件, 如 BTB, 这些组件在 Arm 架构是否也能被用于构建隐蔽信道目前还没有得到研究。本文对 Arm 架构的 BTB 进行了分析研究, 发现 BTB 也能被用于构建隐蔽通道, 我们将这种隐蔽通道称为 BTBC, 这是一种新的隐蔽信道。

如前文所述, PHT 是用于分支方向跳转预测的组件, BTB 是用于分支目标地址预测的组件。CC 以及 RSC 通过木马进程影响 PHT, 使间谍进程在执行到分支语句, 处理器在预测分支方向时误预测率上升或下降。同理, BTB 作为分支预测器中用于预测分支目标地址的组件, 也可被木马进程所影响, 使得间谍进程在执行目标地址的跳转时误预测率上升或下降。

图 2 展示了 Haswell 微架构的 BTB 索引机制。虽然根据微架构的不同, 分支预测器实现也会发生变化, 但是其索引机制大体是相同的。BTB 类似 Cache 的结构, 图 2 是一个 8 路组相联的 BTB, 分支指令的虚拟地址中的一部分索引位会用于索引到其中一组。然后虚拟地址的标签位与偏移位会和 BTB 一组内所有表项的标签位与偏移位做比较, 如果相等则认为命中该表项, 并进行预测。在间接分支指令执行完成后, 会用该指令对 BTB 表项进行更新, 替换掉老旧的表项。

在 BTBC 中, 间谍程序包含了大量间接分支指令。间谍程序在执行后, BTB 会相应更新, 并在下次间谍程序运行时进行分支目标地址预测, 因而几乎不会有分支误预测数。木马程序则根据传递的私密数据不同, 执行大量间接分支指令或大量空指令。当木马程序执行间接分支指令时会更新 BTB, 并替换掉之前与间谍程序有关的表项, 因此间谍程序在下次执行时, BTB 将不能再给出正确的预测。而当木马程序执行空指令时不会更新 BTB, 因此间谍程序在下次执行时, BTB 依然可以给出正确预测。其原理如图 10 所示。

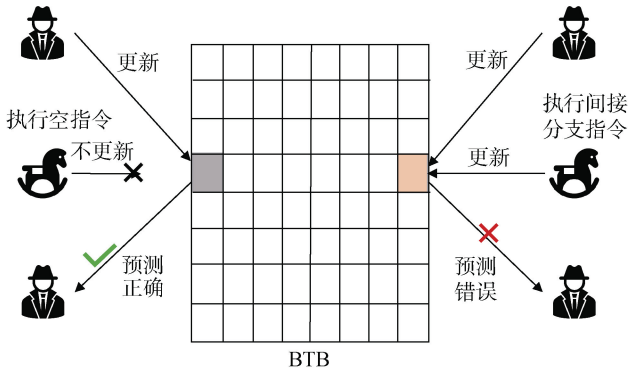


图 10　BTBC 原理图

Figure 10　Principle diagram of BTBC

根据此原理, BTBC 伪代码如图 11 和图 12 所示。

```
Algorithm 5  BTB-based Covert channel
  trojan:
  if n == 0 then
    branches()
  else
    nops()
  end if


  spy:
  while true do
    sleep(5000)
    start=measurePMU(MISPRED)
    branches()
    end=measurePMU(MISPRED)
    if (end-start) is high then
      n = 0
    else {(end-start) is low}
      n = 1
    end if
  end while
```

图 11　BTBC 隐蔽信道伪代码

Figure 11 BTBC covert channel pseudocode

```
Algorithm 6  code blocks
  branches:
  void target();

  ldrx0, [x29, #1624] // address of target()
  blr x0
  nop
  nop
  ...
  ldr x0, [x29, #1624]
  blr x0
  nop
  nop
  ...
  ...

  nops:
  nop
  nop
  nop
  ...
  ...
```

图 12　BTBC 隐蔽信道 branches()和 nops()伪代码

Figure 12　Pseudocode of BTBC covert channel branches() and nops()

在 BTBC 中, 当木马进程传递 0 时, 执行大量的间接分支指令, 更新大量 BTB 表项; 而在传递 1 时, 则执行大量的空指令, 不对 BTB 造成影响。间谍进程则定期执行大量间接分支指令, 并在执行前后均采集当前的分支误预测数, 然后根据分支误预测数之差来推断木马程序所要传递的私密数据。

4　实验

4.1　实验环境与设置

本文的实验基于 Hikey 970 开发板以及 Juno r2 开发板进行。Hikey 970 开发板包含 4 个 Cortex-A73 核心以及 4 个 Cortex-A53 核心, 运行内核为 GNU/Linux 4.9.78, 运行发行版为 Ubuntu 18.04.6 LTS OS。Juno r2 开发板包含 2 个 Cortex-A72 核心以及 4 个 Cortex-A53 核心, 运行内核为 GNU/Linux 4.14.59, 运行发行版为 Ubuntu 20.04.4 LTS OS。我们的所有实验在上述两块开发板的 Cortex-A53 核心以及 Juno r2 的 Cortex-A72 核心进行。

隐蔽信道由木马进程与间谍进程组成, 这两个进程均运行于 EL0, 同时开启了 EL0 对性能计数单元的访问权限。

虽然 EL0 开启 PMU 访问权限的情况较少, 但是由于 Cortex-A53 的分支预测器在不同异常等级与上下文间切换时并不进行状态清理, 因此对于 EL1 及以上的异常等级, 该隐蔽信道依然有效。同时在 EL1 及以上的异常等级, PMU 访问权限即可自由开启。因此本实验虽仅针对 EL0 进行验证, 但实际上该隐蔽信道仍可对更高的异常等级造成威胁。

如上文所述, 木马进程的任务是根据所要传递的私密信息, 执行不同行为完成分支预测器的毒化, 而间谍进程的任务是探测分支预测器状态, 并由此推理木马进程所传递的信息。实验中, 木马进程持续运行, 使得分支预测器的毒化完全; 而间谍进程则周期性地探测分支预测器状态, 通过 usleep()函数来控制探测间隔。其核上执行时序如图 13 所示。

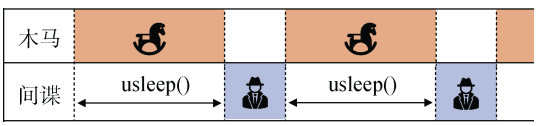


图 13　木马进程与间谍程序核上执行时序图

Figure 13　Trojan process and spy process execution sequence on single core

由于当前实验平台无 SMT 技术, 因此仅存在单核多进程、上下文调度的时序关系。实验中所有的隐蔽信道均使用该时序关系。

为保证木马进程与间谍进程不因调度原因而运行于其他核上, 导致实验误差。实验中使用 taskset 指令进行绑核操作, 通过 taskset 指定进程的 CPU 亲和性后, 进程不会运行于其他核上。

实验核上不运行其他进程, 以保证实验结果不受其他进程干扰。

4.2 参数设置

基于分支预测器的隐蔽信道利用执行流影响分支预测器, 并传递私密信息。木马程序和间谍程序的代码细节将决定该隐蔽信道的传输能力。因此本节将对前文所描述的伪代码的参数进行分析, 为隐蔽信道参数设置提供指导。

在隐蔽信道伪代码中, 涉及到三个可变参数: 间谍程序分支数, 木马程序分支数, 分支间隔指令数。

由于隐蔽信道中, 需要由木马程序对分支预测器进行毒化, 所以木马程序一般有大量分支语句以实现充分毒化。而间谍程序由于需要探测, 并测量其产生的分支误预测数。所以间谍程序通常也需要大量分支, 但该分支不可以过多。现代分支预测器通常只有有限多表项, 并以某种索引方式进行索引, 多个地址是有可能被索引到同一地址的。由于间谍程序本身的执行也会影响分支预测器, 如果分支预测器被前面执行的大量分支所影响, 后面执行的分支指令, 其预测将使用被前面分支所影响的表项进行预测。

除此之外由于索引机制中通常会利用分支地址的虚拟地址作为索引的一部分, 因此为了尽量避免多个分支地址被索引到同一表项, 调整分支间隔指令数这一参数, 从而调整分支地址的虚拟地址。由此使其尽可能索引到不同表项, 提高分支误预测数。

实验的评估指标为分支误预测数。虽然分支目标预测错误与分支方向预测错误均会计入分支误预测数中, 但是隐蔽信道在执行时, 木马程序和间谍程序通常持续占用处理器, 并且间谍程序内部仅执行直接跳转或间接跳转。以 BTBC 为例, 其执行时间谍程序所探测的分支误预测数以分支目标预测错误为主。除非进程调度导致间谍程序探测时引入了其他进程所带来的分支误预测噪声, 但是由于木马程序持续占用, 噪声通常很小。因此可将分支误预测数作为三种隐蔽信道执行时的评估指标。

理论上木马程序在传递 0 和传递 1 时, 间谍程序所探测到的分支误预测数差距越大, 则信号可区分度也越高, 同时也意味着当前的参数设置越好。因此本节实验共设有三个程序: 间谍程序 spy, 仅传递 0 时的木马程序 trojan-0 和仅传递 1 时的木马程序 trojan-1。注意到 trojan-0 和 trojan-1 仍然为木马程序, 只是它们仅传递 0 或 1。在之后的实验中, 会针对连续传递随机 01 串的木马程序进行通信速率的评估。

三个程序以如下形式组合。

1) Spy: 仅 Spy 程序运行在核上。

2) Spy+trojan-0: Spy 程序和 trojan-0 程序同时运行在同一个物理核上。

3) Spy+trojan-1: Spy 程序和 trojan-1 程序同时运行在同一个物理核上。

Spy 组可以评估当木马程序未运行时, 间谍程序所探测到分支误预测数。Spy+trojan-0 组可以评估, 当木马程序持续传递 0 时, 间谍程序所探测到的分支误预测数。Spy+trojan-1 组可以评估, 当木马程序持续传递 1 时, 间谍程序所探测到的分支误预测数。通过对比 Spy+trojan-0 与 Spy+trojan-1 两组的分支误预测数, 可以得到较优的参数设置。

4.2.1 间谍程序分支数

在评估间谍程序分支数对于信道的影响时, 将木马程序分支数设置为一个非常大的值, 以保证其能够达到完全的毒化效果。并在此基础上, 评估间谍程序分支数的合适参数设置。

三种隐蔽信道的间谍程序分支数对其分支误预测数的影响是类似的, 分支误预测数均随着间谍程序分支数的上升而上升, 直至达到一个最大值后, 其分支误预测数稳定。

CC 的间谍程序分支数在 2500 条以上可以有稳定的区分度, 如图 14 所示。Spy+trojan0 组的分支误预测数最高可达到 480 左右, Spy+trojan1 组的分支误预测数在稳定后则在 130 左右。

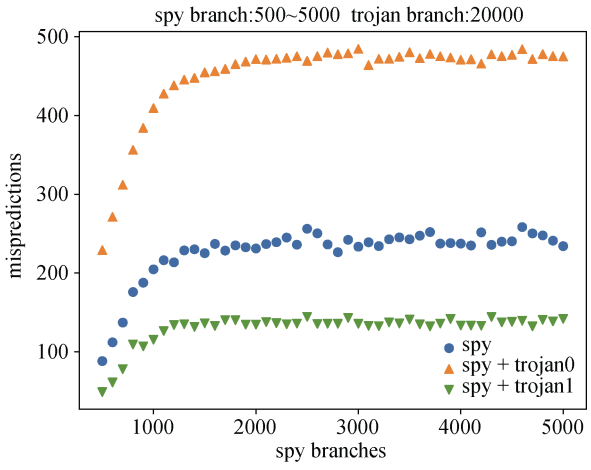


图 14 CC 间谍程序分支数对分支误预测数的影响

Figure 14 Influence of branch number of Spy program on branch mispredictions in CC

RSC 的间谍程序分支数在 1500 条以上可以有稳定的区分度, 如图 15 所示。Spy+trojan0 组的分支误

预测数最高可达到 780 左右，而 Spy+trojan1 组的分支误预测数则稳定在 0 左右。

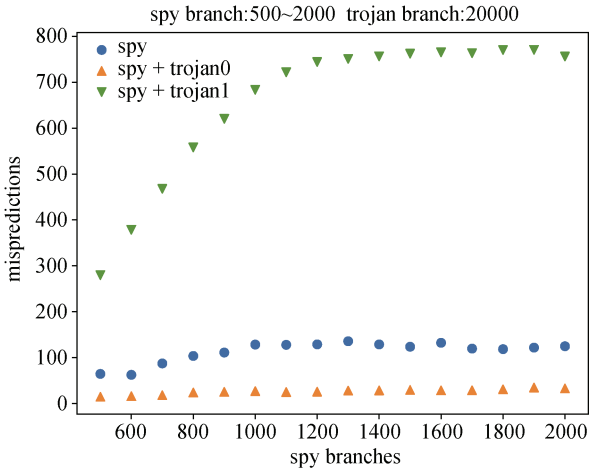


图 15 RSC 间谍程序分支数对分支误预测数的影响
Figure 15 Influence of branch number of Spy program on branch mispredictions in RSC

实验使用的 Cortex-A53 处理器的分支预测器包含一个 3072 个表项的分支历史预测表(Pattern History Prediction Table)[14]。但是无论是 CC 还是 RSC 的实验中没有观察到其分支误预测数可以达到 3072 个左右。其原因本文认为可能是受到了 PHT 索引机制的影响，程序分支地址、目标地址与分支历史使得其最多可索引表项数受限。因而即使间谍程序分支数继续上升，由于只能索引至有限数目的表项，后续的分支都将受益于之前分支的训练而预测正确。由于目前少有对于 Arm 处理器的分支预测器索引机制的研究，因此本文不对该现象的具体形成过程进行分析。

如图 16 所示，基于 BTB 的隐蔽信道 BTBC，其间谍程序的分支指令数在 300 条以上可以有稳定的区分度。Spy+trojan0 组的分支误预测数最高可达到 250 左右，而 Spy+trojan1 组的分支误预测数则稳定在 0 左右。最高分支误预测数与分支目标地址缓冲(Branch Target Address Cache)所包含的表项数 256 条相吻合[14]。

4.2.2 木马程序分支数

在 4.2.1 节评估了间谍程序分支数这一参数对信道的影响，在该节中，间谍程序分支数作出如下设置: CC 为 3000 条, RSC 为 2000 条, BTBC 为 500 条。

对于 CC 隐蔽信道，如图 17 所示，其木马程序在 3000 条分支进行毒化时，就可以达到明显的区分效果，相比在 Intel 上数万条的分支指令毒化[3]，我们认为由于 Cortex-A53 的分支预测器表项较少，因此用于毒化的分支指令也相对少很多。而如果小于 3000 条，则木马程序的毒化效果不稳定。

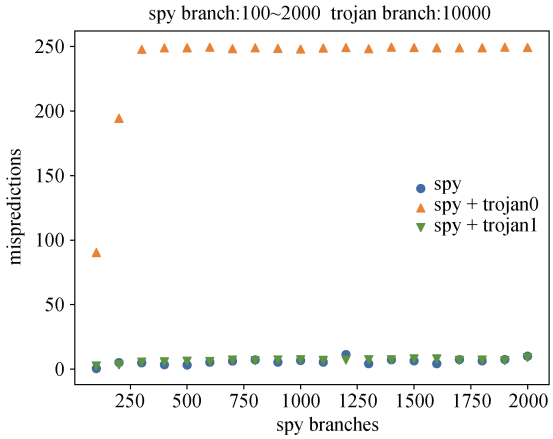


图 16 BTBC 间谍程序分支数对分支误预测数的影响
Fig 16 Influence of branch number of Spy program on branch mispredictions in BTBC

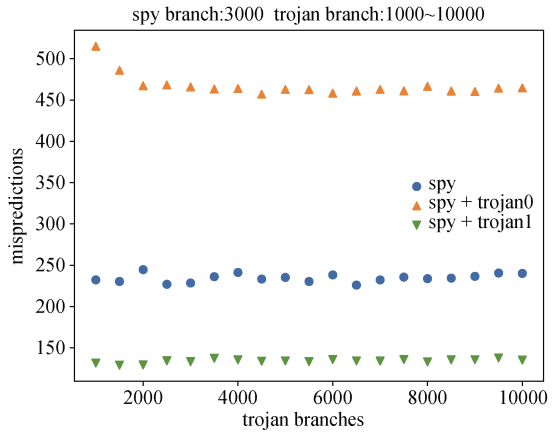


图 17 CC 木马程序分支数对分支误预测数的影响
Figure 17 Influence of branch number of trojan program on branch mispredictions in CC

在 RSC 上，分支指令数在 2000 条以上即可以达到稳定的毒化效果，最终间谍程序探测时也可以有较好的信号区分度，如图 18 所示。

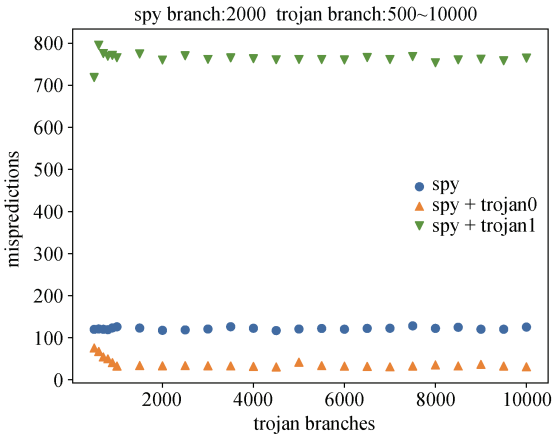


图 18 RSC 木马程序分支数对分支误预测数的影响
Figure 18 Influence of branch number of trojan program on branch mispredictions in RSC

对于 BTBC, 同样是 2000 条分支语句以上时, 毒化效果趋于稳定。Spy+trojan0 组的误预测数在 256 左右。如图 19 所示。

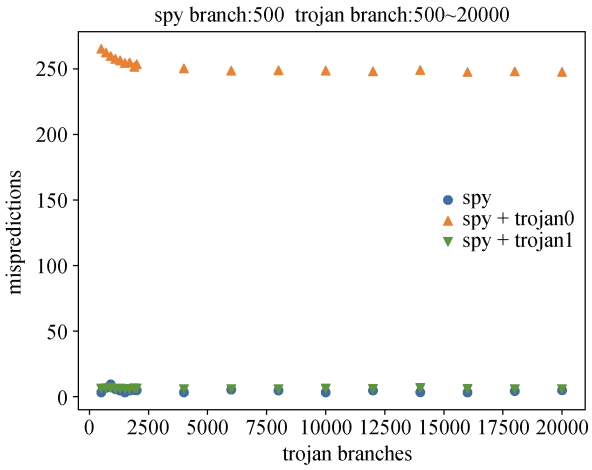


图 19 BTBC 木马程序分支数对分支误预测数的影响

Figure 19 Influence of branch number of trojan program on branch mispredictions in BTBC

4.2.3 分支间隔指令数

由于多个地址可能被分支预测器索引到同一表项, 通过改变分支指令间隔的指令数来改变分支的地址, 由此索引到分支预测器的不同表项。实验结果表明分支间隔对分支误预测数有非常明显的规律性影响。如图 20 所示, BTBC 在 $4n(n=1, 2, 3, \ldots)$条指令间隔时, Spy+trojan0 组的分支误预测会有一个非常明显的下降, 并且对于其下降后的分支误预测数也有规律性。我们分析可能是索引机制导致在某些分支间隔指令数下, 多条分支指令可以被索引至同一表项, 导致间谍程序探测的分支误预测数呈现规律性下降。这个结论在基于 PHT 的隐蔽信道也同样成立。

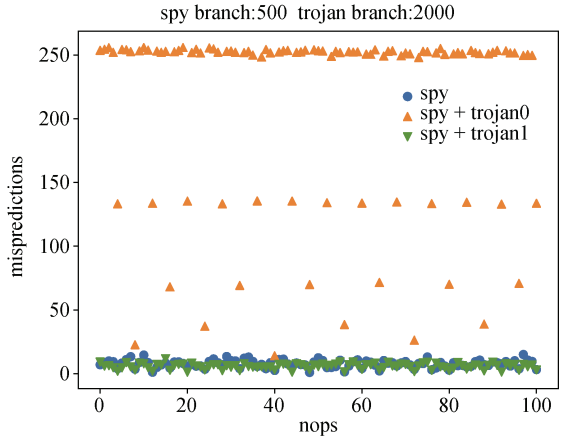


图 20 BTBC 分支间隔指令数对分支误预测数的影响

Figure 20 Influence of instruction interval number on branch mispredictions in BTBC

4.3 连续传输时信号特性评估

在 4.2 节中验证了三种隐蔽信道有良好的信号区分度, 本节实验在木马程序持续传输的情况下, 分析三种隐蔽信道的信号特性。参数设置参考 4.2 节, 在保证隐蔽信道稳定的前提下, 提供尽可能低的参数。在使用隐蔽信道通信时, 我们通常希望其能够更快传输, 而在较低的参数设置下, 传输速率相对更高。

实验中, 木马程序传输速率为 1bps, 每次传输 1 比特私密数据后, 木马程序翻转一次传递的信息, 即传递的是 01 序列; 间谍程序的探测频率为 10Hz。

对于 CC, 间谍程序分支数设置为 3000 个, 木马程序分支数为 5000 个。实验结果如图 21 所示。

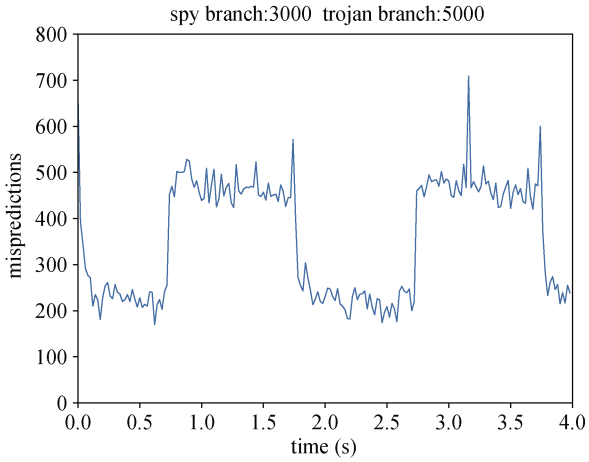


图 21 CC 隐蔽信道, 间谍进程探测的分支误预测数

Figure 21 Branch mispredictions of Spy probing in CC

从实验结果来看, CC 隐蔽信道在持续传递信息时, 可以观察到分支误预测数上的区分, 证明该隐蔽信道在 Arm 架构上的可行性。

但是, 间谍程序探测到的分支误预测数存在两个问题:

1) 信息改变时, 间谍程序的分支误预测数边缘不清晰。木马程序每 1s 翻转一次传递信息, 但是分支误预测数并没有严格按照 1s 有一个上升沿或下降沿。

2) 分支误预测数的震荡比较严重。在当前的实验参数设置下, 当木马程序传递 0 时, 间谍程序的分支误预测数在 400～550 个之间持续震荡; 在木马程序传递 1 时, 也同样出现了间谍程序的分支误预测数在 200～300 个之间震荡的现象。

对于 RSC, 间谍程序分支数设置为 2000 个, 木马程序分支数为 3000 个。实验结果如图 20 所示。

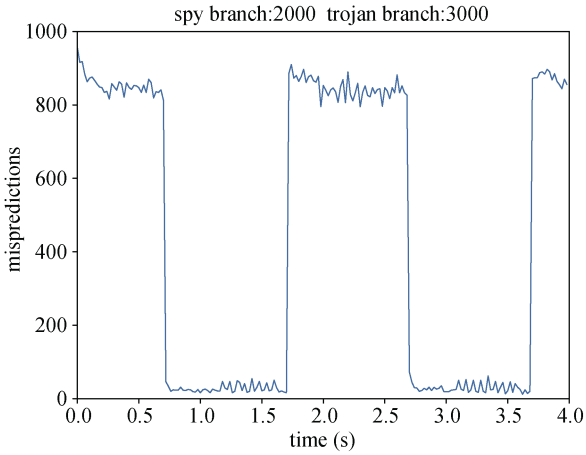


图 22 RSC 隐蔽信道，间谍进程探测的分支误预测数

Figure 22 Branch mispredictions of Spy probing in RSC

从实验结果来看，RSC 隐蔽信道在持续传递信息时，能够有比较明显的分支误预测数的区分。在木马程序传递 1 时，间谍程序的分支误预测数在 800～900 个之间；在木马程序传递 0 时，间谍程序的分支误预测数在 20～50 之间。其信号震荡程度较 CC 隐蔽信道更低。并且信息改变时，间谍程序的分支误预测数边缘也比较清晰。

与 CC 相比，RSC 明显在信道的稳定性以及区分性上更优。从这两类隐蔽信道对分支预测器的影响手段来看，CC 通过随机的分支跳转对分支预测器中的 PHT 组件进行了类似于驱逐的效果。在 Intel 上由于 SMT 机制的存在，同时运行的超线程会对物理核上唯一的分支预测器产生争用，导致执行速度减缓以及分支误预测的上升[2]。但是在 Arm 架构下，由于不存在 SMT 机制，木马进程以及间谍进程只能分时占用物理核，因此实际上 CC 隐蔽信道的随机分支跳转起到了 PHT 驱逐的效果。从图 21 可见，当木马程序传递 0 时，由于 PHT 驱逐，分支误预测上升；而当木马程序传递 1 时，由于仅运行空指令，未对 PHT 造成影响，其分支误预测数较低。RSC 通过大量同方向的分支跳转对分支预测器中的 PHT 组件进行了毒化，并残留下大量预测同方向的 PHT 表项。当木马程序传递 0 时由于间谍进程与木马进程毒化的分支跳转方向一致，分支误预测数较低；而木马程序传递 1 时由于两者分支跳转方向相反，分支误预测数较高。从实验结果也可看出，残留状态的毒化相比随机驱逐的毒化效果更优。

对于 BTBC，间谍程序分支数设置为 500 个，木马程序分支数为 2000 个。实验结果如图 23 所示。

从实验结果来看，BTBC 表现出良好的稳定性以及较高的区分度。在木马程序传递 0 时，其分支误预测数在 256 个左右；在传递木马进程传递 1 时，其分支误预测数始终保持 20~30 个之间。分支误预测数的上升沿和下降沿非常清晰，且没有发生如 CC 或 RSC 同等程度的震荡。其中的几个离群值本文认为受到调度程序的影响，引入了其他程序导致分支误预测上升。但离群值也同样没有大幅偏离。分支误预测数的区分度较高，但是由于 Arm 的分支目标地址缓存表项仅 256 项，信号振幅不如利用 PHT 的 CC 和 RSC。

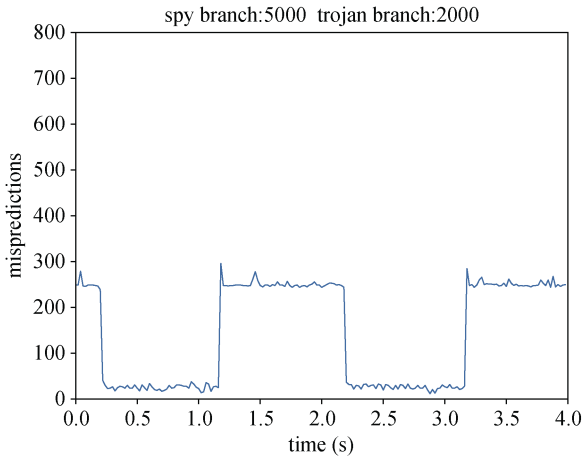


图 23 BTBC 隐蔽信道，间谍进程探测的分支误预测数

Figure 23 Branch mispredictions of Spy probing in BTBC

以上结果说明，BTBC 在信号稳定性上相比 CC 和 RSC 更优，但是在信号区分度上与 CC 接近，低于 RSC。

4.4 通信速率评估

信道评估时一个重要的指标是通信速率，我们期望信道能够在保持连续传递信息的前提下，得到更高的通信速率以及可接受的错误率。不同于普通的信道，隐蔽信道所利用的传输介质本身并不用于传输信息，其传输特性可能会受到传输介质特性、实现细节以及优化的影响。对于利用不同介质的隐蔽信道，其传输速率可能有非常大的差距。如利用电磁信号的隐蔽信道，在达到 10 bps 速率的同时仅有 12.5%的错误率[16]，而利用热量的隐蔽信道，其传输速率可以慢到 500 天仅可传递一张 4 MB 的图片[17]。

本节将对以上三个隐蔽信道的传输速率进行讨论。一些编码方式可以提升信道的抗噪声能力，如汉明码等，在本文中不做考虑，而仅讨论基于分支预测器的隐蔽信道的特性。

对传输速率的评估我们选用两个指标: 误码率及传输速率。在本节, 我们改变两个参数: 间谍程序探测的频率 F_{spy} 以及木马程序信息传输的频率 F_{trojan}。传输的信息也不采用之前 01 交替的序列, 而使用随机的 01 序列, 以测量实际使用场景下的误码率与传输速率。测试核仅运行木马程序和间谍程序, 不运行其他程序以避免引入噪声。

实验中, 为了保证间谍程序探测结果的可靠性, 我们设定间谍程序探测的频率 F_{spy} 始终是木马程序信息传递频率 F_{trojan} 的 5 倍。实验结果也表明相对木马程序信息传递频率提高间谍程序探测的频率有助于降低信息误码率。

信道在传输数据时将不可避免地引入噪声, 因此间谍程序接收到的信息可能会在任意位置增加 1 比特、减少 1 比特, 或者该比特错误。将木马程序传递的比特串定义为 $BITS_{true}$, 将间谍程序探测并推理出的比特串定义为 $BITS_{probe}$。它们之间错误的比特位数可以被抽象为 $BITS_{probe}$ 对于 $BITS_{true}$ 的莱温斯坦距离(或称为编辑距离) $\varphi\left(BITS_{probe}, BITS_{true}\right)$。那么信道的误码率 SER 即:

$$SER = \frac{\varphi\left(BITS_{probe}, BITS_{true}\right)}{BITS_{true}}$$

在本节实验中, 参数设置与 4.3 节相同。对于 CC 隐蔽信道, 采用 3000 个间谍程序分支, 5000 个木马程序分支, 分支间隔指令数为 3 条; 对于 RSC 隐蔽信道, 采用 2000 个间谍程序分支, 3000 个木马程序分支, 分支间隔指令数为 3 条; 对于 BTBC 隐蔽信道, 采用 500 个间谍程序分支, 2000 个木马程序分支, 分支间隔指令数为 3 条。

实验测试了上述三种隐蔽信道, 信道的传输速率即对应于木马程序信息传递的频率 F_{trojan}。例如, 当 F_{trojan} 为 10Hz 时, 信道的传输速率即为 10 bps。

传输速率与误码率的关系如图 24 所示。在传输速率低于 120 bps 时, RSC 与 BTBC 表现出了接近于 0 的误码率, 而 CC 的误码率则相对高一些。在大于 100 bps 后, 三种隐蔽信道的误码率均逐步上升。其中, RSC 的误码率相比 CC 与 BTBC 均更低。三种隐蔽信道在传输速率上升时, 均会出现误码率的震荡。

同误码率下三种信道可达到的最高传输速率如图 25 所示。其结果与图 24 所表现的相同。RSC 可以在相同的误码率下达到更高的传输速率, 而 BTBC 与 CC 近似。

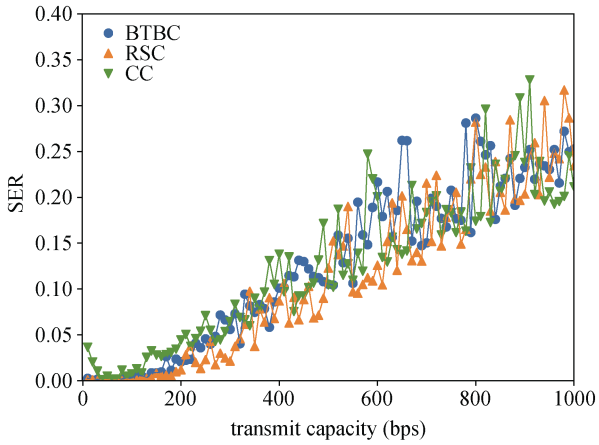


图 24 三种隐蔽信道的 SER 随传输速率变化曲线图

Figure 24 SER and capacity for three covert channel

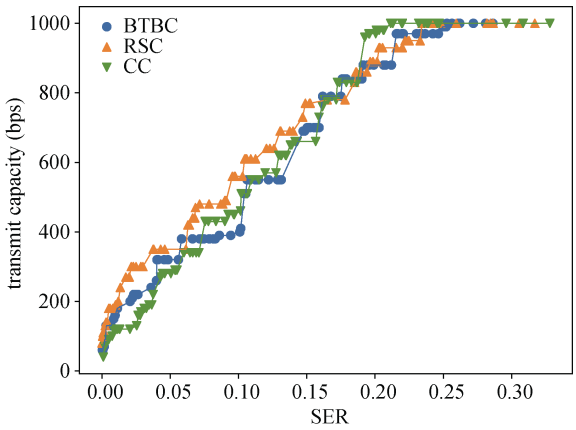


图 25 三种隐蔽信道的同 SER 下最大传输速率图

Figure 25 The maximum capacity of the three covert channels under the same SER

Dmitry Evtyushkin[3]在测量 RSC 信道时, 采用了不同的方式。在其实验方案中, 木马程序和间谍程序分别调用 sched_yield()函数以主动让出处理器, 而此时由于处理器上仅运行有木马和间谍, 因此实现了木马和间谍的连续传递与接收。这种实验设计将会带来更高的传输速率, 相当于每次木马传递信息, 间谍仅探测一次。但是现实的隐蔽信道使用中, 很难保证同一个核上仅运行两个进程, 这种测量受到系统调度的影响更大。

在实验中, 我们对 SER 上升的原因进行了分析。在实验方案中, 木马会在传输 1 比特时持续运行, 由间谍程序在这段时间内探测 5 次, 然后通过 5 次探测的分支误预测数来推理当前木马程序传递的信息。但是木马程序传输速率提升, 时间间隔因此相应变短, 间谍程序将难以在短暂的时间内探测到 5 次。调度算法无法保证能调度间谍进程来占用处理器, 探测次数也因此存在严重的波动。极端情况下, 如木马程序的传输速率为 10000 bps 时, 实验观测的结果是

间谍程序在 100 us 内的探测次数在 1～5 次间波动。最终导致了探测到的结果很难推理出木马程序所传递的信息。如 4.1 节所述，间谍程序通过 usleep()函数来控制间谍程序的探测时机，由于 Linux 并不是一个实时操作系统，传输速率越高，时间间隔越短，其受到调度的干扰也就越严重。

除此之外，误码率并非随着传输速率而稳步上升，其同样存在着震荡。其原因我们同样认为受到了进程调度的影响，由于间谍程序通过 usleep()函数以控制探测时机，在确定的传输速率下，间谍程序的休眠时间也是确定的。由于进程调度的不确定性，某些休眠时间可能会更容易导致间谍程序探测时错位，无法将五次探测准确对应到每次木马的数据传输。其最终导致间谍程序无法正确推理出木马传递的信息，误码率上升。

由上述分析，我们建议在使用基于分支预测器的隐蔽信道时，尽量提高间谍探测次数以避免噪声导致的干扰，同时如果能够在已知存在调度算法干扰的情况下，适当修改推理时的参数，同样可以降低信道的 SER。根据处理器与内核的不同，需要设定合适的传输速率，以保证其误码率处于较低水准。

4.5 Cortex-A72 下隐蔽信道传输速率评估

前文中的实验均基于 Cortex-A53 进行，本节我们同样将其在 Cortex-A72 下进行实验，评估信道传输时的传输速率及其误码率。

虽然 Cortex-A72 手册中并未给出分支预测器比较详细的参数，但是由于 Cortex-A72 相比 Cortex-A53 性能更高，其分支预测机制相应更加复杂，分支预测器表项也相应会更多。与 4.2 节相同，我们对三种隐蔽信道的参数进行了评估。实验结果证实在 Cortex-A72 下，PHT 和 BTB 的索引机制更加复杂。如图 26 所示，对于 RSC，随着间谍程序分支数的增加，Spy+trojan0 和 Spy+trojan1 的分支误预测数是同步增加的，并且 CC 的情况与 RSC 类似。

BTBC 也有相同的情况，但其相对会有更加明显的分支误预测区分，如图 27 所示。

在改变木马程序分支数时，三种隐蔽信道表现的情况类似，当木马程序分支数大于间谍程序分支数后，其分支误预测数的区分即稳定。如图 28 所示。

在改变分支间隔指令数时，三种隐蔽信道的分支误预测数的变化并未如在 Cortex-A53 上有规律性，如图 29 所示。本文认为其原因在于 Cortex-A72 的复杂索引机制使得无法通过改变分支间隔指令数来控制分支所索引到的表项数。

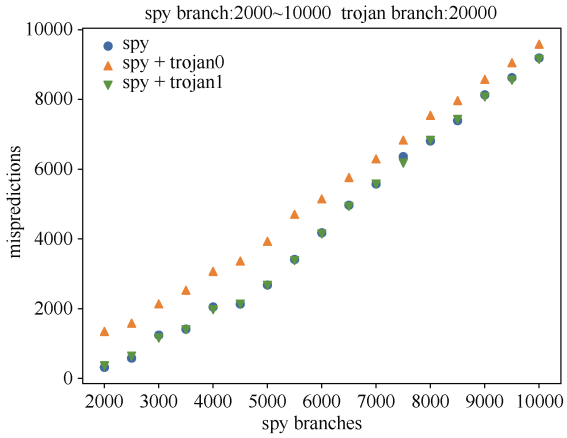


图 26 A72 上 RSC 的间谍程序分支数对误预测数的影响

Figure 26 Influence of branch number of spy program on branch mispredictions of RSC in A72

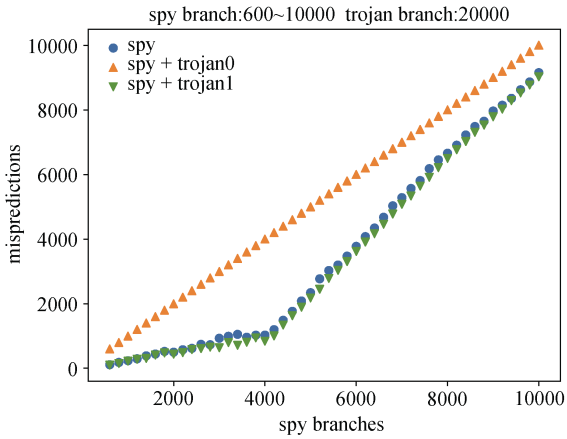


图 27 A72 上 BTBC 的间谍程序分支数对误预测数的影响

Figure 27 Influence of branch number of Spy program on branch mispredictions of BTBC in A72

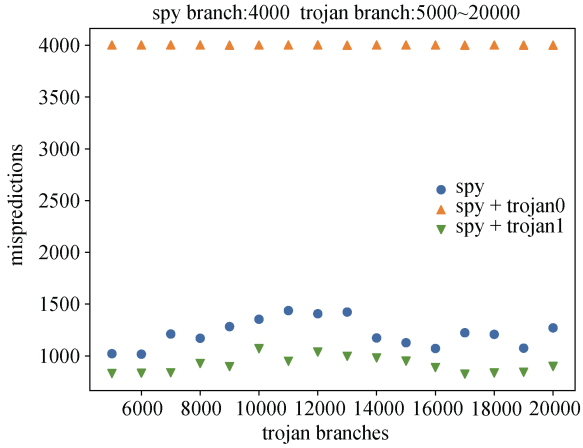


图 28 A72 上 BTBC 的木马程序分支数对误预测数的影响

Figure 28 Influence of branch number of trojan program on branch mispredictions of BTBC in A72

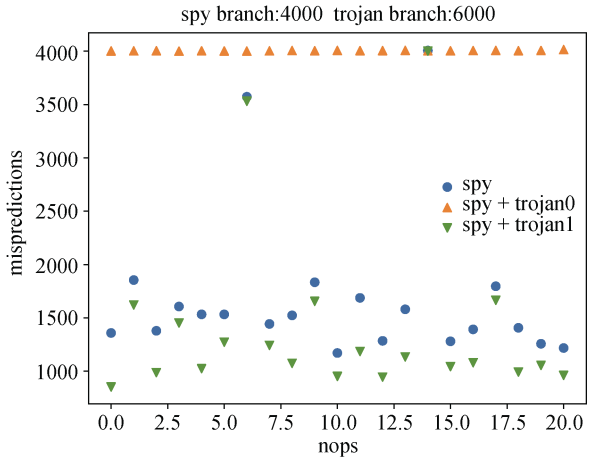


图 29　A72 上 BTBC 的分支间隔指令数对误预测数的影响

Figure 29　Influence of instruction interval number on branch mispredictions of BTBC in A72

参数设置如下。对于 CC 隐蔽信道, 采用 4000 个间谍程序分支, 9000 个木马程序分支, 分支间隔指令数为 3 条; 对于 RSC 隐蔽信道, 采用 5000 个间谍程序分支, 7000 个木马程序分支, 分支间隔指令数为 3 条; 对于 BTBC 隐蔽信道, 采用 4000 个间谍程序分支, 6000 个木马程序分支, 分支间隔指令数为 3 条。

实验结果如图 30 和图 31 所示。CC 与 BTBC 随着传输速率的升高, 误码率的升高轨迹类似。而 RSC 则在传输速率尚低时即表现出 30%左右的误码率, 并且一直维持。在查看 RSC 隐蔽信道中间谍程序探测得到的误预测数后, 本文认为其高误码率的原因有两点。第一点在于信号非常不清晰, 其信号振幅也很小。如图 26 所示, 当木马传递 0 和 1 时, 间谍程序所探测到的分支误预测数的区别并不大。第二点在于隐蔽信道原理上的不同, RSC 基于残留状态, 而 CC 和 BTBC 基于驱逐。我们推测 Cortex-A72 对于分支预测器表项有比较严格的进程隔离, 这导致 RSC 中木马进程通过不同方向的毒化来传递信息不再可行。而 CC 与 BTBC 本身基于驱逐, 即使有进程隔离, 木马进程驱逐与不驱逐分支预测器表项, 间谍进程仍然可以探测到区别。

Cortex-A72 下三种隐蔽信道并未因处理器主频的提高而可以在与 Cortex-A53 同等传输速率下得到更低的误码率或同等误码率下得到更高的传输速率。同传输速率下相比 Cortex-A53, CC 与 BTBC 的误码率近似。而 RSC 则在 Cortex-A72 下表现较差, 误码率一直在 30%以上。

4.6　实验结果分析

我们的实验验证了以上三种隐蔽信道在 Arm 架构上的存在。其中的 CC 与 RSC 在 Intel 上存在。Dmitry Evtyushkin 在 Intel i7-4800MQ 上实现了 RSC, 其传输速率在高达 121 kbps 的同时, 错误率为 3.9%。相比之下, Arm 架构处理器上的 RSC 不仅无法达到如 intel 下的传输速率, 同等的传输速率下其误码率也更高。其原因本文认为一方面是嵌入式设备的处理器相比于桌面端处理器性能更羸弱, 而另一方面更重要的原因则是实验设置中, Dmitry Evtyushkin 通过 sched_yield()进行木马和间谍的调度, 而本文中通过 usleep()进行控制, 在核上无其他进程的情况下, 使用 sched_yield()能更快进行调度。

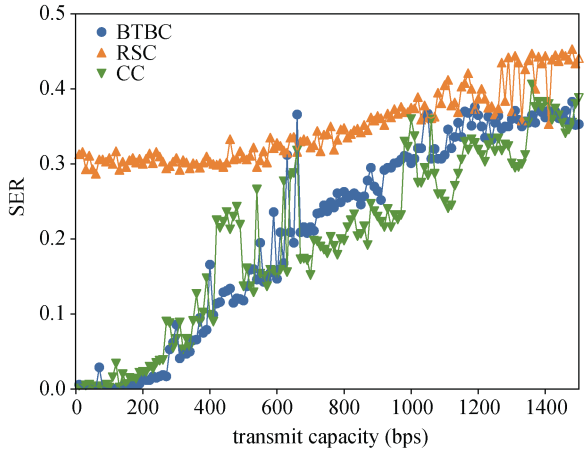


图 30　A72 上三种隐蔽信道的 SER 随传输速率变化曲线图

Figure 30　SER and capacity for three covert channel in A72

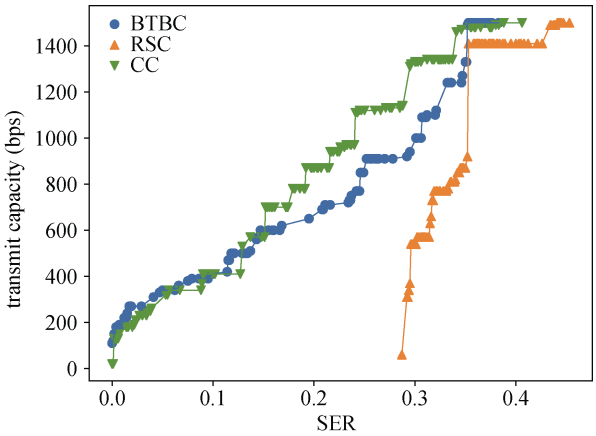


图 31　A72 上三种隐蔽信道的同 SER 下最大传输速率图

Figure 31　The maximum capacity of the three covert channels under the same SER in A72

同时, Cortex-A53 上对于三种隐蔽信道的实验结果表明新发现的 BTBC 隐蔽信道, 其信号振幅上弱于 RSC, 由于分支目标地址缓存仅有 256 个表项, 因此 BTBC 信号振幅最大也不超过 256, 而 RSC 利用

了 PHT 组件, 因此其信号振幅更大。BTBC 在信号稳定性上相比CC和RSC更优, 但是在信号区分度上与 CC 接近, 低于 RSC。在传输速率和误码率上 BTBC 表现出了比 CC 和 RSC 近似的信道性能。

Cortex-A72 上的实验表明, 在更加复杂的分支预测机制的情况下, BTBC 依然表现出了良好的信道性能, 其与 CC 近似, 优于 RSC。

在 Cortex-A53 和 Cortex-A72 上的实验均表明, BTBC 在 200 bps 的传输速率下, 仅有 2%的误码率。

5 防御措施

基于分支预测器的隐蔽信道通过改变分支预测器状态, 进而在分支误预测数这一观测指标上产生显著的区别, 从而传递私密信息。因此防御点共有两个: 分支预测器状态改变和观测指标。根据这两个防御点我们给出如下的防御措施的建议。

对于第一个防御点“分支预测器状态改变”。在硬件设计层面, 可以在不同异常等级或不同进程切换时清除分支预测器的表项或初始化表项状态, 以此来避免攻击者能够精确探测分支预测器的表项或状态变化。但这种方式可能会导致较大的性能开销。除此之外, 也可以引入程序噪声, 使得分支预测器状态即使被木马程序所毒化, 但是由于噪声的存在, 其毒化并不充分, 在间谍程序探测时也难以得到良好的信号区分度。

对于第二个防御点“观测指标”, 如果是针对 EL0 则可以直接关闭性能计数单元的访问权限, 避免 PMU 事件的滥用。对于 EL1 及以上的异常等级, 则可监测关键的 PMU 事件, 在 PMU 事件计数值超出正常执行的范围时, 判断可能有基于分支预测器的隐蔽信道的发生[18]。

6 总结与展望

本文在真实的 Arm 硬件平台上实现了基于分支预测器的隐蔽信道 CC 和 RSC, 证明了这两种 x86 架构上存在的隐蔽信道在 Arm 架构下同样存在。同时我们也发现了一个新的基于分支预测器 BTB 组件的隐蔽信道 BTBC。我们系统性地对其进行了分析和对比, 说明了这三种隐蔽信道在毒化原理和所利用组件的区别, 以及由此所导致的信道特性上的不同。我们评估并分析了不同参数设置对上述三个隐蔽信道的性能影响, 给出了较优的参数设置方案。并在此基础上对三种隐蔽信道的信道性能进行了评估。CC 由于通过随机争用的方式对分支预测组件PHT进行毒化, 其信道性能相对 RSC 较弱。RSC 则由于通过残留状态的方式对分支预测组件PHT进行毒化, 在Cortex-A53的实验中信号的振幅更明显, 信号边缘更清晰, 同传输速率下相比 CC 和 BTBC 稍优。但是在 Cortex-A72 的实验中, 受到分支预测机制的影响, 同传输速率下其误码率相比 CC 与 BTBC 过高。BTBC 在振幅上弱于RSC, 与CC接近, 但是其信号的震荡程度最小, 信号边缘也最为清晰, 同时在 Cortex-A72 与 Cortex-A53 下均保持良好的信道性能。BTBC 在 200 bps 的传输速率下, 仅有 2%的误码率。最后, 我们根据此类隐蔽信道特性, 给出了相应的防御措施。

下一步工作中, 可以探索将该隐蔽信道与瞬态攻击等微架构攻击组合, 以实现更强力的攻击, 如窃取 TrustZone 中的私密数据。除此之外, 当前对于 Arm 架构处理器的分支预测器 BTB 组件的索引机制尚不明确, 后续对索引机制进行逆向, 可以实现 BTB 侧信道。

参考文献

[1] Hennessy J L, Patterson D A. Computer Architecture: A Quantitative Approach[M]. Burlington: Morgan Kaufmann Publishers, 2011: 658-670.

[2] Hunger C, Kazdagli M, Rawat A, et al. Understanding Contention-Based Channels and Using Them for Defense[C]. *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, 2015: 639-650.

[3] Evtyushkin D, Ponomarev D, Abu-Ghazaleh N. Understanding and Mitigating Covert Channels through Branch Predictors[J]. *ACM Transactions on Architecture and Code Optimization*, 2016, 13(1): 1-23.

[4] Acıiçmez O, Seifert J, Koç C K. Predicting secret keys via branch prediction[C]. *Cryptographers' Track at the RSA Conference*, 2007: 225-242

[5] Lee S, Shih M W, Gera P, et al. Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing[EB/OL]. 2016: 1611.06952. https://arxiv.org/abs/1611.06952v3.

[6] Template attack on blinded scalar multiplication with asynchronous perf-ioctl calls. https://eprint.iacr.org/2017/968.pdf. Dec 2022.

[7] Topic 14: Dealing with Branches. https://www.cs.princeton.edu/courses/archive/fall15/cos375/lectures/14-bpred-2x2.pdf. Dec 2022.

[8] Mittal S. A Survey of Techniques for Dynamic Branch Prediction[EB/OL]. 2018: 1804.00261. https://arxiv.org/abs/1804.00261v1.

[9] Jiménez D A. An Optimized Scaled Neural Branch Predictor[C]. *2011 IEEE 29th International Conference on Computer Design*, 2011: 113-118.

[10] McFarling S. Combining branch predictors. Technical Report. Digital Western Research Laboratory, 1993, 49

[11] Yeh T Y, Patt Y N. Two-Level Adaptive Training Branch Prediction[C]. *The 24th annual international symposium on Microarchitecture - MICRO 24*, 1991: 51-61.

[12] Zhang T, Koltermann K, Evtyushkin D. Exploring Branch Predictors for Constructing Transient Execution Trojans[C]. *The Twenty-Fifth International Conference on Architectural Support for*

Programming Languages and Operating Systems, 2020: 667-682.

[13] Single Thread Indirect Branch Predictor. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html. Mar 2018.

[14] Holdings ARM. ARM Cortex-A53 MPCore Processor. Technical Reference Manual. Arm, 2014.

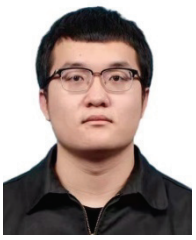
[15] Lampson B W. A Note on the Confinement Problem[J]. *Communications of the ACM*, 1973, 16(10): 613-615.

[16] Guri M. LANTENNA: Exfiltrating Data from Air-Gapped Networks *via* Ethernet Cables Emission[C]. *2021 IEEE 45th Annual Computers, Software, and Applications Conference*, 2021: 745-754.

[17] Guri M, Monitz M, Mirski Y, et al. BitWhisper: Covert Signaling Channel between Air-Gapped Computers Using Thermal Manipulations[C]. *2015 IEEE 28th Computer Security Foundations Symposium*, 2015: 276-289.

[18] He Z C, Lee R B. CloudShield: Real-Time Anomaly Detection in the Cloud[EB/OL]. 2021: 2108.08977. https://arxiv.org/abs/2108.08977v2.

杨毅 于 2020 年在燕山大学计算机科学与技术专业获得学士学位。现在北京邮电大学网络空间安全专业攻读硕士学位。研究领域为处理器安全。研究兴趣包括: 处理器微架构攻击、硬件辅助的安全防护。Email: yangyi0570@outlook.com



吴凭飞 于 2022 年在清华大学计算机科学与技术专业获得学士学位, 现在清华大学计算机系攻读硕士学位。研究领域为处理器硬件安全。研究兴趣包括硬件漏洞分析和检测。Email: wpf22@mails.tsinghua.edu.cn

邱朋飞 于 2020 年在清华大学计算机科学与技术专业获得博士学位, 现任北京邮电大学特聘副研究员。研究领域为计算机硬件安全。研究兴趣包括: 处理器硬件漏洞挖掘与利用、处理器安全模型设计、人工智能加速器安全。Email: qpf@bupt.edu.cn

王春露 于 1994 年在哈尔滨工业大学获得硕士学位, 现在北京邮电大学网络空间安全学院任教授, 研究领域为网络空间安全, 研究兴趣包括处理器安全、人工智能安全。Email: wangcl@bupt.edu.cn

张锋巍 于 2015 年在美国乔治梅森大学计算机专业获得博士学位, 现任南方科技大学副教授/研究员。研究领域为系统安全。研究兴趣包括: 可信执行、硬件辅助安全、恶意软件透明分析等。Email: zhangfw@sustech.edu.cn

赵路坦 于 2021 年在中国科学院大学网络空间安全专业获得博士学位。现任中国科学院信息工程研究所信息安全国家重点实验室副研究员。研究领域为计算机体系结构、处理器芯片安全。研究兴趣包括: 推测执行漏洞、分支预测器侧信道、缓存侧信道等处理器芯片安全漏洞防御。Email: zhaolutan@iie.ac.cn

王博 于 2018 年在清华大学电子科学与技术学科获得博士学位, 现于飞腾信息技术有限公司担任副研究员, 研究领域为处理器安全与硬件安全, 研究兴趣包括: 处理器安全架构、侧信道安全等。Email: wangbo@phytium.com.cn

吕勇强 于 2006 年在清华大学计算机科学与技术专业获得博士学位, 现于清华大学北京信息科学与技术国家研究中心担任副研究员, 研究领域为处理器安全, 研究兴趣包括: 处理器漏洞挖掘、侧信道安全。Email: luyq@tsinghua.edu.cn

王海霞 于 2004 年在中国科学院计算技术研究所获得博士学位, 现任清华大学信息科学与技术国家研究中心副研究员, 研究领域为计算机系统结构, 研究兴趣包括: 处理器结构设计、处理器安全验证。Email: hx-wang@tsinghua.edu.cn

汪东升 于 1995 年在哈尔滨工业大学获得博士学位。现任清华大学计算机系长聘教授。研究领域为计算系统安全、处理器设计。Email: wds@tsinghua.edu.cn