中国科学院大学

University of Chinese Academy of Sciences

博士学位论文

处理器的指令安全脆弱性检测技术研究

作者姓名：王光

指导教师：程旭 教授

北京大学

学位类别：工学博士

学科专业：计算机系统结构

培养单位：中国科学院信息工程研究所

2022 年 12 月

Research on the Detection Technology of Processor Instruction Vulnerabilities

A dissertation submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Doctor of Philosophy

in Computer Architecture

By

WANG Guang

Supervisor: Professor CHENG Xu

Institute of Information Engineering, Chinese Academy of Sciences

December, 2022

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。承诺除文中已经注明引用的内容外，本论文不包含任何其他个人或集体享有著作权的研究成果，未在以往任何学位申请中全部或部分提交。对本论文所涉及的研究工作做出贡献的其他个人或集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院大学有关收集、保存和使用学位论文的规定，即中国科学院大学有权按照学术研究公开原则和保护知识产权的原则，保留并向国家指定或中国科学院指定机构送交学位论文的电子版和印刷版文件，且电子版与印刷版内容应完全相同，允许该论文被检索、查阅和借阅，公布本学位论文的全部或部分内容，可以采用扫描、影印、缩印等复制手段以及其他法律许可的方式保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘 要

处理器是所有计算的最终执行单元。攻击者能够利用处理器中的硬件资源对计算机系统发起攻击，这些能被攻击者利用的硬件资源被称为处理器中的脆弱性，它们与指令系统中的指令有一定的对应关系，我们称之为指令安全脆弱性（Vulnerability）。有些安全脆弱性对应单条指令，例如隐藏指令和指令缺陷；有些安全脆弱性对应指令组合，例如指令组合触发的 Cache 侧信道攻击 [1, 2] 和随机数发生器对应的隐蔽信道攻击 [3]。攻击者在软件中嵌入隐藏指令或者能够触发侧信道攻击的指令组合就能达到窃取计算机中私密信息、使计算机系统宕机、改变程序执行行为等目的；而且这些攻击行为的隐蔽性较高，很难被现有的恶意代码检测工具检测到；同时由于这些安全脆弱性是处理器硬件本身的问题，很难通过软件补丁的方法对相关的攻击展开防御。所以，我们有必要在指令层面对我们使用的处理器进行系统的、全方面的测试，以发现其中存在的安全脆弱性，并分析这些安全脆弱性的威胁和可能带来的安全风险。

本论文主要聚焦对处理器中存在的隐藏指令和触发侧信道的指令组合的检测方法。现有的检测隐藏指令的方法存在着测试效率低、覆盖率不足、以及测试精确度不高的问题，针对上述问题，我们提出了一种测试方法 Skipscan。针对指令层面检测处理器中存在的侧信道的覆盖率不足（未考虑隐藏指令）和效率低的问题，我们提出了一种针对时间侧信道的高效、高覆盖率的检测方法 TSCMiner。本文具体包含以下创新点：

研究点一：为了提升对 x86 指令空间的测试效率，我们通过实时的检测指令长度的变化来控制深度优先遍历算法（Depth First Search，DFS）的搜索深度，从而跳过大量的无效指令空间；同时，我们提出了**合法指令操作数的最小测试集**的方法，减少了对冗余的含有立即数和偏移量的合法指令的测试。从而，达到了预留指令在所测试指令中的占比增加的效果，提升了对 x86 处理器中隐藏指令的测试效率。

研究点二：为了解决指令前缀域的测试覆盖率较低的问题，在研究点一提升测试效率的方法的基础之上，我们提出了**基于组合优化的指令前缀生成方法**。由于 x86 指令前缀是一组离散的值，我们将生成指令前缀组合的问题抽象成为对离散值的排列组合的生成问题。通过对现有的指令前缀的使用规则的分析，结合现有的排列组合的生成算法，我们提出了基于组合优化的指令前缀生成算法，

算法能够高效的、高覆盖率的生成 x86 指令前缀组合；将生成的指令前缀组合与预留操作码和预留操作数结合能够形成更多的预留指令空间，从而提升对 x86 处理器中隐藏指令的测试覆盖率。

研究点三：为了解决指令操作码和操作数域的覆盖率不足的问题，在研究点一提升测试效率的方法的基础之上，我们提出了**基于必要搜索深度实时检测的 DFS 算法**。通过对现有的 x86 指令格式的分析，我们总结出了每种指令格式对应的必要搜索深度。算法实时的检测指令空间的搜索深度，当检测到搜索深度的值小于必要搜索深度值的时候，算法就及时的增加指令空间的搜索深度为必要搜索深度和指令长度两个中的较小值，从而保证了对预留操作码和预留操作数的覆盖率，提升对 x86 处理器中隐藏指令的测试覆盖率。

研究点四：为了解决隐藏指令测试精度较低的问题，我们提出了**基于多种反汇编器交叉检查的方法**。我们将指令集相关应用的语义差异问题进行了分析和汇总，总结出了不同指令类型的评判标准。然后，我们将研究点一、二、三检测出来的隐藏指令通过多种反汇编器进行重新检查，将反汇编器的输出结果按照类别进行分组，并使用一致性系数标记每组的可信度。最后，我们结合一致性系数最高的反汇编器的输出结果与处理器的输出结果判断指令的类型，从而过滤掉了误报的指令，提高了 x86 处理器中隐藏指令的测试精度。

研究点五：为了提升在指令层面检测处理器中微架构脆弱性的效率，我们提出了**基于指令操作码分类的时间侧信道检测方法**。对微架构脆弱性的检测需要结合具体的威胁模型，本论文我们重点研究检测基于时间的侧信道的方法。首先，我们使用深度优先遍历的方法对 x86 的指令空间进行遍历，得到所需要测试的处理器中具备功能的指令，包括合法指令和隐藏指令；然后我们将指令进行反汇编，通过反汇编结果中的指令操作码（助记符）将指令进行分类，从而提升对共享计算资源引起的时间侧信道的搜索效率。然后，采用预置、触发、检测处理器微架构状态的三步测试方法将同类的指令进行组合测试，将能够形成侧信道的指令组合放入具体的攻击模型中进行测试，如果指令组合具备窃取信息的功能，则将其放入报告文件。

关键词：指令集，处理器脆弱性，隐藏指令，时间侧信道，微架构

Abstract

The processor is the final execution unit for all computations. An attacker can exploit the hardware resources in the processor to launch an attack on a computer system. These hardware resources that can be exploited by attackers are called security vulnerabilities in the processor, and they have a certain corresponding relationship with the instructions in the instruction system. Some vulnerabilities correspond to single instructions, such as hidden instructions and instruction flaws; some vulnerabilities correspond to combinations of instructions, such as cache side-channel attacks triggered by instruction combinations [1, 2] and covert channel attacks[3] corresponding to random number generators. The attacker can embed hidden instructions or a combination of instructions that can trigger side-channel attacks in the software to steal private information in the computer, cause the computer system to crash, and change the program execution behavior. It is difficult to detect such hardware attacks by existing malicious code detection tools; at the same time, because these vulnerabilities are problems of the processor hardware itself, it is difficult to defend against related attacks through software patches. Therefore, it is necessary for us to systematically test the processors we use at the instruction level and discover their security vulnerabilities, and analyze the threats of these security vulnerabilities and the possible security risks.

This paper mainly focuses on the detection method of hidden instructions in the processor and the combination of instructions that trigger the side channel. The existing methods for detecting hidden instructions have the problems of low test efficiency, insufficient coverage, and low test accuracy. To solve the above problems, we propose a test method Skipscan. Aiming at the problem of detection processor's side channel in the instruction-level is low coverage (hidden instructions are not considered) and low efficiency, we propose an efficient and high-coverage detection method TSCMiner for the timing-based side channel. This paper specifically includes the following innovations:

Research issue 1: In order to improve the test efficiency of the x86 instruction space, our method controls the search depth of the depth-first search algorithm (DFS) by detecting the change of instruction length in real-time, thereby skipping a large number of invalid instruction spaces; and we proposed a **minimum test set of legal instruc-**

tion operands approach which reduces the testing of redundant legal instructions with immediate and displacement operands. Therefore, our method achieves the effect of increasing the proportion of reserved instructions in the tested instructions and improves the test efficiency of hidden instructions on x86 processors.

Research issue 2: In order to solve the problem of low test coverage in the instruction prefix field, we propose a method of **instruction prefix generation that is based on the optimized combination**, which is based on the method of improving test efficiency in research issue 1. Since x86 instruction prefixes are a set of discrete values, we abstract the problem of generating instruction prefix combinations into the problem of generating permutations and combinations of discrete values. Through the analysis of the existing instruction prefix usage rules, combined with the existing permutation and combination generation algorithms, we propose an instruction prefix generation method based on the optimized combination algorithm, which can generate x86 instruction prefix combinations efficiently and with high coverage; combining the generated instruction prefix combination with the reserved opcodes and operands can form reserved instructions, thereby achieving the effect of improving the test coverage of hidden instructions on x86 processors.

Research issue 3: In order to solve the problem of insufficient test coverage of instruction opcodes and operand fields, we propose the method of **DFS algorithm based on the detection of essential search depth in real-time**, which is based on the method of improving test efficiency in research issue 1. Through the analysis of the existing x86 instruction format, we summarized the corresponding essential search depth. The DFS algorithm detects the search depth of the instruction space in real-time, when it is detected that the value of the search depth is less than the essential search depth, the DFS algorithm increases the search depth of the instruction space to the minimal value between essential search depth and instruction length in time, thus ensuring the improvement of opcodes and operations, thereby achieving the effect of improving the test coverage of hidden instructions on x86 processors.

Research issue 4: In order to solve the problem of low test accuracy of hidden instructions, we proposed a **method based on cross-checking multiple disassemblers**. We analyzed and summarized the semantic differences of instruction set-related applications, and summarized the criteria for evaluating instruction types. Then, our method

disassembles the hidden instructions detected in the primary stage through multiple disassemblers, groups the output results of the disassemblers into categories, and uses the consistency coefficient to mark the credibility of each group. Finally, we combined the outputs of the disassemblers with the highest consistency coefficient and the outputs of the processor to determine the type of instructions, thereby filtering out fault-positive instructions and improving the test accuracy of hidden instructions on x86 processors.

Research issue 5: In order to improve the test efficiency of detecting micro-architectural vulnerabilities in processors at the instruction level, we proposed **a timing-based side channel detection method based on instruction opcode classification**. The detection of microarchitectural vulnerabilities needs to be combined with specific threat models. In this paper, we focus on methods to detect timing-based side channels. First, we leveraged the depth-first traversal method to traverse the x86 instruction space to obtained the functional instructions in the processor that to be tested, including the legal instructions and hidden instructions; then we disassembled the instructions. Then we classified the instructions by using the opcodes (mnemonics) of the instruction, thereby improving the search efficiency of the timing-based side channel caused by the shared computing resources. Then, we leveraged the process of three-step testing: presetting, triggering, and detecting the state of the processor's micro-architecture to conduct a combination test of instructions in the same group, and put the combination of instructions that can form a side channel into a specific attack model for testing. If it has the function of stealing information, the instruction combination is placed into the report file.

Key Words: Instruction Set Architecture, Processor Vulnerability, Hidden Instructions, Timing-based Side Channel, Micro-architecture

目 录

图目录

表目录

符号列表

缩写

CISC	Complex Instruction Set Computer
RISC	Reduced Instruction Set Computer
ISA	Instruction Set Architectures
PC	Program Counter
TLB	Translation Look-aside Buffer
TSC	Timing-based Side Channel
x86	CISC ISA Developed by Intel and AMD (includes x86-64/AMD64)
AMD	Advanced Micro Devices
DFS	Depth-first Search
RNG	Random Number Generator
LLC	Last Level Cache
LFB	Line Fill Buffer
MDS	Microarchitectural Data Sampling

第 1 章　绪论

1.1　研究背景

我们的日常生活的正常运行依赖于国家的十几个关键基础设施部门，这些部门提供金融服务、公共卫生、通信和电力以及其他网络和系统相关的服务。这些关键基础设施都需要信息技术。随着信息化的不断发展，信息安全越来越受到人们的关注，同时也成为国家战略的组成部分。构建安全可控的信息技术体系是保障国家安全、国家和个人财产安全的重要方面。计算机系统是维持信息化正常运作的基本载体之一，计算机系统包括软件系统和硬件系统。安全可信的计算机系统称之为可信计算基（Trusted Computing Base, TCB），构建可信计算基需要考虑操作系统和其上面执行软件的安全，同时也需要考虑硬件处理器的安全。处理器是计算机系统中所有计算的最终执行单元，因此，安全可信的处理器是保障可信计算基的基础，同时也是保障安全可控信息技术体系的基石。

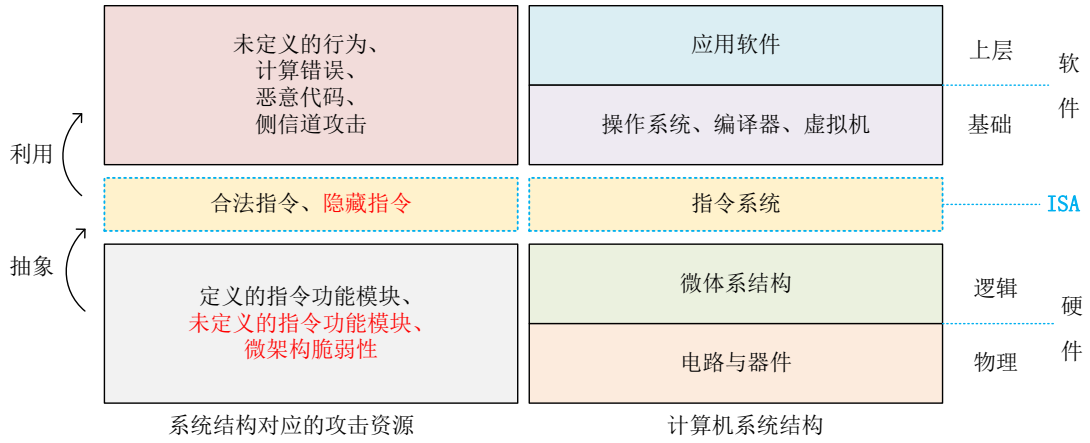

图 1-1　计算机的系统层次图以及对应的可被利用的攻击资源

Figure 1-1　The system hierarchy of computer and the corresponding exploitable attack resources

计算系统的安全不仅依赖于软件系统而且也依赖于处理器等硬件系统。处理器是计算机系统的运算和控制的核心部件。从处理器的诞生之日起，人们关心的更多的是处理器的性能、功耗、成本等参数。然而，近些年来不断曝光的处理器脆弱性（Vulnerability）[1]使得处理器安全也备受人们关注，例如，隐藏指令和

[1]The existence of a weakness, design, or implementation error that can lead to an unexpected, undesirable event compromising the security of the computer system, network, application, or protocol involved [4].

微架构脆弱性。隐藏指令的行为和功能未在处理器指令集文档中定义，但是它能够被处理器执行且不报非法指令异常。隐藏指令存在被攻击者利用发起攻击的可能，其根本原因是隐藏指令对应处理器中的硬件脆弱性。需要注意的是，隐藏指令与硬件木马存在区别 [5]，它们之间有小部分的交集（例如隐藏指令触发的硬件木马），但不存在相互包含的关系。隐藏指令可以是处理器不断修改和进化的过程中无意引入的功能，也可以是处理器设计人员故意设计的一个隐藏的功能模块，还可以是处理器厂商为调试处理器而专门设计的功能模块；这些功能在逻辑电路层面都有具体的实现（未定义的功能模块，如图1-1左侧下面框图中所示）。而硬件木马是处理器设计者故意加入的功能电路，它有多种表现形式，例如指令触发的硬件木马（单条指令和指令序列）[6–11]、组合电路型的硬件木马 [12]、时序电路型的硬件木马 [13, 14] 等。微架构脆弱性一般意义上是指能够被攻击者利用窃取计算机中的私密信息的控制通路和数据通路的子模块，它们在微架构层可被抽象为提升处理器性能的控制机制（乱序执行和推测执行等）以及执行（指令执行端口，图2-4中 Port）和存储数据（高速缓存、寄存器、缓存队列等）的功能模块；上述的处理器脆弱性的共同点是都能通过功能抽象表示在指令层面，而且，如图1-1中计算机系统结构所示，它们都能被操作系统或者应用软件所利用从而对计算机系统产生安全威胁。然而，由于隐藏指令和微架构脆弱性是处理器硬件层面的安全攻击面，开发者不能通过软件升级的方式对它们进行有效的防御。

隐藏指令会影响处理器的执行行为，使得系统不稳定，甚至使计算机系统对外拒绝服务；如果攻击者将它们注入到攻击代码中，将很难被研究人员通过反汇编器发现，同时也很难被恶意代码检测工具发现。例如，在 Intel 的奔腾处理器中的未公开指令 F0 0F C7 C8[2][15]，这条指令能够使处理器宕机，只有重启才能解决问题。最近，俄罗斯的研究人员在 Intel 的 Atom 处理器中发现了两条隐藏指令（0F 0E 和 0F 0F）[16]，它们能够完全的获得和控制处理器中的寄存器；虽然它们被证明是处理器中的调试指令，但是它们对处理器用户是不公开的。不仅单条的隐藏指令会对处理器的安全产生直接的威胁，它们还可能被用来发起侧信道攻击和组成恶意代码，而且很难被检测工具发现。有些隐藏指令是与其邻近的合法指令的衍生品，它们与合法指令有着相同的功能 [17]。攻击者可以利用它们发起基于时间争用的侧信道攻击 [18, 19]，而很难被检测到。因为，通常开发者会使用反汇编工具对二进制形式的攻击指令进行反汇编，然而，隐藏指令不会

[2]在本论文中我们使用十六进制的字节序列表示一条指令的机器码。

被反汇编工具识别到。同时，如果隐藏指令被注入到恶意代码中 [17]，也很难被现有的恶意代码检测工具发现。因为现有的恶意代码检测工具都是基于合法指令的组合表现出来的特征或者行为的检测。对于隐藏指令，恶意代码检测工具的设计者如果不知道其攻击特点，就不能设计出对应的检测机制。

处理器中存在的微架构脆弱性会被攻击者利用窃取计算机中的私密信息，而这些脆弱性对应的处理器中的控制通路和数据通路资源在过去是被用来提升计算机性能的重要方法。例如，2018 年初曝出的 Meltdown[1] 和 Spectre[2]，分别利用了处理器中的乱序执行机制和分支预测机制（这两种机制都是提升处理器的指令吞吐率的方法），同时结合 Cache 侧信道攻击的方法窃取计算机中的私密信息。而且这两种利用微架构脆弱性发起的侧信道攻击能够影响市面上大多数的处理器平台，包括 Intel、AMD 和 ARM 等。到目前为止，已经有超过 14 种的 Meltdown 漏洞的变种和超过 13 种的 Spectre 漏洞的变种被发布和认定 [20, 21]。起初 AMD 被认为是 Meltdown 免疫，但是后来也有发布能对 AMD 处理器发起攻击的变种。通常，利用处理器中的微架构脆弱性窃取计算机中的私密信息都要结合侧信道攻击的方法，较为常见的是 Cache 侧信道攻击。此外，还有与处理器微架构相关其他类型的侧信道攻击方式，例如，基于指令执行端口（PortSmash[18, 22]）、基于处理器中的数据缓存（TLBleed[23]）、基于随机数生成器共享（RNG[3]）等。可见，处理器中存在大量的可以被攻击者利用的微架构脆弱性，这对计算机系统的信息安全存在较大的威胁。

1.1.1　现有问题

针对处理器中可能存在的安全脆弱性，已经有一些自动化测试的研究工作，但是现有的检测方法的效率、覆盖率和精度都不高。接下来，我们分别面向自动化的检测隐藏指令和处理器微架构脆弱性的方法展开分析，并对不同的方法的优缺点进行阐述，最后概括出需要解决的具体问题。

对处理器中的隐藏指令的测试，不同指令集类型对应着不同的方法。一般而言，现有的处理器有两大类，即复杂指令集处理器（Complex Instruction Set Computer，CISC），其中较为常见的是 x86 处理器（Intel 和 AMD）；以及精简指令集处理器（Reduced Instruction Set Computer，RISC），其中较为常见的是 ARM、MIPS 和 RISC-V 等处理器。x86 处理器指令集其指令格式较为复杂，指令长度不定长，可以是 1 到 15 个字节，指令格式中包含有前缀、操作码、ModR/M、SIB、偏移量和立即数。x86 的指令空间最大可以是 2^{15*8}，在现有的 x86 处理器的运行速度的范畴内，无法实现对 x86 指令空间的全遍历。采用随机指令生成的方法

来测试 x86 处理器中存在的隐藏指令，其测试效率不高，也无法实现较高的覆盖率。因为，随机指令生成的方法会产生大量的含有立即数和偏移量的冗余的合法指令，且无法保证对指令前缀域和操作码等关键字节部分的全部覆盖。为了解决全遍历 x86 指令空间不可实现以及随机的方法测试效率和覆盖率不高的问题，Sandsifter[24] 首次使用深度优先遍历算法（Depth-first Search，DFS）对 x86 指令空间进行搜索；它通过实时的观测指令长度的变化和搜索深度所在字节的值的变化来控制 DFS 算法的搜索深度，从而跳过了大量的无效指令；相比于全空间遍历的方法，Sandsifter 提升了测试效率，而且一定程度上能保证对操作码和操作数域的覆盖。但是，Sandsifer 存在以下的一些不足：第一，它测试了大量的冗余的含有偏移量和立即数的合法指令，从而使得其测试效率较低；第二，它没有对指令前缀域进行系统的测试，从而使得对 x86 指令空间的覆盖率较低；第三，它只通过观测指令长度的变化来增加 DFS 算法的搜索深度，这使得它的测试深度不足，从而导致对操作码和操作数域的覆盖率较低；第四，它只使用了单一的反汇编器 Capstone 作为与处理器运行结果作为比对的参考对象，这使得其测试精度较低。UISFuzz[25] 尝试提升 Sandsifter 的测试效率，但是它在测试的过程中丢失了对部分预留指令的测试，这使得它的覆盖率相比于 Sandsifter 而言有所降低（详见第二章）。总的来看，采用深度优先遍历算法来测试 x86 处理器的指令空间，能够较好的兼顾测试覆盖率和效率，但是我们需要进一步的探索提升测试的覆盖率、效率和精度的方法。

测试 RISC 指令集处理器中的隐藏指令，一般使用全空间遍历的方法。RISC 指令集中指令长度一般为 4 个字节，其指令空间最大可以是 2^{4*8}，为了达到较高的覆盖率，现有的研究工作 [5, 17, 26, 27] 使用了基于遍历的方法。为了提升测试效率，Examiner[27] 采用了只测试立即数和偏移量操作数的边界值的方法，从而跳过了大量的冗余的立即数和偏移量操作数。Wang 等 [5] 提出了只测试未定义指令（Undefined Instructions）的方法，因为他们的目标是发现处理器中的隐藏指令。IDev[17] 采用了指令域精简的方法，其目的是为了跳过无效的指令。现有的针对 RISC 处理器指令集的测试方法已经较为完善。所以，本论文不重点研究针对 RISC 处理器中隐藏指令的测试方法。

针对处理器中存在的微架构脆弱性的检测，已经有一些自动化的测试方法，但现有的方法少有考虑隐藏指令可能带来的侧信道攻击，而且其检测微架构脆弱性的效率较低。SPEECHMINER[28] 通过分析现有的利用微架构脆弱性发起攻击的代码，将代码归纳分类并归纳为几种代码框架，然后使用能够触发处理

器异常的指令对代码框架中的指令进行替换，并观察新的攻击代码的执行对处理器状态的影响，从而发现新的攻击代码；但是这种方法只能发现特定的攻击类型的指令组合。Osiris[29] 从指令序列的角度出发检测处理器中能够发起时间侧信道攻击的微架构脆弱性，作者提出了一种预置、改变和观察处理器微架构状态的方法来检测可能存在的微架构脆弱性。Osiris 通过自动化的产生一组指令序列，然后测试指令序列是否能够形成一个基于时间的侧信道（Timing-based Side Channel），通过在多种 x86 处理器中的实验评估，它最后发现了 4 个能够被利用的隐蔽信道（Covert Channel）。ABSynthe[19] 是通过机器学习的方法自动化的检测特定的加密软件在特定的处理器硬件上运行是否存在可能的基于竞争（Contention-based）的侧信道攻击，它针对不同的处理器产生对应的泄露图例（Leakage Maps），从而发现在特定的处理器上泄露特定软件私密信息效果最好的指令序列。Revizor[30] 使用基于模型关系的模糊测试（Model-based Relation Fuzzing）的方法测试处理器中的微架构脆弱性，此方法基于一个投机合约 [31]（合约是一种增强指令集架构的规范，它描述攻击者可以通过侧信道观察到哪些 CPU 操作，以及哪些操作可以投机性地改变控制或者数据流），将合约在对应的模拟器中实现。然后，将随机生成的指令序列同时送入处理器（Hardware Traces）和模拟器 (Contract Traces)，观察处理器和模拟器执行指令序列后的行为，从而发现与合约行为不一致的指令序列，这些指令序列就对应一个处理器微架构脆弱性。总的来看，对处理器微架构脆弱性的检测的方法比较多样，这源于 x86 处理器微架构的复杂性，以及攻击形式的多样性；例如，利用不同的控制通路和数据通路的组合就能形成不同的威胁模型，而不同的威胁模型对应不同的检测方法。本论文聚焦研究自动化的检测处理器中指令序列触发的时间侧信道的方法。

综上所述，为了减少计算机系统中由于处理器安全脆弱性被攻击者利用带来的安全风险，本论文重点展开对处理器中存在的隐藏指令和指令序列触发的处理器微架构脆弱性的自动化检测方法进行系统的研究。

1.2　研究内容与创新性

根据上一节对处理器中存在的安全脆弱性危害的分析以及对现有的研究方法的阐述和总结，接下来我们展示本论文使用的方法以及创新点。

指令集是计算机硬件和软件之间的接口，是软硬件交互的界面，有着非常关键的作用。计算机系统可以概括的分为四个层次：应用软件、基础软件、硬件电路和物理载体。软件以指令的形式运行在 CPU 硬件上，而指令系统介于软件和

硬件之间。计算机系统中所有软件层的操作，最终都将转化为处理器硬件的操作。软硬件本身的更迭速度很快，而指令系统则可以保持较长时间的稳定，程序员根据指令系统设计软件。在指令层面展开对处理器的测试具有较好的可行性，同时研究出来的方法和工具可以长时间被使用。所以，本论文在指令层面研究针对处理器中安全脆弱性的检测方法。

脆弱性：脆弱性可以是硬件本身的弱点，也可以是硬件上运行的软件的弱点[32]。系统的复杂程度的增加、对硬件和软件的测试的不充分、软硬件设计的过程中的缺陷等都会增加计算系统的脆弱性发生的概率 [32]。美国的国家标准技术局（National Institute of Standards and Technology, NIST）发表的白皮书 [33] 这样定义脆弱性：系统安全程序、设计、实施或内部控制中的缺陷或弱点可能会被执行（意外触发或故意利用）并导致安全漏洞或违反系统的安全策略。欧盟网络安全机构（European Union Agency for Cybersecurity）这样定义脆弱性：存在的弱点、设计或实施错误可能会导致意外的不良事件危及计算机系统、网络、应用程序或所涉及协议的安全 [4]。现有的定义更倾向于认为脆弱性是一种带来危害的可能性，一般用来形容可以被不同的攻击类型所利用的硬件或者软件资源。一般的，提及脆弱性，我们一般会想到 Meltdown[1] 和 Spectre[2]，它们都是瞬态执行处理器脆弱性 [34, 35]，属于微架构脆弱性的范畴。隐藏指令是对硬件脆弱性在指令层面的抽象。值得注意的是，现有的研究工作将隐藏指令作为脆弱性[16, 25]、安全脆弱性 [36]、指令集脆弱性 [37]、指令引起的硬件脆弱性 [5] 等。理想情况下，处理器只应该提供指令集中定义的合法指令的功能。处理器中存在的隐藏指令是处理器的执行行为与指令集中的定义不一致，隐藏指令存在被攻击者利用而发起攻击的可能性，所以，本论文我们将其归类为指令的安全脆弱性的范畴。总的来看，我们本论文研究的处理器安全脆弱性包括隐藏指令对应的硬件脆弱性和指令序列对应的微架构脆弱性。

处理器指令安全脆弱性检测：本论文主要研究在指令级（相比于微架构级、寄存器传输级、逻辑门电路级等）研究处理器中存在的脆弱性的检测方法。处理器中的脆弱性在微架构层面或者逻辑电路层面可以表现为未定义的指令功能模块，以及能够被攻击者利用发起侧信道攻击的微架构功能模块；在指令层面，它们可以被抽象成为隐藏指令和指令序列。我们要从指令的自动化生成和测试的方法出发，将生成的指令或者指令序列放入处理器中执行，从而发现隐藏指令和能够触发微架构脆弱性的指令序列。值得注意的是，本论文重点关注的是对检测技术的研究，而对于检测出来的脆弱性的攻击场景的研究和应用，需要结合具体

的攻击类型展开，我们本论文只对部分攻击场景进行了分析。

研究目标：从指令层面检测处理器中存在的隐藏指令和处理器微架构脆弱性，旨在减小处理器的安全攻击面，提升计算机系统的安全性。首先，本论文着力于研究隐藏指令的检测算法，方法需要尽可能的搜索和测试所有的预留指令，因为隐藏指令是具备某种功能的预留指令。通过分析 x86 指令的前缀、操作码和操作数等不同字节域的特点，研究高效率、高覆盖率的搜索 x86 指令空间的方法；同时还要保证对隐藏指令检测的精度。其次，本论文结合检测出来的隐藏指令以及指令集中已有的合法指令，研究高效的检测指令序列引起的基于时间的侧信道的方法。

研究方法：针对以上的研究目标，我们结合上一小节提炼出的相关问题，列出了对应的解决方法和创新点。

第一，针对现有的隐藏指令检测方法**测试效率不高**，测试了大量的冗余的含有立即数和偏移量操作数的合法指令的问题，我们提出了合法指令中立即数和偏移量操作数的**最小测试集**的方法，方法减少了对合法指令中冗余的立即数和偏移量操作数的测试，从而减少了合法指令在所测试的总的指令中的占比，提高了预留指令在所测试的总的指令中的占比，所以提升了对预留指令的测试效率。

第二，针对现有的隐藏指令检测方法对**指令前缀域测试覆盖率低**的问题，我们通过分析 x86 指令前缀域的特点，提出了**基于组合优化的指令前缀生成**的方法，增加了对前缀长度在 2 到 4 个字节之间的情况的测试，方法能够高效的产生现有的指令前缀规则条件下所有的合规的指令前缀组合，增加了对预留指令的测试覆盖率（增加了指令前缀组合与预留操作码和预留操作数的组合的测试），从而增加了对隐藏指令的测试覆盖率。

第三，针对现有的隐藏指令检测方法对**操作码和操作数域的测试覆盖率低**的问题，我们通过分析现有的合法指令格式，总结了每种指令格式对应的必要搜索深度，提出了**基于必要搜索深度实时检测的 DFS 算法**；当算法检测到指令的搜索深度小于必要搜索深度的时候，它就会及时增加搜索深度到必要搜索深度和指令长度两者中的较小值，从而保证了对指令预留操作码和预留操作数域的覆盖，达到了增加隐藏指令的搜索覆盖率的效果。

第四，针对现有的隐藏指令检测方法**测试的精度低**的问题，我们提出了**基于多种反汇编器的交叉检测**方法，方法使用多种反汇编器对测试出的指令进行重新检查。我们将多种反汇编器的输出进行规范化处理，同时使用一致性系数来衡量反汇编器输出结果的可信度，一致性系数越高，则可信度越高。将可信度最高

的反汇编器输出结果与处理器的执行结果进行对比，作为最终的指令类型的判定条件，过滤掉了误报的指令，从而降低测试的误报率，达到了提升隐藏指令测试精度的效果。

第五，针对现有的处理器微架构脆弱性**自动化检测效率低**的问题，我们提出了**基于指令操作码分类的时间侧信道检测**的方法。对微架构脆弱性的检测需要结合具体的威胁模型，本论文我们重点研究检测基于时间的侧信道的方法。首先，我们使用深度优先遍历的方法对 x86 的指令空间进行遍历，得到所需要测试的处理器中具备功能的指令，包括合法指令流和隐藏指令；然后我们将指令进行反汇编，通过反汇编结果中的指令操作码（助记符）将指令进行分类，隐藏指令与其相近的合法指令归为一类，从而提升对共享计算资源引起的时间侧信道的搜索效率。然后，采用预置、触发、检测处理器微架构状态的三步测试方法将同类的指令进行组合测试，将能够形成侧信道的指令组合放入具体的攻击模型中进行测试，如果指令组合具备窃取信息的功能，则将其放入报告文件。

1.3 研究意义

理论意义：高效的处理器指令空间的搜索方法是检测与指令集相关的应用需要共同面对并解决的问题，例如，处理器、CPU 模拟器、ISA 模拟器和解码器等。本论文主要从指令层面出发，研究高效的指令空间搜索方法来测试处理器中存在的隐藏指令和微架构脆弱性，所提出的研究方法对检测模拟器和解码器也有较好的理论参考意义。

现实意义：对处理器的性能评估，现在已经有较为成熟的测试工具和测试例集合。然而，对于处理器中存在的隐藏指令和微架构脆弱性等脆弱性的测试，成熟的第三方的检测工具还较少。这对于处理器的使用者而言，很难全面的了解和评估处理器的安全性。本研究对处理器安全漏洞检测和评估具有较为实用的现实意义。

1.4 论文内容组织结构

如图1-2所示为本学位论文的结构图。第一章介绍了本论文的研究背景、问题、研究内容和创新点。第二章介绍了现有的研究方法，同时阐述了对应研究方法的不足，进而总结出对应的研究问题。第三章介绍了基于合法指令中操作数的最小测试集的方法，展示了**研究点**一的具体的研究方法和评估结果；值得注意的是，本章节的方法与现有的方法相比，只是提升了测试效率，并没有发现新类型

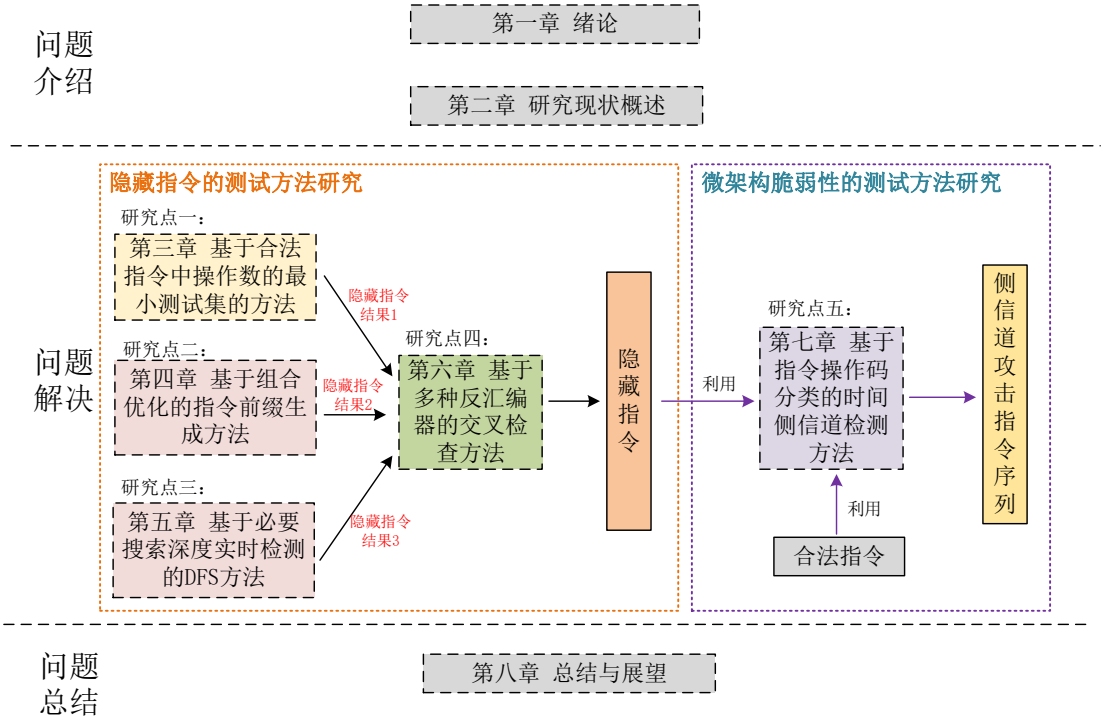

图 1-2　学位论文结构图

Figure 1-2　Thesis Architecture

的隐藏指令。第四章介绍了基于组合优化的指令前缀生成方法，展示了**研究点**二的具体的研究方法和评估结果；与现有的方法相比，本章节描述的方法在研究点一提升效率的基础之上，增加对指令前缀域的系统的测试，从而增加了 x86 指令空间的覆盖率，发现了更多的新类型的隐藏指令。第五章介绍了基于必要搜索深度实时检测的方法，展示了**研究点**三的具体研究方法和评估结果；与现有的方法相比，本章节描述的方法在研究点一提升效率的基础之上，保证了对指令操作码和操作数域的覆盖，从而增加了对 x86 指令空间的覆盖率，发现了更多的新类型的隐藏指令。值得注意的是，研究点一、研究点二和研究点三可以综合应用，从而可以在保证测试效率的基础之上，进一步的提升测试的覆盖率。第六章介绍了基于多种反汇编器的交叉检查方法，展示了**研究点四**的具体的研究方法和评估结果；本章节对第三、四、五章检测出来的隐藏指令进行重新检查，过滤掉误报的隐藏指令，从而提升对 x86 处理器中隐藏指令的检测精度。第七章介绍了基于指令操作码分类的时间侧信道检测方法，展示了**研究点五**的具体的研究方法和评估结果；本章节利用第五章确定了的隐藏指令，结合所测试处理器指令集中的合法指令，形成指令组合，通过对指令组合的测试，从而发现指令序列触发的微架构脆弱性。第八章展示了我们对论文的总结和对未来工作的展望。

第一章，阐述了处理器脆弱性的研究背景、现有的问题、本论文的研究内容和创新点、以及研究的意义。

第二章，展示了论文所需要的基础知识；分析了隐藏指令检测方法和微架构脆弱性检测方法的发展现状，对现有的方法做了详细的介绍和分析，阐述了它们的优点和不足之处；最后凝练出本论文需要研究的科学问题。

第三章，详细阐述了合法指令中操作数的最小测试集的方法，减少了对冗余的立即数和偏移量操作数的测试，提升了预留指令在所测试的指令中的占比。描述了测试所需要的差分测试框架，并将方法在 8 种 x86 处理器上进行了评估测试，目的是验证提出的方法提高了对隐藏指令的测试效率。

第四章，详细阐述了基于组合优化的指令前缀生成的方法。方法增加对合规的指令前缀组合的覆盖，从而覆盖了更多的指令前缀组合和预留操作码、预留操作数所构建的预留指令。本章节描述了测试所需要的差分测试框架，并将方法在 8 种 x86 处理器上进行了实验评估，验证了所提出的方法提升了对隐藏指令的测试覆盖率。

第五章，详细阐述了基于必要搜索深度实时检测的 DFS 方法。方法增加对 x86 指令空间的搜索深度，从而增加了对指令预留操作码和预留操作数的覆盖率。本章节描述了测试所需要的差分测试框架，并将方法在 4 中 x86 处理器上进行了实验评估，验证了所提出的方法提升了对隐藏指令的测试覆盖率。

第六章，详细阐述了基于多种反汇编器的交叉检查方法，通过使用多种反汇编器对测试阶段检测出的隐藏指令进行重新测试，从而提升隐藏指令的测试精度，降低误报率。

第七章，详细阐述了基于指令操作码分类的时间侧信道的检测方法，方法首先对合法指令和隐藏指令的操作码进行分组处理，从而提升了对时间侧信道的搜索效率，然后对指令组合进行测试，发现可能存在的处理器微架构脆弱性。我们将方法在 4 种 x86 处理器上进行了评估，不仅发现了已有的基于时间的侧信道攻击形式，还发现了新的时间侧信道攻击类型。

第八章，对全文进行了总结与展望，总结了本论文的主要的创新点和贡献，并提出了未来的研究方向和挑战。

第 2 章　处理器脆弱性研究现状概述

本章节首先展示了现有的精简指令集处理器（Reduced Instruction Set Computer, RISC）和复杂指令集处理器（Complex Instruction Set Computer, CISC）指令集架构的特点和处理器微架构相关的知识。然后，展示了针对不同处理器的隐藏指令的搜索方法和微架构脆弱性的搜索方法，并总结了现有的工作的优点和不足之处。最后，提炼出了需要解决的问题。

2.1　指令相关概念

- 合法指令：在指令集手册中定义，开发者能够公开获取和使用的指令，且处理器有对应的功能实现。合法指令在 RISC 和 CISC 指令集中被定义，是对处理器功能的抽象，例如 Add、Sub、Mov 等指令。
- 隐藏指令：没有在指令集手册中定义，但是处理器能够执行且不报非法指令异常。隐藏指令在 RISC 和 CISC 指令空间中都可能存在，隐藏指令不对外公开，有可能是处理器厂商故意加入的后门，也有可能是在处理器设计和生产过程中无意引入的功能。
- 无效指令：不能形成有效的合法指令编码和预留指令编码的指令，只存在于 x86 指令空间中，由于 x86 指令是变长指令，可以是 1 到 15 个字节，指令长度以外的字节为无效指令字节。由于 RISC 指令集中指令都定长，一般为 4 个字节，所以，RISC 指令空间中只有合法指令和预留指令，不存在无效指令。

2.2　指令集架构

指令集架构（Instruction Set Architecture, ISA）又简称为指令集或者指令集体系，是计算机体系结构中程序设计的有关部分，包含了基本数据类型、指令集、寄存器、寻址模式、存储体系、中断、异常处理以及外部 I/O。指令集架构包含一系列的操作码，以及由特定处理器执行的基本命令，不同的处理器“家族”有不同的指令集架构 [38]。

现有的处理器根据指令集类型可以分为两种。第一类是 CISC 指令集处理器，其中较为出名的是 x86 处理器，例如，Intel、AMD、兆芯、海光和 VIA 等。CISC 是变长指令长度的指令集，其指令可以是 1 到 15 个字节。CISC 指令集的发展已经有 40 多年的历史，从 1976 年的 8086 开始，x86 的 ISA 就开始不断的

扩展，而且始终保持向后的兼容性，这使得古老的 x86 软件仍然可以在较新的处理器上运行。然而，这也导致了 x86 处理器的指令集变得冗余，其中也包含一些已经不再使用的功能。第二类是 RISC 指令集处理器，例如，MIPS、ALPHA、ARM、RISC-V 等。相比于复杂的 CISC 处理器，RISC 处理器指令集最显著特点是指令长度固定，为 4 个字节。

CISC 处理器和 RISC 处理器面向的使用场景有一定的差异，这与其性能、功耗、价格等因素相关。x86 处理器一般是面向台式机和服务器等场景而设计的，其主要特点是高性能、高功耗、单个处理器价格昂贵。对高性能的优化需要使用指令级并行，例如多发射、乱序执行、分支预测等技术；以及线程级并行，例如同时多线程，多核心处理器等技术；这也使得处理器微架构变得较为复杂。RISC 处理器主要面向的是嵌入式设备，例如单片机、手机、可穿戴设备和物联网（Internet of Things, IoT）设备等，其显著提点是低功耗和单个处理器价格便宜；而且，近年来随着 ARM 进军服务器市场，ARM 指令集架构也有其高性能版本，这也使得 ARM 处理器的指令集变得更加的丰富。

隐藏指令和微架构脆弱性的根本原因是处理器设计或者流片过程中引入的问题，所以 CISC 处理器和 RISC 处理器都不可避免的会出现这两种脆弱性。然而，在指令层面对 CISC 处理器和 RISC 处理器中的脆弱性的测试，由于指令集的复杂程度不同，所需要解决的具体问题会存在差异。CISC 处理器虽然指令集种类较少，一般都使用 x86 的指令集格式，但是由于其指令格式本身的复杂性，最直观的就是 x86 指令可以是 1 到 15 个字节，导致其潜在的指令编码空间巨大，这使得对其指令空间的研究变得困难；同时由于 x86 处理器本身微架构的复杂性也使得对 x86 处理器中微架构脆弱性的测试变得困难。RISC 处理器指令集的特点是种类较多，针对特定的指令集处理器相对容易展开测试，例如使用全遍历的方法，但是不同的 RISC 处理器平台之间存在着测试软件环境不兼容的问题。

接下来，我们展示 x86 处理器和 RISC 处理器的基本指令格式以及 x86 处理器微架构的知识。

2.2.1 x86 指令格式简述

x86 指令集是在 1978 年推出的，是目前为止较早的一种指令集系统。如图2-1（a）所示，x86 指令的指令长度可以是 1 到 15 个字节，其基本指令格式包含有前缀（Prefix）、操作码（Opcode）、ModR/M、SIB、偏移量（Displacement）、立即数（Immediate）等。如图2-1中（b）、（c）、（d）、（e）所示，现有的指令格式从

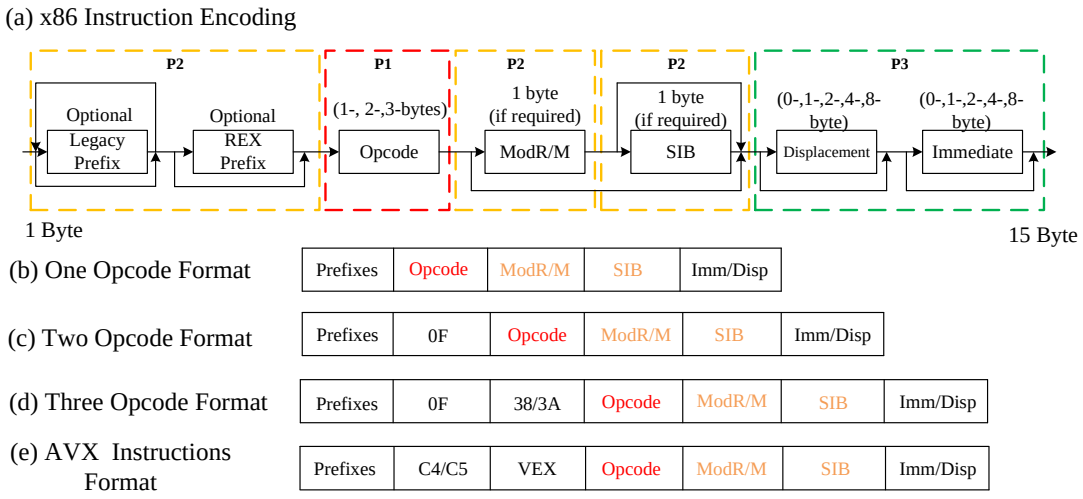

图 2-1　x86 处理器指令的基本格式

Figure 2-1　The basic instruction format of x86 processor

形式上可以分为四种，操作码（Opcode）的长度为 1 字节，2 字节和 3 字节，以及 VEX 指令类型。我们根据指令域对指令功能的影响程度来对其优先测试权限的等级进行划分。

优先级一（P1）：操作码域。操作码域是组成指令必须要有的，因为它被解码成指令对应的控制通路的信号。同时，它还可以被用来定义指令的行为、偏移量的尺寸、寄存器编码、条件码和符号扩展等。值得注意的是，有些指令操作码中的三个比特会被定义在 ModR/M 字节中。

优先级二（P2）：前缀域、ModR/M 域和 SIB 域。这几个指令域是可选的，可以与操作码结合来扩展指令的功能。例如，Legacy 前缀（从 Group 1 到 Group 4）、REX 前缀、Escape 前缀（0x0F、0x0F38 和 0x0F3A）、以及 VEX 前缀（0xC4 和 0xC5）等。随着指令集的发展，有些 Legacy 前缀会被当作强制（Mandatory）前缀来使用，例如，前缀 0x66，0xF2 和 0xF3。以指令 CVTDQ2PD 为例，它的机器码包含如下的字节 0xF3 0F E6，这里的 0xF3 就是一个强制前缀，而不是一个 Legacy 前缀（0xF3 使后面的指令重复执行），它的目的是扩展 0x0F E6 操作码的功能。一些指令的操作数在内存中，就需要一个地址格式认定字节，这就是 ModR/M 字节。其中，ModR/M 字节包含三个子域的信息，也就是 mod（2 个比特）、reg/opcode（3 个比特）和 r/m 域（3 个比特）。第一个，mod 域与 r/m 域结合可以形成 32 种值，即 8 个寄存器值和 24 种地址模式。第二个，reg/opcode 域可以辅助认定一个寄存器序号或者表示操作码中的三个比特位，此三个比特位的功能在主操作码中指定。第三个，r/m 域可以辅助指定一个寄存器为操作数或

者与 mod 域结合编码一种寻址模式。有些时候，固定的 mod 域和 r/m 的组合可以用来表示一些指令的操作码信息。有些 ModR/M 域的在使用的时候需要 SIB 字节，即在 base-plus-index 和 scale-plus-index 模式的 32 比特内存寻址的情况下需要 SIB 字节。需要说明的是，这里的 scale 域指的是比例因子（2、4 或者 8），index 域指的是 index 寄存器的序号，base 域指的是 base 寄存器的序号。

优先级三（P3）：偏移量和立即数域。这两个指令域是可选的，它们不能被用来决定指令的功能，所以提升指令空间搜索效率的时候，采用跳过冗余的立即数和偏移量操作数的方法。有些寻址模式需要一个偏移量（Displacement）紧随 ModR/M 字节或者是 SIB 字节。如果指令需要偏移量或者立即数（Immediate），它可以是 1 个、2 个或者 4 个字节。

2.2.2 RISC 指令格式简述

RISC 指令的格式较为多样，目前较为常见的是 ARM、RISC-V 和 MIPS。我们只展示比较规整的 MIPS 和 RISC-V 的指令格式，ARM 的指令格式种类较多，且没有固定的格式定义，可以参考其指令手册。

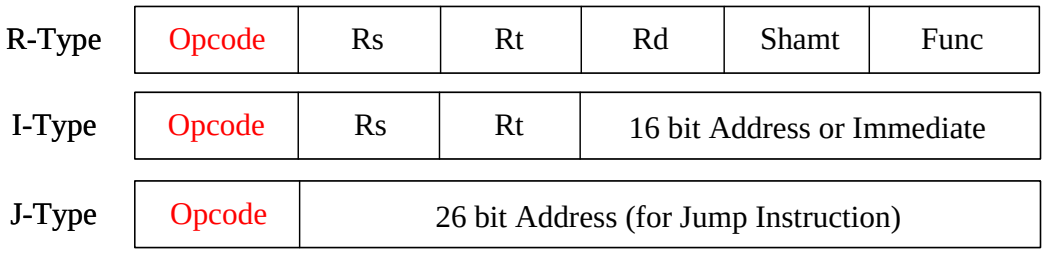

图 2-2　MIPS 处理器指令的基本格式

Figure 2-2　The basic instruction format of MIPS processor

MIPS 架构是在 1985 年推出的指令集，其指令集中有三种指令格式，R 型指令、I 型指令和 J 型指令，分别对应寄存器-寄存器类型的操作、寄存器-立即数类型的操作和跳转或调用操作，如图2-2中所示。这三种指令格式的操作码（Opcode）都是 6 个比特的，代表寄存器的比特域 Rs、Rt 和 Rd 都是 5 个比特的，可以表示 32 个不同的寄存器号。其中 Rs 代表第一个源操作数寄存器号，Rt 代表第二个源操作数寄存号，Rd 代表目的操作数寄存器号。另外，R 类型的指令中的 Shamt 代表位偏移量，仅在位移指令中使用，Func 代表指令函数码（有的也称之为辅助操作码），用于选择操作码中的具体函数（区别同样指令操作码中的子类）。

RISC-V 指令集是由加州大学伯克利分校在 2010 年推出的，有六种指令格式，如图2-3所示，分别为 R 类型指令、I 类型指令、S 类型指令、B 类型指令、U

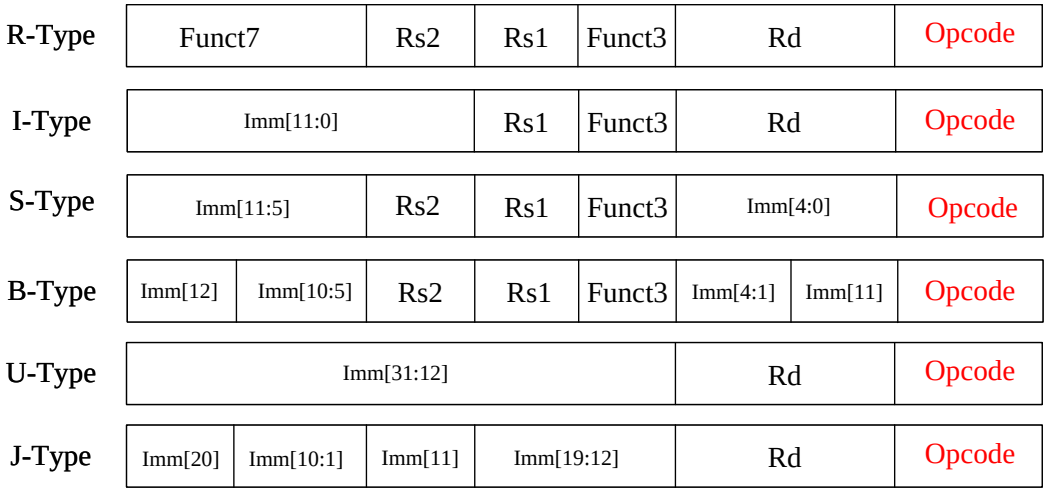

图 2-3　RISC-V 处理器指令的基本格式

Figure 2-3　The basic instruction format of RISC-V processor

类型指令和 J 类型指令，分别对应寄存器-寄存器操作、短立即数和访存 load 操作、访存 store 操作、条件跳转操作、长立即数操作和无条件跳转操作。RISC-V 指令集架构的操作码放在了指令编码的低比特位，其操作码所在的位置与 MIPS、ARM 和 ALPHA 等指令集架构有较大的差异。RISC-V 的指令架构相比于 MIPS 做了对齐优化，使得处理器的解码器的设计得到了简化。这六种指令格式的操作码都是 6 个比特的，代表寄存器的比特域 Rs1、Rs2 和 Rd 都是 5 个比特的，可以表示 32 个不同的寄存器号。其中 Rs1 代表第一个源操作数寄存器号，Rs2 代表第二个源操作数寄存号，Rd 代表目的操作数寄存器号。另外，指令中的 Func7 和 Func3 代表指令函数码（与 MIPS 指令格式中的 Func 功能一致），用于选择操作码中的具体函数。Imm 代表指令所需要立即数的数据存放在指令中的位置。

x86 处理器的复杂性最为直观的表现是其指令的编码格式较为复杂，而 RISC 处理器的复杂性在于指令集的多样性。单从指令格式方面来看，x86 指令集处理器与 RISC 指令集处理器最直观的差别在指令长度，x86 的指令长度可以是 1 到 15 个字节，而 RISC 的指令长度一般为 4 个字节。此外，x86 的指令编码更加的复杂，其寄存器号的编码被隐藏在操作码和 ModR/M 中，相比较而言 RISC 处理器指令集有直观的寄存器域。x86 指令集的优势是多个处理器厂商遵循同样的编码形式，然而，RISC 平台的处理器遵循不同的指令集编码形式。研究 x86 处理器中隐藏指令的搜索，最大的挑战是其 x86 指令空间巨大，无法通过遍历的方法进行测试，需要重点研究指令编码的特点，从而开发高效、高覆盖率的测试方法。研究 RISC 处理器中的隐藏指令，一般使用遍历的方法可以完成，但需要针

对特定的处理器指令集开发特定的测试代码。

2.3 处理器微架构简述

微架构（Microarchitecture）也被称之为计算机组织，微架构使得指令集架构可以在处理器上执行 [39]。微架构通常被抽象表示成流程图，以描述处理器内部控制通路（Control Path）和数据通路 (Data Path) 的各个组件的连接关系。流程图连接高速缓存（Cache）、解码器 (Decoder)、算数逻辑单元（ALU）、控制单元 (Scheduler)、缓存 (Buffer) 等组件。

指令集架构可以在不同的微架构上执行。Intel 和 AMD 虽然都遵循 x86 指令集架构，但是有着不同的微架构。例如，Intel 的微架构有 Core、Nehalem、Sandy Bridge、Haswell、Skylake、Kaby Lake、Coffee Lake、Comet Lake、Ice Lake 等；AMD 的微架构有 K10、Bulldozer、Zen、Zen 2、Zen 3 等。接下来我们展示较为经典的 Skylake 微架构，并讨论微架构各个部件之间的关系。

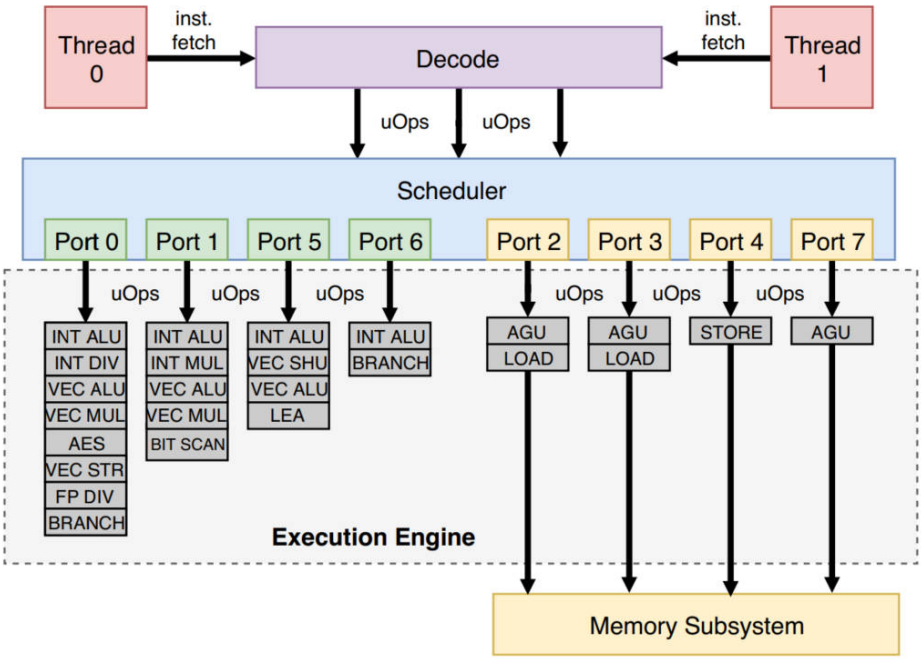

图 2-4 Skylake 的微架构

Figure 2-4 Microarchitecture of Skylake

一般的，处理器中的流水线模型可简单划分为三个阶段：取址阶段、译码阶段、执行阶段。如图2-4所示，两个逻辑核心共享一个执行单元，也就是同时多线程（Simultaneous Multi-Threading, SMT）技术。两个进程中的指令被从指令 Cache 中取出，取出后的指令会被解码器解码为微操作码，然后微操作码会被放

入发射队列等待，当寄存器中的值可获得且所需要的执行端口处于空闲的条件下，则执行此微操作码。端口具体的实现整型计算、浮点型计算、分支操作、取数据操作、存储数据操作等功能。在多发射处理器中，微操作码以乱序执行的形式在流水线中被执行，然后按照指令的顺序提交。

执行端口是执行单元的一个通道，也就是微码的最终执行功能单元。在微码的调度阶段，微码会被动态分配到空闲的可以执行的端口去执行，以实现最大化的指令吞吐率。为了突出执行阶段动态调度的端口分配细节，我们使用图2-4来抽象的概括执行阶段的流水线功能，图中使用灰色的方块表示执行端口的功能。端口 0、1、5 和 6 可以被用来执行算术指令，它们都是 ALU 单元中的功能子模块；端口 2、3、4 和 7 代表内存相关的微操作，例如端口 2 具体的实现 load 指令的功能，端口 4 实现 store 指令的功能。

2.4　指令的测试方法

理想条件下，x86 指令集的指令空间中有合法指令、预留指令和无效指令。合法指令是在指令集中定义的指令编码，开发者可以公开的获得合法指令，并用它们设计程序。预留指令是指令集中未定义的指令编码，主要包括未定义的操作码（Undefined Opcodes）和未定义的操作数（Undefined Operands, 例如 x86 指令集中预留的 ModR/M 和 SIB 字节)，它们可以被用来扩展新的指令功能。隐藏指是具备某种功能的预留指令。无效指令是因 x86 的指令长度最大值为 15 个字节的约定而产生的，在合法指令已经被定义了的前提下剩余的未被使用的字节对此合法指令是无效的编码空间。例如，某合法指令的长度为 4 个字节，其指令长度小于 15 个字节，那么剩余的 11 个字节空间就是无效的指令编码空间，简称为无效指令。

理想情况下，RISC 指令集的指令空间中只有合法指令和预留指令。RISC 指令集中没有无效指令空间的最主要的原因是 RISC 指令集中指令的长度都为 4 个字节。

对指令集相关应用开展测试的研究人员会过滤掉对测试不需要的指令类型，从而在不影响所需指令覆盖率的前提下提升测试的效率。需要注意的是合法指令空间和预留指令空间是相互交错的，无法只搜索其中一种类型的指令，但可以通过反汇编器或者指令库等工具对生成的指令进行筛选，从而获得需要测试的指令类型。在对处理器指令功能验证、CPU 模拟器的测试和反汇编工具的测试的工作中，一般的，研究人员关注的是合法指令功能是否符合指令集规范中定义

的指令行为，所研究的方法重点关注的是对合法指令的搜索和测试，应尽量跳过预留指令和无效指令。现有的研究人员已经提出了多种验证合法指令功能正确性的方法，工业界的研究人员发表了多篇论文，例如 Intel 处理器相关的 [40]，[41]，[42]，[43]，[44]，IBM 相关的 [45]，[46]，以及 ARM 相关的 [47]。学术界也发表了相关的一些处理器合法功能测试和验证的论文，例如 [48]，[49]，[50]，[51]，[52]，[53]。而且，不仅是针对处理器中的合法指令功能的验证，对于 CPU 模拟器和指令反汇编工具等的测试也有相关的论文发表，例如 [54]，[55]，[56]，[57]，[58]。但是，现有的工作较少的测试隐藏指令。

在测试处理器中的隐藏指令的时候，研究人员关注的是威胁处理器安全的未定义的指令行为，所研究方法需重点搜索和测试预留指令，应尽量跳过合法指令和无效指令。因为隐藏指令的判定规则是指令编码没有在指令集规范中定义，但是指令可被处理器执行且不报非法指令异常（具备某种功能）；一般而言，处理器在执行预留指令的时候会报非法指令异常。

现有的指令的测试方法主要有三种：（1）基于遍历指令生成的测试方法；（2）基于随机指令生成的测试方法；（3）基于深度优先遍历的方法。在测试 x86 指令集相关应用的工作中，一般使用随机和深度优先遍历的测试方法；在测试 RISC 指令集相关应用的工作中，一般使用基于遍历指令生成的测试方法。接下来我们对此三种方法进行详细的阐述，并总结其优缺点。

2.4.1 基于遍历指令生成的方法

全指令空间遍历方法的优点是覆盖率高，其不足是效率较低；它适用于对 RISC 指令空间的搜索，不适用于对 x86 指令空间的搜索。对于 RISC 指令空间的搜索，采用全空间遍历的方法所需要测试的指令的条数为 4.29×10^9 （2^{4*8}），假定 RISC 处理器每秒钟测试 1 万条指令（对 RISC 处理器已算较高，可参考 [17]），则完成测试需要 119.3 个小时，这个时间量级在可接受的范围内。对于 x86 处理器指令空间的搜索，采用全空间遍历的方法所需要测试的指令条数为 1.33×10^{36} （2^{15*8}），假定 x86 处理器每秒钟测试 10 万条指令（对 x86 处理器已算较高），在现有的 x86 处理器上完成测试需要 4.21×10^{23} 年，很显然这种时间量级是不可接受的。即使我们使用并行测试（Parallel Testing）的技术，将 x86 的指令空间平均分成 1 亿份，同时在 1 亿台处理器上展开测试，在现有的 x86 处理器上使用遍历指令生成的方法仍然需要 4.21×10^{15} 年，显然这种时间量级也是不可接受的。

现有的针对 RISC 指令集的测试方法采用基于遍历指令生成的较多，但为了

提升测试效率，每种方法会根据所需要测试的指令类型的差异，对指令空间中指令的取舍有所不同。iScanU[26] 是较早测试 RISC 中隐藏指令的方法，但是它并没有过滤掉冗余的合法指令，这使得它的测试效率不高。作者分别在 ARM 和 RISC-V 的处理器上评估了 iScanU，并在所评估的 RISC-V 处理器中发现了隐藏指令。Yuze 等 [5] 提出了一种自动化测试 RISC 中隐藏指令的方法，方法使用反汇编器筛选出预留指令，并对预留指令进行测试，分析其执行结果，判定是否为隐藏指令。同时，对于特定的处理器，当反汇编器不支持其指令集的时候，他们提出了一种基于指令编码表的筛选预留指令的方法。相比于 iScanU，他们的方法提升了对隐藏指令的测试效率。Shisong 等 [17] 提出了一种发现不同 ARM 指令集相关应用（处理器、CPU 模拟器、指令解码器等）之间指令语义差异的测试框架 iDEV，此方法同时测试了合法指令和预留指令。在展开测试之前，他们首先精简了所需要测试的指令空间。根据 ARM 指令手册中对指令编码的定义，他们发现指令中有三个重要的字节域：操作码域（Opcode Field）、条件域 (Condition Field)、操作数域 (Operand Field)。iDEV 对操作码域采取尽可能遍历的方法，因为操作码域最能够表示指令的整体语义信息。iDEV 对条件域和操作数域进行精简，从而达到精简所测试的指令空间的目的。对条件域，iDEV 将值设置为固定值 0xE，从而避免了对条件域的遍历。对操作数域，iDEV 只测试了操作数的边界值，也就是 0x0 和 0xF（操作数域为 4 个比特）。经过对指令空间的精简，iDEV 只需要测试原指令空间的 1/128，也就是大约 3300 万条指令（2^{25}）。同时 iDEV 错过了对一些指令格式比较特殊的指令的测试，但是作者指出，这些指令都对应了处理器的特定的功能，例如低功耗管理，这些指令的功能类似于 NOP 指令，不会影响到处理器的执行状态。Muhui 等 [27] 提出了一种测试 ARM 处理器与模拟器之间指令不一致的方法 EXAMINER，此方法只针对合法指令的测试。EXAMINER 通过只测试立即数的边界值和中间任意值来精简指令空间，从而提升了测试效率。

对 x86 指令集相关测试的工作中，只有少量的工作使用遍历的方法，但也只是遍历了 x86 指令的操作码，对前缀和操作数域的覆盖率不高。例如 N-version[56]，此方法是在使用了随机测试方法的基础之上，为了提升测试的操作码覆盖率，增加了遍历操作码的方法。值得注意的是，x86 指令的操作码可以是 1 到 3 个字节，遍历所有可能的操作码需要测试 2^{3*8} 种可能的编码空间，这其中含有合法指令的操作码，也包含有预留的操作码；但是论文 [56] 并没有对此进行详细的说明，只是的简单的描述了遍历了操作码（我们推测它们只遍历了已知

的合法指令操作码库），然后将每一个生成的操作码与已知的前缀和操作数进行组合，并将组合的指令进行测试。很显然，这种方法并没有较好的覆盖指令前缀域和操作数域，而在第2.2.1小节，我们已经阐述过，x86 指令中的前缀以及操作数中的 ModR/M 和 SIB 域也能决定指令的功能。所以，有必要对前缀、操作码、ModR/M 和 SIB 域进行系统的测试，从而达到较好的 x86 指令空间的覆盖率。

表 2-1　基于遍历指令生成方法的比较

Table 2-1　Comparison of methods that based on traversal instruction generation

方法	发表时间	指令类型	变化策略	指令集	优点	不足
N-version[56]	2010/7	合法	遍历操作码	x86	提升操作码的覆盖率	前缀、操作数的覆盖率低
iScanU[26]	2020/6	合法、预留	无	RISC	易于实现	效率低
iDEV[17]	2021/7	合法、预留	精简条件和操作码域	RISC	效率提升	无
EXAMINER[27]	2022/2	合法	精简立即数	RISC	效率提升	无
Yuze 等 [5]	2022/6	预留	过滤合法	RISC	效率提升	无

如表2-1所示，我们对已有的基于遍历方法的研究工作进行了分析和总结，并按照论文发表的时间顺序进行了汇总。我们发现，基于遍历指令生成的方法较多的使用于对 RISC 指令集相关应用的测试中（处理器、CPU 模拟器、指令解码器等），而较少的使用在对 x86 指令集相关应用的测试中；而且随着时间的发展，在保证覆盖率的前提下，后续发表的测试方法都比较关注提升测试效率。在测试合法指令一致性的时候，使用遍历的方法保证覆盖率，再通过精简立即数和条件码等技术提升测试效率；在测试隐藏指令的过程中，同样使用遍历的方法保证覆盖率，再使用反汇编器或者指令编码库筛选出预留指令，只对预留指令进行测试，从而实现提高测试效率的目的。

2.4.2　基于随机指令生成的方法

基于随机指令生成的方法较多的应用在对 x86 指令集相关应用的测试中。随机方法的优点是比较容易实现，不需要研究复杂的 x86 指令的格式。但是随机方法的不足之处是其测试效率较低，因为会产生很多重复的指令类型（同样的操作码但不同的立即数和偏移量操作数），而且它不能保证对 x86 指令空间的覆盖率[56, 59, 60]。为了提升随机方法的覆盖率，EmuFuzzer[59, 60] 增加了对所测试处

理器中指令的整理和遍历，但是这种方法不具备普适性，因为整理处理器对应所有的合法指令是一件繁琐的事情。N-version[56] 同样使用了基于随机指令生成的方法生成测试例，与 EmuFuzzer 的不同之处在于，它使用了遍历操作码域的方法增加对操作码的覆盖率。但是，N-version 缺少了对指令前缀和操作数域的全面覆盖，总的来看，它对指令空间的覆盖率还是较低。为了提升随机指令生成方法的效率，Nathan 等 [57] 提出了一种基于指令结构推测的随机指令生成方法 Fleece。Fleece 使用四种标签来标记指令中的比特位：域标签（Field）、预留标签（Reserved）、无用标签（Unused）和结构标签（Structural）。Fleece 通过改变每一个指令编码中的比特值，然后观察比特位的翻转对指令反汇编结果的影响，进而判定每个指令编码中比特位对应的标签类型。如果翻转比特位改变了指令的操作码、寻址模式、寄存器大小，则此比特位是结构标签；如果翻转比特位只是改变了立即数、偏移量和寄存器的值，则此比特位是域标签；如果翻转比特位使得指令从合法指令变为了预留指令，则此比特位是预留标签；如果翻转比特位对指令的的反汇编结果毫无影响，则此比特位是无用标签。通过识别指令格式再加上打标签的方法，Fleece 识别了正在测试的指令的结构，然后引导随机的方法优先对结构标签对应的比特位进行突变，从而形成新的指令；尽量不改变域标签对应的比特位，从而减少了生成冗余指令的概率（例如操作码相同但是操作数不同的指令或者无效的指令）。Fleece 虽然能够增加对识别指令类型的有效指令域的突变概率，从而实现提升对合法指令的测试效率。但是 Fleece 在给指令编码中的比特位定标签的过程中，需要不断的对指令进行位翻转和反汇编，这也大大增加了测试所需要的时间，再者，Fleece 仍旧无法保证对前缀、操作码和操作数的覆盖率。

表 2-2　基于随机指令生成的方法比较

Table 2-2　Comparison of methods based on random instruction generation

方法	发表时间	指令类型	变化策略	指令集	优点	不足
EmuFuzzer[59]	2009/7	合法	无	x86	易于实现	覆盖率低、效率低
N-version[56]	2010/7	合法	无	x86	易于实现	覆盖率低、效率低
Fleece[57]	2018/3	合法	指令操作数识别	x86	效率提升	覆盖率低

如表2-2所示，我们将现有的使用随机指令生成方法的工作按照发表时间的顺序进行了汇总。我们发现，在使用随机的方法之初，研究人员就已经认识到其

覆盖率不足，所以论文 [56, 59] 通过增加对所测试处理器的操作码的遍历来增加指令空间的覆盖率；后来，研究人员使用指令格式识别的方法增加对指令功能影响较大的字节域的突变概率，从而提升了随机指令生成方法的测试效率 [57]。整体来看，随机指令生成方法的优点是易于实现，在未研究指令格式之前，对 x86 指令集相关应用进行测试是一种较容易上手的方法；但是其缺点就是覆盖率和效率都较低。若想提升对 x86 指令集中指令测试的覆盖率和效率，就需要借助于 x86 指令集规范或者研究指令格式并找到指令编码的规律，形成约束条件，并对随机方法进行约束，引导随机方法的指令生成行为，从而有针对性的增加“有效指令”在整体生成的指令序列中的比例；与随机的方法相比，在同样的时间内，具备约束条件的随机指令生成方法能测试更多的“有效指令”，即可实现测试的高效率和高覆盖率的效果。

2.4.3 基于深度优先遍历的方法

基于深度优先遍历（Depth-first Search，DFS）的指令空间搜索方法适合于指令长度不定长的 x86 指令集，相比于基于遍历的方法和基于随机的方法，它能较好的平衡 x86 指令空间的搜索覆盖率和效率。Sandsifter[24] 是现有的第一个自动化的搜索 x86 处理器中隐藏指令的方法。Sandsifter 使用了深度优先遍历的搜索方法，重点对指令前缀、操作码、ModR/M、SIB 等字节域进行测试，绕过了大量的无效指令，使得对预留指令的搜索效率相比于直接的遍历的方法有了极大的提升。然而，Sandsifter 仍然测试了数量巨大的含有立即数和偏移量操作数的合法指令。UISFuzz[25] 在 Sandsfiter 工作的基础之上使用了指令格式识别的方法跳过了冗余的立即数和偏移量。但是从 UISFuzz 的论文的实验结果上来分析，它错过了对一些预留指令的测试。总的来看，UISFuzz 尝试提升 Sandsifter 的测试效率，但是它降低了对预留指令的覆盖率。

表 2-3 基于深度优先遍历方法的比较

Table 2-3 Comparison of depth-first search methods

方法	发表时间	变化策略	指令空间	优点	不足
Sandsifter [24]	2017/6	指令长度	合法和预留	高效排除无效指令	冗余操作数
UISFuzz [25]	2019/10	指令长度 + 格式	合法和预留	提升 Sandsifter 效率	遗失预留指令

如表2-3所示，我们将已有的深度优先遍历方法按照发表年份进行了汇总。

值得注意的是，现有的深度优先遍历算法需要借助于合法指令的长度和指令搜索深度所在字节的值来引导对 x86 指令空间的测试深度。为了获得合法指令的长度，DFS 算法需要执行合法指令，因此 DFS 算法无法只测试预留指令。使用指令长度信息使得 DFS 算法绕过了大量的无效指令，这也大大增加了 DFS 算法对 x86 指令空间的搜索效率。但是，现有的 DFS 算法仍然测试了大量的冗余的含有立即数和偏移量操作数的合法指令。而且，现有的方法 [24, 25] 没有系统的对指令前缀进行测试，指令前缀与预留操作码和预留操作数的组合也能形成新的预留指令，这使得对 x86 指令空间的测试覆盖率较低，所以，有必要对指令前缀域进行系统的测试。再者，现有的 DFS 算法增加搜索深度只是依据实时的测试指令长度的变化，当指令长度由短变长的时候，DFS 的测试深度增加一；这种单一的控制方法会使得 DFS 算法对指令空间的搜索深度不足，不能保证对预留操作码和预留操作数的覆盖。最后，现有的方法只是使用了一种反汇编器的输出结果与处理器的输出结果进行对比，从而决定指令是否为隐藏指令，这种单一的判定机制使得方法对隐藏指令的测试精度较低。

通过第2.4.1、第2.4.2和第2.4.3小节的阐述，我们分析了现有的指令空间的搜索方法。本论文研究处理器中存在的隐藏指令，我们对现有的问题进行总结，接下来我们讨论适用于搜索处理器中隐藏指令的方法。

对 RISC 处理器而言，测试各种处理器中的隐藏指令，难点在于测试代码在各种指令集之间不兼容所带来的工程问题，而遍历 RISC 指令集空间不需要过多的研究指令格式的编码规则和复杂的指令生成算法。测试 RISC 指令集处理器中隐藏指令的方法依赖于遍历指令生成，现有的工作对遍历方法的使用和对提升效率所使用的技术已经发展的较为成熟。例如，iScanU[26] 通过直接遍历的方法测试处理器中的隐藏指令能实现较高的覆盖率；同时，研究人员 [5] 通过反汇编器或者指令编码库的方法筛选出预留指令，并只对筛选出的预留指令进行测试，就能实现较高的测试效率。所以，本论文不研究搜索 RISC 处理器中隐藏指令的测试方法。

对 x86 处理器中的隐藏指令的测试，由于 x86 的指令空间巨大，无论是基于随机的方法和现有的 DFS 算法都还未能完成对 x86 指令空间的高覆盖和高效率的测试。对 x86 处理器指令集中预留指令搜索方法的研究难点有四个：第一，如何有效的绕过冗余的合法指令空间（提升测试效率）；第二，如何尽可能的覆盖到指令前缀与预留操作码和预留寄存器操作数的组合（提升测试覆盖率）；第三，如何尽可能的覆盖所有的预留操作码和预留寄存器操作数（提升测试覆盖

率）；第四，如何降低隐藏指令搜索的误报率（提升测试精度）。Sandsifter[24] 已经解决了有效绕过无效指令空间的方法，UISFuzz[25] 尝试解决第一个难点，但是错过了对部分预留指令的测试。在现有的 DFS 方法的基础之上，本论文首先针对难点一提出了更加精确的绕过冗余的立即数和偏移量的方法，更为重要的，本论文还解决难点二、三和四。

2.5 处理器微架构脆弱性的测试方法

2018 年曝出的熔断（Meltdown）[1] 和幽灵 (Spectre)[2] 处理器安全漏洞促使人们开始关注处理器微架构的安全问题。在接下来的时间里，更多的处理器微架构相关的安全漏洞被陆续曝出，例如，熔断的变种 [61]、Zombieload[62]、Fallout[63]、RIDL[64]、Medusa[65] 等，幽灵的变种 [66–71]。这使得处理器厂商不断推出对应的防御方法，包括软件补丁的发布、处理器微码的升级和后续的处理器硬件设计中的微架构脆弱性防御策略的应用。与此同时，研究人员也在研究自动化的微架构安全漏洞的测试工具 [28–30, 65, 72–78]，以及对应的防御方法 [79–92]。其中，检测方法和防御方法是相辅相成的，我们本论文重点研究检测方法，这为防御方法提供了参考依据。

对于挖掘处理器中存在的微架构安全漏洞，相关的研究从半自动化的方法开始到现有的全自动化的方法。需要注意的是，现在还没有一种能覆盖所有微架构功能部件和所有攻击类型的自动化测试工具。现有的自动化的微架构漏洞测试方法可以根据微架构功能进行分类，例如针对分支预测器类型安全漏洞的测试方法、针对乱序执行机制类型安全漏洞的测试方法等；也可以根据攻击类型（威胁模型）进行分类，例如针对幽灵类型的安全漏洞测试方法、针对熔断类型的安全漏洞测试方法等；还有测试新的侧信道攻击类型的方法，例如测试基于竞争的侧信道、测试基于时间的侧信道等。

SPEECHMINER[28] 是自动化的发掘处理器中推测执行机制引起的侧信道攻击的方法。通过选取特定的能够触发处理器异常的指令序列，预置私密信息、初始化内存、段描述符等状态。然后，SPEECHMINER 开始进行测试，在测试的过程中，通过一个已知的隐蔽信道来推测微架构状态的变化，从而尝试推测私密信息。如果异常指令序列能够成功辅助隐蔽信道完成私密信息的窃取，则记录指令序列到报告文件。然而，SPEECHMINER 只针对幽灵类型的处理器微架构安全漏洞的测试。

Daniel 等 [65] 提出了一种自动化的分析熔断类型安全漏洞的工具 Transyn-

ther。Transynther 通过对现有的熔断及其变种的攻击代码进行突变处理，然后对代码片段进行测试，目的是发现新的熔断类型的攻击。他们使用 Transynther 分析了多款处理器，深入理解了熔断类型攻击的根本原因，并发现了新的变种 MDS（Microarchitectural Data Sampling）。新发现的基于 MDS 的攻击可以从隐藏的写合并内存操作中泄露数据。然而，Transynther 只能针对熔断类型的处理器微架构安全漏洞的测试。

Daniel 等 [29] 提出了一种自动化的发掘微架构侧信道的方法 Osiris。Osiris 首先产生三个指令序列，然后自动化的测试这三个指令序列是否形成一个基于时间的侧信道（Timing-based Side Channel）。Osiris 采用基于随机的方法生成新的指令。Osiris 在 Intel 和 AMD 的处理器中发现了 4 个基于时间的侧信道。然而，Osiris 没有展示隐藏指令放入其测试的指令库后相关的测试结果，再者，它使用的基于随机的指令序列生成方法的效率较低。

共享的微体系结构资源与推测执行技术相结合，它们能够引起像熔断和幽灵一样的侧信道攻击。Oleksii 等 [30] 推出了一种自动化的测试微架构信息泄露的方法 Revizor。Revizor 采用的是 Model-based Relational Fuzzing（MRF）的方法来测试处理器，它需要以推测执行协议 [31] 为一个“黄金模型”。推测执行协议是一种规约，它描述攻击者可以利用的侧信道以及能够改变处理器控制流和数据流的方法。在实验阶段，作者使用部署推测执行协议的 CPU 模拟器来执行待测试的指令序列，与此同时将测试例在处理器上运行。然后比较模拟器和处理器的状态差异。Revizor 能够自动化的测试与推测执行协议相悖的测试例，以找到可能的新的攻击形式。作者用 Revizor 测试了 Skylake 和 Coffee Lake 微架构的处理器，发现了 Spectre 和 MDS 等微架构安全漏洞。

如表2-4所示，我们将现有的自动化的测试处理器微架构安全漏洞的方法以时间顺序进行罗列。从表中可知，现有的方法大多是针对挖掘熔断和幽灵相关的侧信道攻击，对于基于时间的和基于争用的侧信道较少。而且，他们的挖掘方法的指令生成效率还有需要进一步提升的地方。更为重要的是，它们都没有分析隐藏指令可能带来的侧信道攻击。

本论文我们会结合检测出来的隐藏指令，探索其与合法指令共同组成的指令库所能引发的基于时间的侧信道。基于时间的侧信道其根本原因是利用了处理器微架构中的共享部件，因为指令的执行会对共享部件产生占用，当再次执行需要同样计算资源的指令的时候，就会出现等待，具体表现为指令执行时间增加。共享部件经常会被攻击者利用发起侧信道攻击，可被利用的共享部件包括

表 2-4　不同的微架构安全漏洞测试方法的比较

Table 2-4　Comparison of different microarchitectural security vulnerability testing methods

方法	发表时间	漏洞类型	新指令	指令序列	微架构机制
SPEECHMINER [28]	2019/12	幽灵，熔断	无	特定模板	乱序执行、分支预测
ABSynthe [19]	2020/2	争用侧信道	遍历指令库	遍历指令组合	无特定机制
Transynther [65]	2020/7	熔断	随机突变	随机突变	乱序执行
Revizor [30]	2021/5	幽灵	随机	随机	分支预测
Osiris [29]	2021/8	时间侧信道	随机	随机指令组合	无特定机制

Cache（L1、L2、LLC），缓存（TLB 和 LFB 等）以及执行端口（Execution Ports）等。在传统的攻击场景中，为了利用共享部件发起侧信道攻击，开发者需要精心的设计发起攻击的指令序列。在使用这些指令序列之前，开发者需要熟悉的掌握处理器底层微架构相关的知识。然而，微架构组件的底层细节对程序开发人员不可见，因此需要逆向工程分析。而且，处理器微架构也在不断的发展变化，不同的处理器平台之间的微架构也有差异，这使得研究侧信道攻击成为一个复杂的工作，彰显出自动化的测试和分析侧信道攻击相关的处理器微架构脆弱性的方法的重要性。因为底层微架构的功能都会表现在指令层，而且，攻击者发起侧信道攻击也是依赖于对指令的使用。所以，我们从指令层面研究自动化的测试可能存在的基于时间的侧信道攻击方式，通过对合法指令和隐藏指令进行分组和测试，尝试发现新的基于时间的侧信道攻击。

2.6　本章小结

本章节我们首先介绍了指令集系统结构和处理器微架构相关的基础知识。然后我们分析了现有的指令空间搜索的方法和微架构安全漏洞的测试方法，并对现有的方法进行了比较和讨论。对 x86 处理器中的隐藏指令的测试，我们分析了现有的方法测试效率不高、覆盖率不高、精度不足等问题。对 x86 处理器中微架构脆弱性的测试，我们分析了现有的测试方法，并计划将测试出来的隐藏指令与合法指令共同形成一个指令库，然后测试指令库中能够触发基于时间的侧信道的指令组合。

第 3 章　基于合法指令中操作数的最小测试集的方法

在测试 x86 处理器中的隐藏指令的过程中，冗余的合法指令和无效指令如果在所测试的总指令中的占比过高会使得测试的效率较低。理想情况下，x86 指令空间由合法指令、预留指令和无效指令组成。隐藏指令是具备某种功能的预留指令。减少对合法指令和无效指令的测试，能够提升隐藏指令的测试效率。测试效率能最直观的体现在测试的时间上，高效率的测试方法能减少开发者测试处理器的等待时间，从而减少分析一款处理器的工作周期。现有的方法 Sandsifter[24] 使用基于深度优先遍历的方法绕过了大量的无效指令，而且有较好的效果。但是 Sandsifter 仍然测试了相当数量的冗余的合法指令，这些冗余的合法指令都有立即数和偏移量操作数。而经过研究我们发现，立即数和偏移量不影响指令功能，也不影响对预留指令的测试，所以需要尽可能的减少冗余的合法指令在所测试的指令中的比例，从而提升整体的测试效率，减少测试所需要的时间。

本章节，我们面对在测试 x86 处理器中隐藏指令的过程中测试了大量的冗余的合法指令的问题展开研究，目的是为了提升隐藏指令的测试效率。我们首先展示了提升隐藏指令测试效率的动机。然后，我们提出了合法指令中立即数和偏移量操作数的最小测试集（简称合法指令中操作数的最小测试集）的方法，方法减少了冗余的合法指令在所测试的总的指令数量中的占比，同时，我们对方法具体的实现原理和对应的伪代码进行详细的阐述。最后，我们将合法指令中操作数的最小测试集的方法在 8 种来自 Intel 和 AMD 的处理器上进行评估测试，并将结果与现有的方法进行对比分析，从而证实所提出方法提升测试效率的有效性。

3.1　研究动机

对预留指令空间的搜索效率指的是预留指令空间占总的测试指令的比例，这个比例越高，则对隐藏指令的搜索效率就越高。理想情况下，我们若能只测试 x86 指令集中的预留指令，那么就能实现对 x86 处理器中的隐藏指令的最高测试效率。然而，x86 指令空间中的合法指令空间、预留指令空间和无效指令空间是相互交错的，无法完全只测试预留指令。现有的深度优先遍历算法（Depth First Search，DFS）能够绕过大量的无效指令，而且已能达到较好的效果。但是现有 DFS 算法仍然测试了大量的冗余的含有立即数和偏移量操作数的合法指令。所以，我们尽量的减少对冗余的合法指令的测试就能实现预留指令在所测试的总

的指令中的占比最大化，提升搜索隐藏指令的效率。

公式3-1表示的是预留指令的占比（Proportion of Reserved Instructions in Total Instructions，PRITI），其中 NLI 表示合法指令的数量（Number of Legal Instructions），NRI 表示预留指令的数量（Number of Reserved Instructions）。为了使得 PRI 最大化，同时，我们需要保证对 NRI 的覆盖率，我们只能通过降低冗余的合法指令数量 NLI。

$$PRITI = \frac{NRI}{NRI + NLI} \times 100\% \tag{3-1}$$

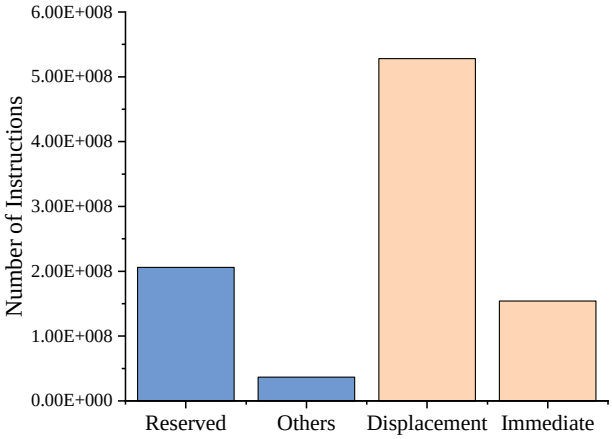

图 3-1　Sandsifter 测试的指令类型的分布

Figure 3-1　The instruction type distribution of Sandsifter

注：图中 Reserved 表示预留指令空间，Others 表示没有立即数和偏移量的指令类型，Displacement 表示有偏移量的指令类型，Immediate 表示有立即数的指令类型。

Sandsifter 的测试效率不高的主要原因是因为它测试了大量的含有立即数和偏移量操作数的合法指令。如图3-1所示，我们对 Sandsifter[93] 测试的指令进行分类，发现立即数和偏移量类型的合法指令数量达到了 78.14%，全部测试这些立即数和偏移量是没有必要的。因为立即数和偏移量操作数并不影响指令的功能。为了提升预留指令空间在所测试的所有的指令中的占比，我们需要减少对冗余的合法指令的测试。

对预留指令的测试效率能直接的表现在测试的时间上，当指令前缀长度增加的时候，所需要测试的指令数量也会成倍的增加。我们使用 Sandsifter 在处理器 i7-6700 上展开测试，在指令前缀长度为 1 个字节的时候完成测试需要 8 个小时，这个时间是可接受的。然而，当指令前缀长度为 4 个字节的时候，即使我们使用基于组合优化的前缀生成方法（第4章详细描述，目的是为了提升 x86 指

令空间指令前缀域的覆盖率和效率），预计所需要的时间也会超过 1266 个小时（$8 \times (4273 \div 27)$，可参考表4-2），这大大增加了我们测试的等待时间。

综上所述，为了提升 x86 处理器中隐藏指令的测试效率，提升预留指令在所测试总的指令中的占比，我们需要探索筛除冗余的合法指令的方法，实现方法的思路是减少含有立即数和偏移量操作数的合法指令在所测试总的指令中的占比。

3.2　控制 DFS 算法的搜索深度

在介绍合法指令中操作数的最小测试集之前，我们首先要介绍深度优先遍历算法对 x86 指令空间的搜索机制，因为我们的方法也需要基于其搜索机制。如表3-1所示，DFS 算法控制搜索深度主要依靠两种机制：第一，观察指令长度的变化，当指令长度从短变长的时候，指令空间的搜索深度增加 1；第二，观察搜索深度所在字节的值，我们命名为**深度字节值**（Depth-Byte），当深度字节值为 0xFF 的时候，指令空间的搜索深度减少 1。

接下来，我们首先以一个加法指令为例子阐述指令树的结构，目的是为了说明深度字节值的概念。然后，我们详细描述控制 DFS 算法搜索深度的机制 1 和机制 2，机制 3 是我们合法指令操作数的最小测试集的具体实施方法，我们将在第3.3小节详细阐述。

表 3-1　控制 Skipscan 中 DFS 算法搜索深度的机制

Table 3-1　The mechanisms that leverage by Skipscan to control the search depth

机制	使用信息	触发条件	搜索深度变化	效果	使用
机制 1	指令长度	长度变化	深度增加 1	增加测试深度	Skipscan、Sandsifter[24]
机制 2	深度字节值	0xFF	深度减少 1	减少测试深度	Skipscan、Sandsifter[24]
机制 3	深度字节值	0x01、0x02	深度不变	增加测试的效率	Skipscan （见3.3小节）

3.2.1　增加 DFS 算法搜索深度的机制

增加指令空间的搜索深度是为了保证对指令操作码和其对应操作数的覆盖。深度优先遍历算法对 x86 指令空间进行遍历的时候，使用指令长度信息来增加 DFS 算法的搜索深度。在测试的初始阶段，指令的搜索深度为 1，指令长度的初始值为 0。深度优先遍历算法测试的第一条指令是 15 个字节的 0x00，我们表示为 {15-Byte 0x00}。当把指令序列 {15-Byte 0x00} 放入内存的时候，处理器尝试

图 3-2　Skipscan 与 Sandsifter 指令流片段的对比

Figure 3-2　Comparison of instruction streams segments of Skipscan and Sandsifter

对指令进行取指、译码和执行。注意，在测试指令的时候处理器被设置为单步执行模式，即一次只能执行一条指令。当指令序列 {15-Byte 0x00} 的前两个字节被取到之后，也就是指令 00 00，则指令被处理器识别为一个加法指令，其指令长度被确定为 2 个字节，则剩下的 13 个字节的 0x00 会被抛弃。在指令 00 00 被执行之后，指令长度从 0 变为 2，机制 1 检测到指令长度由短变长，则指令的搜索深度在初始值 1 的基础之上加 1 变为了 2，从图3-2中 Skipscan 指令流的前两行可以看出来。然后，紧随其后的几条指令的操作码都是 0x00，所以它们都是加法指令类型，只是其操作数的种类有区别，所以指令的长度也会有差异。从图3-2中 Skipscan 指令流中可以看出，当指令长度由 2 变为 3 的时候，以及由 3 变为 7 的时候，DFS 算法的搜索深度都会加 1。机制 1 通过实时检测指令长度的变化及时的增加搜索深度，从而保证了对操作码和操作数的覆盖。

机制 1 是根据指令长度的变化增加测试深度，而得到指令长度需要借助于内存中的两个具备不同执行权限的页来实现。操作系统决定一个物理页是否具备可执行权限是读取 PTE（Page Table Entry）的第 63 比特的值，也就是 NX（No-execute Bit）。当 NX 比特的值为 0 的时候，则此页具备可执行权限，简称可执行页；当 NX 比特的值是 1 的时候，则此页不具备可执行权限，简称不可执行页。当处理器从不可能执行页中取指令的时候，会触发页错误异常（Page Fault

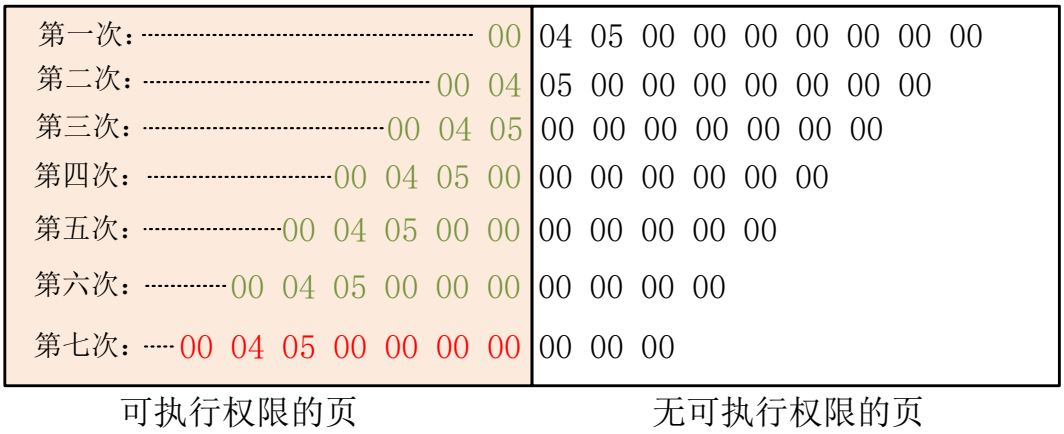

图 3-3　使用不同执行权限的内存页确定指令长度

Figure 3-3　Determine instruction length by using two memory pages with different execution permissions

Exception），操作系统会得到一个 SIGSEGV 类型的信号。

我们以指令 00 04 05 00 00 00 00 为例来说明指令长度的获得过程。如图3-3所示，15 个字节长度的指令被“横跨”着放入可执行页和不可执行页——指令的第一个字节被放入可执行页，剩下的指令字节被顺序的放入不可执行页。处理器被引导从可执行页的特定地址处开始取指令，第一个被取到的字节是 0x00；由于处理器被设置为单步执行模式，取到的指令 0x00 会被处理器解码和执行。然而处理器无法对指令 0x00 解码，则会触发非法指令异常（操作系统中对应信号 SIGILL）。操作系统的异常处理机制会处理此异常，然后处理器会尝试取第二个字节，但是第二个字节被放置在不可执行页中，所以取第二个字节的行为会触发缺页异常（Page Fault，PF）。操作系统的异常处理机制处理完缺页异常之后，Skipscan 会将指令的第二个字节往可执行页中移动，然后处理器进入下一个阶段的取指、译码和执行环节。当处理器取到的指令可被处理器解码和执行且不报缺页异常的时候说明已不再需要从不可执行页中取新的字节，如图3-3中的第七次对指令的取指、译码和执行，这个时候可执行页中的字节序列就是一个可执行的指令，其指令长度可以通过可执行页的地址差计算得到。通过处理器和内存中的不同执行权限页的配合，经过多次尝试，就能得到指令的真实长度。

需要注意的是，图3-3中的计算指令长度的机制不仅可以得到合法指令的长度，同时还能得到隐藏指令的长度，这使得搜索 x86 的预留指令空间变得更加的全面。虽然反汇编器可以解码得到指令的长度，但是其指令长度的可信度不如处理器硬件得到的高，而且使用反汇编器不能得到隐藏指令的长度。

3.2.2 减少 DFS 算法搜索深度的机制

为了容易理解深度优先遍历算法在搜索 x86 指令空间时候的深度控制机制 2，我们首先讲述搜索深度和深度字节值的概念。我们将一个加法指令以指令树的形式表示，图3-4为加法（add）指令 00 04 05 00 {00-FF} 00 00 00 的指令树，其中操作码（Opcode）为第 1 层；ModR/M 字节为第二层；SIB 字节为第三层；偏移量（Displacement）为第四层到第七层（偏移量为 4 个字节，我们图中只展示了两个字节）。图中所示指令树的测试深度为 4，所以其深度字节为指令的第四个字节，图中展示的深度字节值为 {0x00-0xFF}。当**深度字节值**为 0xFF 的时候，DFS 算法的测试深度减 1（表3-1中机制 2），指令 add 测试完毕，进入对下一条指令 00 04 06 {12 个字节的 00} 的测试。

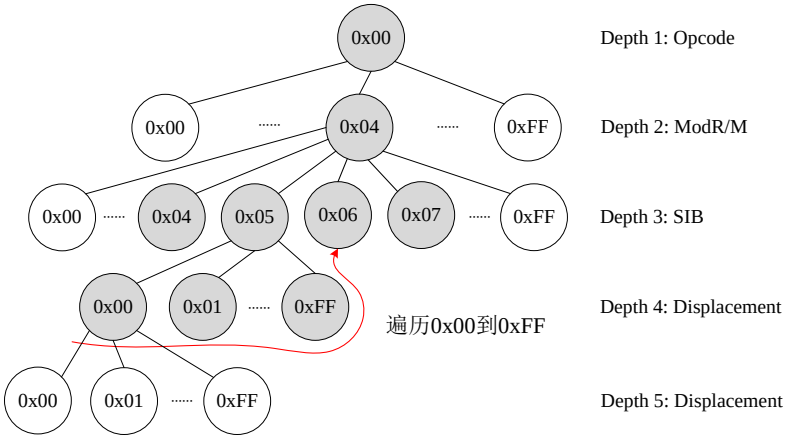

图 3-4　指令 00 04 05 00 00 00 00 的指令树

Figure 3-4　The instruction tree of 00 04 05 00 00 00 00

注：此指令树同时还表明 Sandsifter 的方法遍历 add 指令的时候测试了 256 个偏移量，与图3-5做对比。

机制 2 通过实时的检测深度字节值来判断是否需要及时的减少测试深度，从而有效的绕过无效指令，同时能够一定程度上绕过冗余的立即数和偏移量。对于搜索深度等于指令长度的指令，其指令树中更深一层的字节是不需要遍历的，因为指令长度之外的字节都是无效的字节。所以，当搜索深度值为 0xFF 的时候就需要及时的减小测试深度，从而绕过无效指令的字节。对于搜索深度小于指令长度的指令，有两种情况：第一，搜索深度所在的字节（简称搜索深度字节）是立即数或者偏移量操作数；第二，搜索深度字节是操作码或者寄存器操作数，这种情况是指令空间的搜索深度不足，我们将在第5章节讨论。对于第一种情况，搜索深度字节后面紧随的字节依然是立即数或者偏移量，所以 DFS 算法没有必要

再继续遍历更深一层的字节。如图3-2中 Skipscan 的指令流，当测试到指令 00 04 05 FF 00 00 00 的时候，其深度字节值为 0xFF，它后面紧随着的三个 0x00 就没有必要遍历了，因为他们仍然是偏移量操作数。(x86 是小端字节序 (Little-Endian)，指令 00 04 05 FF 00 00 00 中的偏移量为 0x000000FF。) 机制 2 会及时的将 DFS 的测试深度减小 1，则下一条要测试的指令就是 00 04 06。

但是，机制 2 也存在其弊端：第一，仍然要遍历 256 个立即数或者偏移量的值，我们提出了合法指令中立即数和偏移量的最小测试集的方法近一步提升了机制 2 的效率（详见第3.3节）；第二，只通过识别搜索深度值这一个变量，会使得对指令空间的搜索深度不足，尤其是会错过对一些操作码和寄存器操作数的搜索，这个问题将在第5章进行详细的讨论。

3.3　合法指令中操作数的最小测试集

现有的 DFS 算法使用机制 2 能够绕过大量的无效指令空间和部分冗余的立即数和偏移量操作数，但是它使得 DFS 算法仍然搜索了大量的立即数和偏移量操作数。本节我们主要阐述合法指令中立即数和偏移量操作数的最小测试集。

我们本章节的主要目的是为了提升对 x86 指令空间中预留指令的测试效率，也就是为了提升预留指令在所测试的总的指令中的占比。x86 指令空间中有预留指令、合法指令和无效指令，有效的减少合法指令和无效指令的占比就能够达到提升预留指令占比的效果。跳过无效指令可以通过及时的减小测试深度实现，机制 2 已经能够较好的达到绕过无效指令空间的效果。减少合法指令的占比，主要是通过减少冗余的含有立即数和偏移量的合法指令，从而实现不减少测试的合法指令的种类的同时减少合法指令的占比的效果。

测试合法指令中立即数和偏移量操作数的时候有两种方法能够实现对其最少的测试：第一，随机选取几个值进行测试；第二，采用只测试其边界值的方法。这两种方法在不同的测试工作中都有不同程度的应用。我们采用边界值的思想，因为边界值分析法 [94] 指出大量的错误是发生在输入或输出范围的边界上，而不是发生在输入输出范围的内部。

在结合 DFS 算法实现最小测试集的算法的过程中，为了同时兼顾测试的可实现性和效率，我们使用特征值作为 Skipscan 识别立即数和偏移量操作数的依据。在对 x86 指令空间进行深度优先遍历的时候，我们首先需要识别出指令中有立即数或者偏移量，然后才能跳过它们中的冗余的操作数。我们使用反汇编器 Capstone[95] 识别指令中是否有立即数和偏移量，然而，当使用 Capstone 反汇编

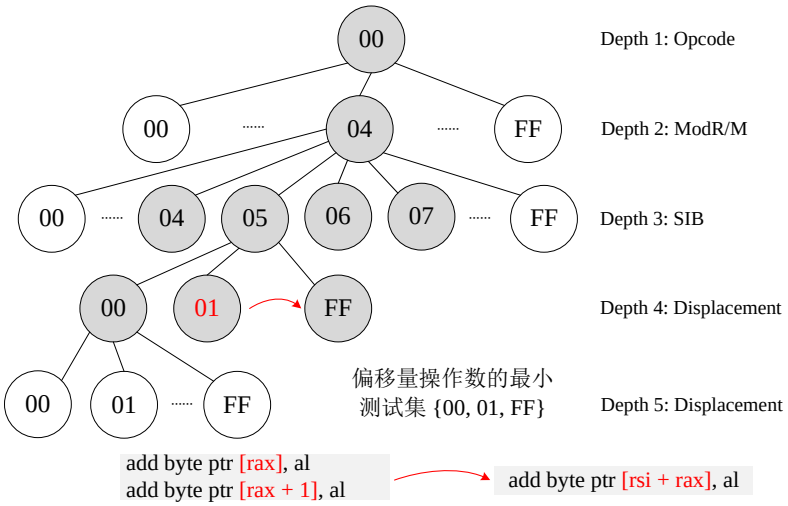

图 3-5　Skipscan 中遍历 add 指令的时候测试了 3 个偏移量

Figure 3-5　The traversal method of Skipscan for the add instruction, which tests 3 displacement value

一个指令序列的时候，它只能返回具体的立即数和偏移量值，且无论指令中是否真的存在立即数和偏移量，Capstone 默认的立即数和偏移量值都为 0x00。所以，为了精确的识别出含有立即数和偏移量操作数的指令，我们需要一个特征值，也就是一个 0x00 以外的值，例如 0x01、0x02、0x03 等等。理论上，在 0x01 到 0xFF 之间的值都可以被用来当做特征值，但是，为了兼顾测试的效率，我们应选取最小的值。当 Skipscan 识别到立即数或者偏移量的特征值的时候，会直接跳转到对 0xFF 的测试，然后机制 2 会及时的减少测试深度。这样，就能减少对冗余的立即数和偏移量的测试。

在测试之初，我们对立即数和偏移量统一使用 0x01 作为特征值，但是发现会丢失一些预留指令。经过分析测试的指令流，我们发现，有一些合法指令中使用了固定的立即数值 0x01，这就导致识别 0x01 后直接跳转到对 0xFF 的测试会丢失一部分合法指令，同时也错过了对合法指令空间中夹杂着的预留指令的测试。所以，我们将立即数的特征值变为了 0x02，经过测试，我们发现测试的预留指令数量与 Sandsifter 保持一致。所以就实现了提升测试效率的同时，保证对预留指令的测试数量不丢失的目的。如表3-1的机制 3 所示，立即数的特征值为 0x01，偏移量操作数的特征值为 0x02。

综上所述，对含有偏移量的合法指令，其操作数的最小测试集为 {0x00, 0x01, 0xFF}，例如图3-5中所示；对于含有立即数的合法指令，其操作数的最小测试集为 {0x00, 0x01, 0x02, 0xFF}。最小测试集中的 0x00 元素是指令中立即数和偏移

量的默认值，在执行了 0x00 所在的指令之后，指令长度的变化就会被检测出来，从而就能确定新的搜索深度。特征值 0x01 和 0x02 是为了区别偏移量和立即数类型，从而实现对冗余的偏移量和立即数操作数的跳过处理。最小测试集中的 0xFF 有两方面的作用：第一，作为字节的边界值来被测试；第二，被用来检查直接跳转到 0xFF 后指令的类型是否改变。例如，通过比较 0x01 和 0xFF 所在的指令的解码情况，比较指令语义是否改变，如果没有改变，则证明我们的方法没有跳过合法指令；如果改变了，则将指令记录下来，并需要对此指令进行重新测试和分析。

3.4　算法的设计

使用深度优先遍历的方法对 x86 的指令空间进行搜索，其指令空间的遍历范围是 {15-Byte 0x00} 到 {15-Byte 0xFF}。深度优先遍历与全遍历最大的区别在于，通过观测指令长度的变化（机制 1）和深度字节值（机制 2）来控制 DFS 算法的搜索深度，从而绕过了大量的无效指令空间以及部分的冗余的合法指令。我们提出的合法指令操作数的最小测试集的方法最大限度的降低了合法指令在所测试总的指令中的占比，从而提升了对预留指令的测试效率。算法1展示了指令的测试过程以及对合法指令中操作数的最小测试集的方法的具体实现。我们使用了表3-1中的机制 3 来减少对合法指令中的冗余的立即数和偏移量的测试。算法中第 17 到 20 行展示了对立即数操作数的具体实现，通过实时的检测指令中是否存在立即数以及立即数的特征值 0x02，当这两个条件同时满足的时候，就触发机制 3 将立即数的值直接跳转到 0xFF，然后对 0xFF 进行测试。同样的，对偏移量操作数，算法中 21 行到 24 行展示了具体的实现。当指令中含有偏移量且特征值 0x01 出现的时候，触发机制 3 将偏移量的值跳转到 0xFF，然后对 0xFF 进行测试。

算法 1　合法指令操作数的最小测试集

Require: $1 < m \leq 4$　　▷ 指令前缀数量

```
1: G ← Instruction_Generation()
2: D ← Disassembler()
3: C ← CPU_Execution()
4: Insn ← G.Instruction                    ▷ Insn 表示 Instruction
5: Imme ← Have_Immediate(Insn)
6: Disp ← Have_Displacement(Insn)
7: Length_Changed ← New_Length - Old_Length
8: Depth_Byte ← Depth_Byte(Insn)
9: for instruction ≤ {15-Byte 0xFF} do
```

```
10:   Test_the_instruction()
       Get_the_instruction_length()
11:   if Length_Changed then
12:       Search_Depth = + 1                          ▷ 增加测试深度，机制 1
13:   end if
14:   if Depth_Byte = 0xFF then
15:       Search_Depth = - 1                          ▷ 减少测试深度，机制 2
16:   end if
17:   if Imme = 0x02 then
18:       Skip_to_0xFF                                ▷ 提升测试效率，机制 3
19:       Return_to_Test_the_instruction()
20:   end if
21:   if Disp = 0x01 then
22:       Skip_to_0xFF                                ▷ 提升测试效率，机制 3
23:       Return_to_Test_the_instruction()
24:   end if
25:   if D = 0 and C = 1 then      ▷ 指令不能被反汇编器解码（D），但是能够被处理器执行（C）
26:       Put_the_instruction_to_logfile              ▷ 将测到的隐藏指令放入 log 文件
27:   end if
```

另外，值得注意的是，我们的算法1中对机制 3 的实现需要依赖于机制 1 和机制 2，机制 1 的具体实现在第 11 到 13 行，机制 2 的具体实现在第 14 到 16 行。当被测试的预留指令具备某种功能的时候，它就会被识别为隐藏指令，然后被输出到 log 文件中，如算法1中第 25 到 27 行所示。

3.5 测试框架

对合法指令中操作数的最小测试集方法的应用需要结合指令测试的差分测试框架。如图3-6所示是基于差分测试的指令测试框架，其基本思想是把生成的指令分别送入处理器和反汇编器，然后比较处理器和反汇编器的输出，如果其输出结果与预期的不一致，则将指令放入错误报告文件中，最后对错误报告中的指令进行重新检查。接下来，我们对测试框架各个模块的功能进行描述。

- 初始化：在对指令进行测试之前，需要对处理器的寄存器状态进行初始化、预置指令测试的内存地址、确保异常处理机制处于工作状态。
- 指令生成：本论文使用的是基于深度优先遍历的指令生成方法，深度优先遍历的方法对操作码和寄存器操作数有较好的覆盖率。注意，生成的待测试的指令是一个长度为 15 个字节的序列，然后将指令序列放入预置的内存地址，引导处理器进行取指令的操作。
- 处理器：处理器作为被测试的对象，它对生成的指令进行取指、译码和执行。

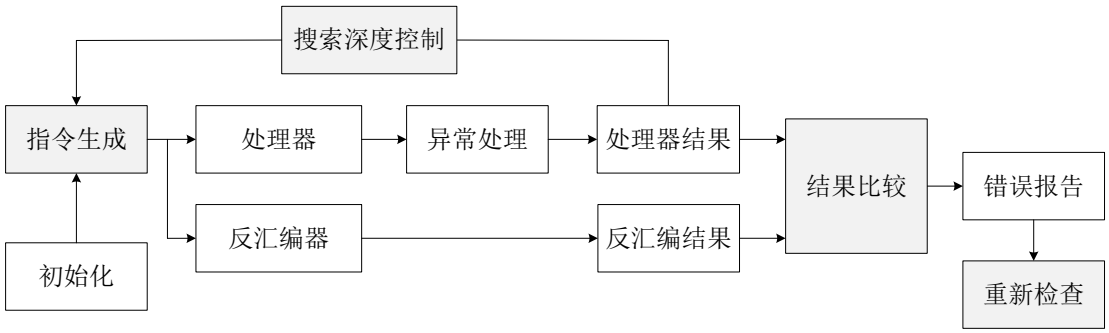

图 3-6　合法指令中操作数最小测试集的测试框架

Figure 3-6　The test framework for the minimum operand test set of legal instructions

处理器从预置的内存地址取指令，由于处理器被设置为单步执行模式，所以每次都只执行一条指令。处理器通过逐个字节的取指令操作，判断有效的指令长度，然后执行指令。

- 反汇编器：反汇编器在测试框架中作为指令集架构规范使用。反汇编器直接从生成指令的缓存中得到指令，然后对指令进行解码，目的是为了判断指令是否具备语义，同时提供指令的长度信息。
- 异常处理：如果指令在执行的过程中报出异常，则需要将程序的执行状态设置到一个已知的状态。注意，处理器的异常在操作系统层面有其对应的异常信号，异常处理模块获得异常信号，判断异常类型，然后针对不同的异常信号有不同的处理机制。异常处理模块调用操作系统中的 sigaction 函数，预置处理器的异常上下文信息到一个初始化的状态，然后引导处理器进入对下一条指令的测试周期。
- 处理器结果：处理器的结果有指令的长度信息和指令执行的时候的异常信息，在 Linux 操作系统中每一种处理器异常都会有其对应的信号量。
- 反汇编结果：反汇编结果为指令的长度信息，指令的语义（主要是获得指令是否可被反汇编的结果，从而判断其是否在指令集规范中定义）。
- 搜索深度控制：根据指令长度的变化和搜索深度所在的指令字节的值来控制深度优先搜索方法的搜索深度。一般情况下，当指令长度从短变长的时候，指令空间的搜索深度会增加 1；当测试深度所在的字节的值变为 0xFF 的时候，指令空间的搜索深度会减少 1。根据指令长度变化及时的增加搜索深度能保证对操作码和寄存器操作数的覆盖；根据搜索深度所在字节的值的变化及时的减少搜索深度能绕过对无效指令的遍历。
- 结果比较：通过比较处理器和反汇编器的输出结果，我们可以判断出指令的类型。一般而言，判断指令是隐藏指令需要两个条件：第一，指令可以被处理器正

常执行且不报非法指令异常，也就是在 Linux 操作系统中没有报 SIGILL 的异常信号；第二，指令不能被反汇编器反汇编，也就是反汇编器对所测试的指令的反汇编结果输出为“Unknown”。

- 错误报告：将隐藏指令放入错误报告文件中。
- 重新检查：错误报告中的指令有可能是误报的，所以，需要使用不同的反汇编器对错误报告文件中的指令进行重新的反汇编，从而检查指令是否真的为隐藏指令。因为反汇编器中有可能存在缺陷，或者反汇编器没有及时的更新指令集的信息，所以导致指令没有被反汇编器正常的解码。很显然，使用多种不同的反汇编器能够降低对隐藏指令的测试误报率，我们将在第6章详细讨论。

我们对隐藏指令的重新检查以及隐藏指令可能存在的功能等都放到第6章节讨论。

3.6 实验评估

我们对提出的合法指令中操作数的最小测试集的方法在 8 种处理器上进行评估，如表3-2中所示为所选处理器平台，表中 x86 处理器是近几年来的主流的 Intel 和 AMD 的处理器。我们所使用的反汇编工具版本是 Capstone-4.0.2；使用的操作系统类型是 Ubuntu 16.04，64-bit 版本，内核版本为 4.15.0。

表 3-2 评估合法指令操作数的最小测试集所用的处理器

Table 3-2 The processors used to evaluate the minimum test set operands of legal instructions

处理器	微架构	处理器系列	发布时间
i5-11600KF	Rocket Lake	Core	March 16, 2021
i7-10700KF	Comet Lake	Core	April 30, 2020
Gold 5218	Cascade Lake	Xeon	April 2, 2019
i5-7300HQ	Kaby Lake	Core	January 3, 2017
i7-6700	Skylake	Core	September 27, 2015
Zen3-5600X	Zen3	Ryzen	November 5, 2020
Zen2-3700X	Zen2	Ryzen	July 7, 2019
Zen-1800X	Zen	Ryzen	March 2, 2017

本小节评估和展示跳过 x86 指令空间中冗余的合法指令的方法。我们将算法1实现在图3-6所示的差分测试框架中。通过对指令空间的深度优先遍历，然后应用机制 1、2 和 3 共同控制指令空间的搜索深度，我们的方法达到了绕过大量无效指令的同时尽可能少的测试合法指令的效果，实现了提升预留指令空间在所测试的总的指令中的占比的目的。我们接下来将从预留指令和合法指令在所测试的指令中的占比的变化来评估我们方法的效果；同时，效率的提升直观的表现在测试时间的减少，我们将 Skipscan 完成同样的指令类型的测试所花费的时间与 Sandsifter 的测试时间进行对比，结果表明 Skipscan 使用较少的时间就能达到与 Sandsifter 一样的预留指令的覆盖率。

3.6.1　度量指标

本小节使用的度量指标有三个：第一，预留指令占总测试指令的占比；第二，合法指令占总测试指令的占比；第三，完成测试所花的时间。

预留指令占比（Proportion of Reserved Instructions in Total Instructions，PRITI）的计算表达式：

$$PRITI = \frac{Number\_of\_Reserved\_Instructions}{Number\_of\_Total\_Instructions} \times 100\% \tag{3-1}$$

我们所测试的指令只包括合法指令和预留指令，所以，合法指令占比和预留指令占比的值的和为 1，合法指令占比（Proportion of Legal Instructions in Total Instructions, PLITI）的计算表达式可以表示为：

$$PLITI = 1 - PRITI \tag{3-2}$$

3.6.2　实验结果

如表3-3所示，为 Sandsifter 和 Skipscan 所测试的总的指令数量、预留指令数量、隐藏指令数量、预留指令占比等数据。首先值得注意的是，Skipscan 的预留指令占比（PRI）的平均值为 79.78%，而 Sandsifter 的预留指令占比的平均值为 21.89%，我们的方法将占比提高了 3.64 倍。而且，Skipscan 能保证测试的预留指令数量与 Sandsifter 保持一致，同时得到的隐藏指令数量也能保持一致。

注意，不同的处理器平台之间所测试的指令数量会存在差异，其中有两个原因：第一，不同的处理器平台实现的 x86 指令数量会有细微的差别；第二，由于不同的处理器中合法指令的运行行为存在一定的差异，所以需要将能够导致程序退出的指令放入黑名单，从而保证测试程序的正常运行。

表 3-3 Skipscan 和 Sandsifter 的测试指令空间的对比

Table 3-3 Comparison of tested instruction space of Skipscan and Sandsifter

处理器	Sandsifter 总指令	Skipscan 总指令	预留指令	隐藏指令	Sandsifter PRITI	Skipscan PRITI
i5-11600KF	878,615,246	249,578,427	204,100,708	3,351,847	23.22%	81.79%
i7-10700KF	878,615,246	249,578,427	204,100,708	3,347,748	23.22%	81.79%
Gold 5218	952,728,446	241,086,203	196,228,010	3,045,384	20.60%	81.39%
i5-7300HQ	936,606,326	242,102,430	197,814,409	3,961,248	21.12%	81.71%
i7-6700	878,615,246	237,335,796	193,556,383	3,347,748	22.03%	81.55%
Zen3-5600X	890,364,631	254,324,432	195,040,790	3,346,148	21.91%	76.70%
Zen2-3700X	925,493,431	254,324,432	195,040,790	3,346,148	21.07%	76.70%
Zen-1800X	940,180,411	263,405,773	205,996,896	3,959,684	21.91%	78.21%
平均值	910,152,373	217,176,436	198,984,837	3,463,244	21.89%	79.78%

如表3-4展示了将 Skipscan 和 Sandsifter 在 8 种不同的处理器上测试同样的预留指令空间所需要的时间。从表中的数据可以看出，Sandsifter 所需要的时间的平均值是 Skipscan 的 4.03 倍。这大大降低测试人员的等待时间，而且为第4章和第5章提升指令空间的覆盖率也提供了效率的支撑。

3.7 讨论

第3.6.2小节我们展示了 Skipscan 的测试结果，并从预留指令占所测试的总的指令数量的占比和测试的时间两个方面与 Sandsifter[24] 进行了对比。从结果上看，我们的方法确实能够提升测试的效率，从时间上来看，测试同样数量的预留指令，测试的效率提升了 4.03 倍。

但是与 UISFuzz[25] 对比我们发现，Skipscan 的效率要比它稍低一些。单从测试时间上来看，UISFuzz 的测试效率是 Sandsifter 的 5.57 倍，这个数值要比 Skipscan 的 4.03 要高。但是我们发现 UISFuzz 总共只测试了 131,270,176 条指令，要远远小于 Skipscan 测试的 217,176,436 条。而且，UISFuzz 只得到了 2,157,079 条隐藏指令，要小于 Skipscan 得到的 3,463,244 条。很显然，UISFuzz 测试了更少的指令数量，所以其效率会高一些，但是它也丢失了部分预留指令，同时，它得到的隐藏指令数量更少。从原理上来分析，UISFuzz 使用的是操作数识别的方

表 3-4　Skipscan 和 Sandsifter 测试时间的对比

Table 3-4　Comparison of testing time of Skipscan and Sandsifter

处理器	Sandsifter 的时间（分钟）	Skipscan 的时间（分钟）	倍数
i5-11600KF	186	48	3.88
i7-10700KF	188	48	3.92
Gold 5218	315	75	4.2
i5-7300HQ	640	145	4.41
i7-6700	360	90	4
Zen3-5600X	162	43	3.77
Zen2-3700X	209	51	4.10
Zen-1800X	333	84	3.96
平均值	299	73	4.03

法，它直接识别指令中是否有立即数或者偏移量就直接跳过对识别指令的测试；而 Skipscan 通过识别具体的立即数或者偏移量的值（特征值 0x01 和 0x02）来判定是否跳过冗余的立即数和偏移量，这使得判定更加的精准，不会丢失对预留指令的测试。

表 3-5　Skipscan、Sandsifter、UISFuzz 所测试的指令数量的对比

Table 3-5　Comparison of the number of instructions tested by Skipscan, Sandsifter, and UISFuzz

方法	预留指令	合法指令	隐藏指令	指令总数	合法指令占比	隐藏指令占比
Sandsifter[24]	198,984,837	711,167,536	3,463,244	910,152,373	78.14%	21.86%
UISFuzz[25]	<131,270,176	N.D.	2,157,079	131,270,176	N.D.	N.D.
Skipscan	198,984,837	18,191,599	3,463,244	217,176,436	20.22%	79.78%

注：N.D. **是** No Data **的简称，表示在论文中没有发现相关数据**。

综合来看，Skipscan 将预留指令的占比从 21.86% 提升到了 79.78%，而且它能够保证不遗失对预留指令的测试。效率的提升为后续章节提升覆盖率打下了基础，例如，第4章通过增加对指令前缀组合的测试增加 x86 指令空间的覆盖率，第5章通过增加指令的搜索深度的实时检测，及时的增加指令空间搜索深度不足

的情况，从而保证对操作码和操作数的覆盖率。

本章节提出的合法指令中操作数的最小测试集的方法也适合于对 RISC 指令集相关应用的测试，但是针对合法指令中的最小测试集中的元素会存在差异。例如，对含有偏移量的合法指令，其操作数的最小测试集为 {0x00, 0x01, 0xFF}，对于含有立即数的合法指令，其操作数的最小测试集为 {0x00, 0x01, 0x02, 0xFF}。然而，RISC 指令集中偏移量和立即数相关的测试集与 x86 的并不同，例如，采用只测试操作数的边界值的方法，针对立即数和偏移量操作数，其最小测试集都为 {0x00, 0xFF}。

3.8 本章小结

本小节主要阐述了提升 x86 指令空间搜索效率的方法，其主要思想是通过提升预留指令在总的测试指令中的占比。我们提出了基于合法指令的操作数的最小测试集的方法，方法首先识别指令中特定的立即数和偏移量操作数（特征值），然后跳过对冗余的立即数和操作数的测试，从而减少了对冗余的合法指令的测试。我们将提出的方法在指令测试框架中进行具体的实现，并在 8 种来自 Intel 和 AMD 的 x86 处理器上进行测试和评估。实验结果表明，Skipscan 能够将合法指令在总测试的指令中的占比（PLITI）从 78.11% 降低到 20.22%，相对应的，将预留指令的占比（PRITI）从 21.89% 提升到了 79.78%。与现有的方法 Sandsifter，Skipscan 具有高测试效率的同时能够保证不减少对预留指令的测试数量。

第 4 章　基于组合优化的指令前缀生成方法

x86 指令集中的指令前缀能改变其后面指令和操作码的行为或功能，然而现有的检测方法并没有对指令前缀域进行系统的测试。指令前缀不仅可以与已定义的操作码结合实现对新指令功能的扩展，还可以与预留的操作码或预留的操作数组合形成预留指令空间。增加对指令前缀域的系统测试，生成更多的指令前缀的组合，能够达到测试更多的预留指令的效果，从而实现提升对 x86 指令空间的测试覆盖率的目的。

本章节我们首先阐述了现有的指令前缀类型，并分析了指令前缀的作用；然后展示了基于组合优化的指令前缀生成算法，并将算法在我们的测试框架 Skip-scan 中展开了具体的应用和实现；最后我们对提出的基于组合优化的指令前缀生成方法在 8 种来自 Intel 和 AMD 的处理器平台上进行了实验评估，分析了实验结果，并与相关的工作进行了比较和讨论。

4.1　研究动机

一般而言，指令前缀的长度在 1 到 4 个字节之间，然而，现有的工作只测试了指令前缀长度为 1 的情况，从而使得现有方法对指令前缀与预留操作码和预留操作数所组成的预留指令空间的覆盖率较低。x86 指令集中存在的指令前缀可以改变其后面跟随的指令的功能，全面的对指令前缀域进行测试会增加 x86 指令空间的覆盖率。现有的指令前缀可以被看做是有限元素的集合，所以对指令前缀域的遍历搜索就可以转化为一个对有限元素集合的排列组合的生成问题。我们根据指令前缀的使用规则抽象出指令前缀生成的约束条件，并用这些约束条件优化对指令前缀的排列组合生成算法。

4.2　指令前缀类型和使用规则

4.2.1　x86 中指令前缀类型

随着 x86 指令功能的不断丰富，现有的指令前缀种类有遗留前缀（Legacy）、REX 前缀（Register Extensions）、强制前缀（Mandatory）以及 VEX 前缀（Vector Extensions）等。需要注意的是，指令前缀可以叠加，而且不同功能类型的前缀之间不会相互干扰。

x86 指令集中有 11 个遗留前缀，如表4-1中的 Legacy 类型。我们用集合的方

式对其进行表示：S_l={26, 2E, 36, 3E, 64, 65, 66, 67, F0, F2, F3}。S_l 是含有 11 个元素的遗留前缀的集合。指令集手册根据遗留前缀的功能将它们分成四个组，如表中的组 1（Group 1）到组 4（Group 4），Intel 的指令集手册 [96] 规定一个指令组合最多只允许包含每个组中的一个前缀。因为一个组中的不同前缀属于同一个类型，同时使用多个相同类型的前缀会使得功能相互冲突。

表 4-1　指令前缀的功能和分组

Table 4-1　Function and grouping of instruction prefixes

类型	组	前缀	Legacy 中的功能	扩展的功能
Legacy	Group 1	0xF0	Lock	
		0xF2	REPNE/REPNZ	Mandatory, BND
		0xF3	REP or REPE/REPZ	Mandatory
	Group 2	0x2E	CS segment override	Branch not taken
		0x36	SS segment override	
		0x3E	DS segment override	Branch taken
		0x26	ES segment override	
		0x64	FS segment override	
		0x65	GS segment override	
	Group 3	0x66	Operand-size override	Mandatory
	Group 4	0x67	Address-size override	
REX		0x40 至 0x4F	Select 64-bit registers	Specify 64-bit operand
VEX		0xC4、0xC5	Vector Extension	

- 组 1（Group 1）中的前缀 0xF0 被用来对指令上锁，前缀 0xF2 和 0xF3 被用来使字符（String）或者输入/输出指令（Input/Output）重复运行；0xF2 的功能是 REPNE/REPNZ（Repeat-Not-Zero），它还可以被用作强制前缀；0xF3 的功能是 REPE/REPZ（Repeat-Zero），而且它也可以被用来当作强制前缀作用于 POPCNT、LZCNT 和 ASOX 等指令。
- 组 2（Group 2）中的指令在遗留指令类型中作段超越前缀（Segment Override Prefix）使用，如表4-1中所示，这些都是 x86 处理器早期的寻址方式，每个段大小为 64KB，段内可以存放程序运行所需要的代码和数据，从而段超越前缀按照

程序在内存中功能区的划分可以分为三大类：代码段（CS）、堆栈段（SS）、数据段（DS、ES、FS、GS）。一般这些指令前缀被用在需要访问内存的指令中，但是随着指令集的发展这些前缀也会有新的功能扩展。例如，前缀 0x2E 和 0x3E 可以被用在 Jcc 指令的前面，作为分支跳转功能使用（0x2E：Branch not taken, 0x3E：Branch taken）。

- 组 3（Group 3）中只有一个指令前缀 0x66，它在遗留前缀中作为操作数尺寸超越前缀（Operand-size Override）改变操作数的大小，例如将操作数在 16 比特和 32 比特之间切换。同时前缀 0x66 还可以被用来作为一些指令的强制前缀，改变原有的操作码的功能，从而实现新的指令功能的扩展。
- 组 4（Group 4）中只有一个指令前缀 0x67，它在遗留指令前缀中作为地址尺寸超越前缀（Address-size Override）改变访存地址的尺寸，例如将访存地址在 16 比特和 32 比特之间切换。指令所在的执行环境决定其默认的访存地址尺寸，加上前缀 0x67 之后变成其非默认的尺寸。例如，某指令的执行环境使得指令的访存地址尺寸为 32 比特，当指令前面加上 0x67 前缀的时候，指令的访存地址尺寸变为 16 比特。
- REX 前缀是在处理器运行在长模式的时候（64-bit 模式）对寄存器数量和大小都进行了扩展。数量上增加了 Reg 8 至 Reg15 的 8 个通用寄存器，并将寄存器大小扩展成了 64 比特，从而提升了处理器流水线处理数据的吞吐率和精度。我们将 REX 系列的前缀定义为一个集合：S_r={40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F}，此集合中有 16 个元素。
- VEX 前缀被用来扩展向量指令，有两种指令前缀类型 0xC4 和 0xC5。其中 0xC4 是三个字节的 VEX 前缀类型的标识；0xC5 是两个字节 VEX 前缀类型的标识。值得注意的是，VEX 前缀不能与 0xF0、0x66、0xF2 和 0xF3 等前缀一起使用，否则会触发非法指令异常。

4.2.2　指令前缀使用规则

x86 指令前缀的使用要遵循一定的规则 [96]，违反使用规则会使得指令前缀组合无效而被忽略。我们将指令前缀的使用规则进行简单的说明。

- 规则一：当需要遗留（Legacy）前缀的时候可以直接使用，去掉遗留前缀并不会改变指令操作码的功能；但是当遗留前缀被当作强制（Mandatory）前缀使用的时候，去掉前缀则会影响到操作码的功能。
- 规则二：遗留前缀必须要在 REX 前缀的前面，REX 前缀是可选的，但是当指令中使用 REX 前缀的时候，它必须要被放置在操作码的前面。

- 规则三：在一个指令前缀组合中，每个组（Group）中的指令前缀最多只能有一种出现在指令前面，否则将会影响同组指令前缀的功能。
- 规则四：来自四个组（Group 1 至 Group 4）的指令前缀可以以任意的顺序放置，排列顺序不影响其指令组合的功能。
- 规则五：不可同时使用多个 REX 指令前缀，如果指令中存在多个 REX 前缀，则只有紧邻操作码的那一个 REX 前缀是有效的。

4.3 指令前缀生成算法

4.3.1 指令前缀生成算法的约束条件

我们根据现有的指令前缀使用规则（见4.2.2节）得到对应的指令前缀生成的约束条件，并用这些约束条件约束指令前缀的遍历生成。目的是筛除冗余的指令前缀组合，同时能够保证对现有的合规的指令前缀组合的覆盖，从而实现对指令前缀域的高效和高覆盖率的测试效果。

- 约束条件一：如果 REX 前缀被放置到遗留前缀之前，则此指令前缀组合是不合规的，指令前缀生成算法需要抛弃此指令前缀组合。
- 约束条件二：如果指令前缀的长度大于 4，则此指令前缀组合是不合规的，指令前缀生成算法需要抛弃此指令前缀组合。
- 约束条件三：如果指令前缀中有重复的遗留前缀，则算法抛弃此指令前缀组合。
- 约束条件四：如果存在几个指令前缀组合，它们中所有的字节类型都一致，但是只有其排列顺序不一致，则只取其中一种前缀组合进行测试。
- 约束条件五：如果指令前缀组合中 REX 前缀数量大于 1，则算法抛弃此指令前缀组合。

4.3.2 排列组合算法

我们将对指令前缀域遍历的问题转化为生成排列组合的问题。通过观察，我们发现现有的指令前缀为有限元素的集合，例如，遗留前缀集合 S_l 和 REX 前缀集合 S_r，这两个集合中共有 27 个元素。注意 VEX 前缀不参与指令前缀组合的生成，因为 VEX 的编码本身就可以解码为强制前缀相关的指令前缀组合。接下来，我们简要的描述全排列、排列和组合的定义，同时针对每一个定义都有一个例子作为说明。

- 有限集合的全排列

定义 4.1. 集合 S 是一个有 n 个元素的集合 $n-set$，集合 S^m $(m \leq n)$ 是集合 S 的 m 全排列（*m-arrangements*），则 m 全排列中元素的数量可以通过表达式 $A(n,m) = n^m$ 得到。

例 4.1. 定义集合 S 为 {a, b, c}，则集合 S 的 2-arrangements 依次排开为：aa、ab、ac、ba、bb、bc、ca、cb、cc，共有 9 种排列方式。根据定义 1 中的表达式，我们也可以计算出集合 S 的 2-全排列的数量：$A(3,2) = 3^2 = 9$。

- 有限集合的排列

定义 4.2. 集合 S 是一个有 n 个元素的集合 $n-set$，将集合 S^m $(m \leq n)$ 中的重复元素删除，剩下的元素是集合 S 的 m 排列（*m-permutations*），则 m 排列中元素的数量可以通过表达式 $P(n,m) = n(n-1)(n-2)...(n-m+1)$ 得到。

例 4.2. 定义集合 S 为 {a, b, c}，则集合 S 的 2-permutations 依次排开为：ab、ac、ba、bc、ca、cb，共有 6 种排列方式。根据定义 2 中的表达式，我们也可以计算出集合 S 的 2-排列的数量：$P(3,2) = 3 \times 2 = 6$。

- 有限集合的组合

定义 4.3. 集合 S 是一个有 n 个元素的集合 $n-set$，从集合 S 中不按照顺序选 m 个元素 $(m \leq n)$，所有可能的选取方式就是集合 S 的 m 组合（*m-combinations*），则 m 组合中元素的数量可以通过表达式 $C(n,m) = P(n,m)/P(m,m)$ 得到。

例 4.3. 定义集合 S 为 {a, b, c}，则集合 S 的 2-combinations 依次排开为：ab、ac、bc，共有 3 种组合形式。根据定义 3 中的表达式，我们也可以计算出集合 S 的 2-组合数量：$C(3,2) = (3 \times 2)/(2 \times 1) = 3$。

我们使用定义4.1、4.2、和4.3中的表达式计算全排列、排列和组合等方法对应的指令前缀组合的数量，从而在保证不丢失合规指令前缀组合的前提下，量化比较不同方法之间的效率。

4.3.3　现有的排列组合方法的不足

指令前缀域的全遍历可以看作是对指令前缀集合元素的全排列的生成。如第4.3.2节中的定义4.1和例4.1所示，使用全排列的方法会产生大量的冗余的指令前缀组合：（1）形如 aa、bb、cc 样式的前缀组合，前缀组合中存在重复的相同元素的情况；（2）形如 ab、ba 样式的前缀组合，多个前缀组合之间元素种类相同但只有元素的顺序不同。我们使用第4.3.1节中的约束条件三和四可以有效的筛除掉全排列中的情形（1）和（2），把全排列的前缀生成方法约束成为组合的前缀生成方法。

然而，在遍历前缀域的过程中，使用基于组合的前缀生成方法还不能筛除指

令前缀组合中含有多个 REX 前缀以及前缀组合中唯一的 REX 前缀不在前缀组合的最后一个字节位置的情况（REX 前缀没有紧挨着指令操作码）。

4.4 基于组合优化的指令前缀生成算法

根据上一个小节的分析，我们需要对现有的基于组合的前缀生成方法进行优化，目的是筛除指令前缀中含有多个 REX 前缀的情况和前缀组合中唯一的 REX 前缀不在前缀组合的最后一个字节位置的情况。我们使用第4.3.1节中的约束条件一和五来限定基于组合的前缀生成方法。

我们提出一种基于组合优化的指令前缀生成方法来绕过对冗余的含有 REX 前缀的组合。不失一般性，我们将基本的组合优化方法用数学表达式表示为：

$$O(27, m) = C_{11}^{m} + C_{11}^{m-1} \times 16, m \in \{1, 2, 3, 4\} \tag{4-1}$$

$O(27, m)$ 表示基于组合优化的前缀生成方法产生的指令组合的数量。m 表示指令前缀的长度。11 表示遗留前缀集合 S_l 中元素的数量。16 表示 REX 前缀集合 S_r 中元素的数量。C_{11}^{m} 表示前缀组合没有 REX 前缀的情况。$C_{11}^{m-1} \times 16$ 表示在指令前缀组合中只有一个 REX 前缀，且 REX 在前缀组合的最后一个字节位置的情况。当指令前缀长度是 4 的时候，使用基于组合优化的方法生成的指令前缀组合的数量是 $O(27, 4) = C_{11}^{4} + C_{11}^{3} \times 16 = 2,970$。

表 4-2 指令前缀组合的个数与指令前缀长度的对应关系

Table 4-2 The correspondence between the number of instruction prefix combinations and the instruction prefix length

前缀长度	1	2	3	4	元素总数
m-arrangements	27	729	19,683	531,441	551,880
m-permutations	27	702	17,550	421,200	439,479
m-combinations	27	351	2,925	17,550	208,853
m-optimizations	27	231	1,045	2,970	4,273

使用基于组合优化的方法生成指令前缀组合能够获得较高的覆盖率，同时也能实现较好的效率。如表4-2所示，当设置指令前缀最大长度为是 4 的时候，指令前缀长度可以是 1、2、3 和 4 个字节中的任意值。而且，基于组合优化的方法只是去除掉了无效的指令前缀组合。将生成的前缀组合与预留的操作码和预

留的操作数结合能够构建更多的预留指令空间，从而实现提升对预留指令空间的测试覆盖率的目的。此外，同各个指令前缀组合的生成方法（基于全排列的方法、基于排列的方法和基于组合的方法）进行比较，我们发现基于组合优化的指令前缀生成方法将全排列的方法所产生的指令前缀组合的数量从 551,880 个减少到了 4,273 个，从这个量化比较的结果来看，用基于组合优化的方法生成指令前缀能够提升对 x86 指令空间的整体的测试效率，因为此方法绕过了大量的无效的指令前缀组合。

将4.3.1节中的 5 个约束条件全部应用在基于全排列的指令前缀生成方法中就得到了基于组合优化的指令前缀生成方法。采用基于组合优化的指令前缀生成方法生成的指令前缀组合，第一个组合结果是 0x26 2E，然后依次有 2 个字节、3 个字节、4 个字节的指令前缀组合生成，最后的一个指令前缀组合是 0x67 F0 F2 F3，如表4-2的最后一行所示，总共有 4,273 个指令前缀组合需要测试（包括 27 个指令前缀长度为 1 的情况）。

接下来，我们用算法2表示本章节使用的基于组合优化的指令前缀生成方法。本算法优先测试了指令前缀长度为 1 到 4 个字节的情况（约束条件二），如算法2中的第 15 行所示对前缀长度进行了限定。对约束条件一和约束条件五的应用是为了绕过指令前缀组合中存在多个 REX 前缀以及存在的唯一的 REX 前缀没有紧挨着指令操作码的情况（也就是没有在指令前缀组合的最后一个字节位置）；在算法中具体实现如第 17 行到第 19 行所示。对约束条件三的应用是为了筛除指令前缀中存在重复的前缀字节的情况；在算法中的具体实现如第 20 行到 22 行所示。对约束条件四的应用是为了筛除多个指令前缀组合中指令前缀类型完全一致，只是排列顺序不同的情况，算法只取了其中一种进行测试，为了使得筛选方便实现，算法直接取前缀中字节为升序（或者降序）的前缀组合；在算法中具体实现如第 23 行到 32 行所示。需要注意的是，算法 1 中的 m 表示指令前缀组合的最右边的那个字节位；指令前缀组合的最左边是“1”字节的位置，如图4-1所示。

算法 2　基于组合优化的前缀生成

Require: $1 < m \leq 4$

1: G ← InstructionGeneration()
2: PL ← PrefixLength()
3: IsLP ← IsLegacyPrefix()
4: RexP ← HasREXPrefix()
5: RepP ← HasRepeatedPrefix()
6: Order ← OrderCheck()
7: Insn ← G.Instruction　▷ Insn 表示 Instruction

```
 8: Prefix ← Insn.prefix
 9: if PL = 0 then
10:     Generate instructions without prefixes
11: end if
12: if PL = 1 then
13:     Generate instructions with only one prefix
14: end if
15: if 1 < PL ≤ 4 then
16:     //Generate instructions with prefix combinations
17:     if RexP(Prefix(1 to m-1)) then                          ▷ 约束 1 和 5
18:         Skip_the_prefix_combination
19:     end if
20:     if RepP(Prefix(1 to m)) then                            ▷ 约束 3
21:         Skip_the_prefix_combination
22:     end if
23:     if IsLP(Prefix(1 to m)) then
24:         if Order(Prefix(1 to m) then                        ▷ 约束 4
25:             Test_the_prefix_combination
26:         end if
27:     end if
28:     if IsLP(Prefix(1 to m-1)) and RexP(Prefix(m)) then
29:         if Order(Prefix(1 to m-1)) then                     ▷ 约束 4
30:             Test_the_prefix_combination
31:         end if
32:     end if
33: end if
34: if 4 < PL then                                              ▷ 约束 2
35:     Skip_the_prefix_combination
36: end if=0
```

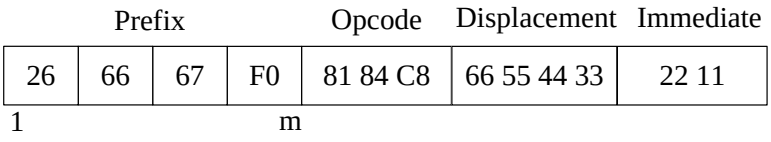

图 4-1　说明算法2中的前缀组合字节序的例子

Figure 4-1　As an example to illustrate the byte order of instruction prefix combinations

注：“0x266667F0” 是前缀组合，“0x81” 是操作码，“0x84” 是 ModR/M 字节，“0xC8” 是 SIB 字节，指令被反汇编成 lock add word ptr es:[eax + ecx*8 + 0x33445566], 0x1122

4.5　测试框架

对指令前缀组合的测试需要结合指令测试的框架，如图4-2所示。本章节主要展示的是对指令前缀域的测试方法。基于差分测试的指令测试框架，其基本思想是把生成的指令分别送入处理器和反汇编器，然后比较处理器和反汇编器的输出，如果其输出结果与预期的不一致，则将指令放入错误报告文件中，最后对错误报告中的指令进行重新检查。

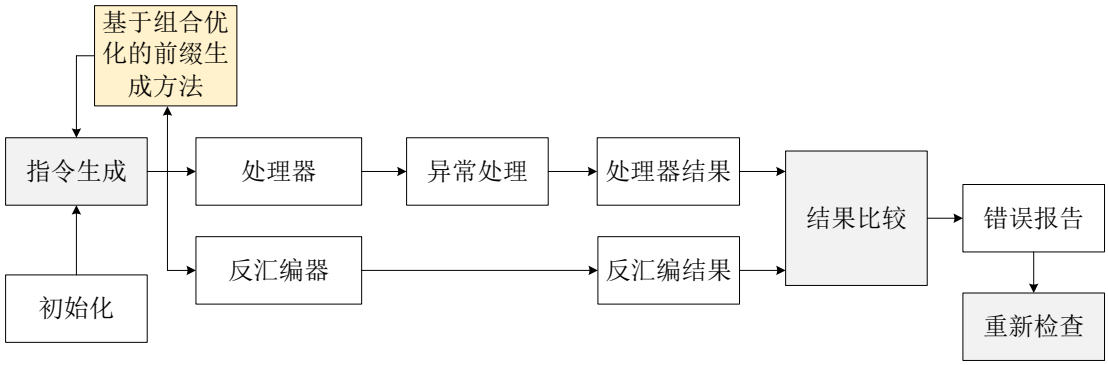

图 4-2　基于优化组合的指令前缀生成方法的测试框架

Figure 4-2　Test framework of instruction prefix generation method based on optimized composition

基于组合优化的前缀生成方法模块：通过对生成的指令进行分析，如果存在指令前缀，则通过 5 种约束条件对指令前缀组合进行筛选，跳过冗余的指令前缀组合，从而在保证对合规的指令前缀组合的覆盖的前提下提升测试的效率。值得注意的是，绕过不合规的指令前缀组合，是为了在有限的时间内测试更多的合规的指令前缀组合，从而达到提升测试覆盖率的目的。

图4-2中的初始化、处理器、反汇编器、异常处理、处理器结果、反汇编结果、结果比较、错误报告、重新检查等模块的功能与第3章的图3-6对应的功能描述一致。

我们对检测出的隐藏指令、对隐藏指令重新检查以及隐藏指令可能存在的功能等都放到第6章节讨论。

4.6　实验评估

我们对提出的基于组合优化的指令前缀生成方法在 Intel 和 AMD 的处理器上进行评估，所使用的处理器如表4-3所示。我们使用的操作系统是 Ubuntu 16.04 64-bit，内核版本为 4.15.0。

表 4-3　评估指令前缀生成方法所使用的处理器

Table 4-3　The processors used to evaluate the method of instruction prefix generation

处理器	微架构	处理器系列	发布时间
i5-11600KF	Rocket Lake	Core	March 16, 2021
i7-10700KF	Comet Lake	Core	April 30, 2020
Gold 5218	Cascade Lake	Xeon	April 2, 2019
i5-7300HQ	Kaby Lake	Core	January 3, 2017
i7-6700	Skylake	Core	September 27, 2015
Zen3-5600X	Zen3	Ryzen	November 5, 2020
Zen2-3700X	Zen2	Ryzen	July 7, 2019
Zen-1800X	Zen	Ryzen	March 2, 2017

本小节评估和展示我们的方法增加了对 x86 指令空间的测试覆盖率。需要注意的是，增加对指令前缀域的覆盖，能增加对预留指令空间的覆盖。指令空间中的预留操作码和预留操作数都能够构成预留指令。基于组合优化的指令前缀生成方法能够产生大量的指令前缀组合，每个指令前缀组合又可以与现有的预留操作码和操作数结合组成预留指令，从而增加了对隐藏指令的测试覆盖率。

4.6.1　度量指标

我们在不改变现有的深度优先遍历算法的测试深度的前提下，展示我们本章节提出的基于组合优化的指令前缀生成方法的测试结果。

覆盖率提升的度量指标：覆盖率的增加主要表现在所测试的预留指令数量的增加 (Number of Reserved Instructions，NRI)，所以我们提出使用预留指令数量的多少来衡量对 x86 指令空间的覆盖率的提升效果。

最为直观的反映我们增加对指令前缀长度测试的效果的是发现的隐藏指令的数量 (Number of Hidden Instructions，NHI)。还有一个辅助的指标是隐藏指令在所测试的预留指令的数量中的占比（Proportion of Hidden Instructions in Reserved Instructions，PHIRI）。

$$PHIRI = \frac{Number\ of\ Hidden\ instructions}{Number\ of\ Reserved\ Instructions} \times 100\% \tag{4-1}$$

4.6.2　实验结果

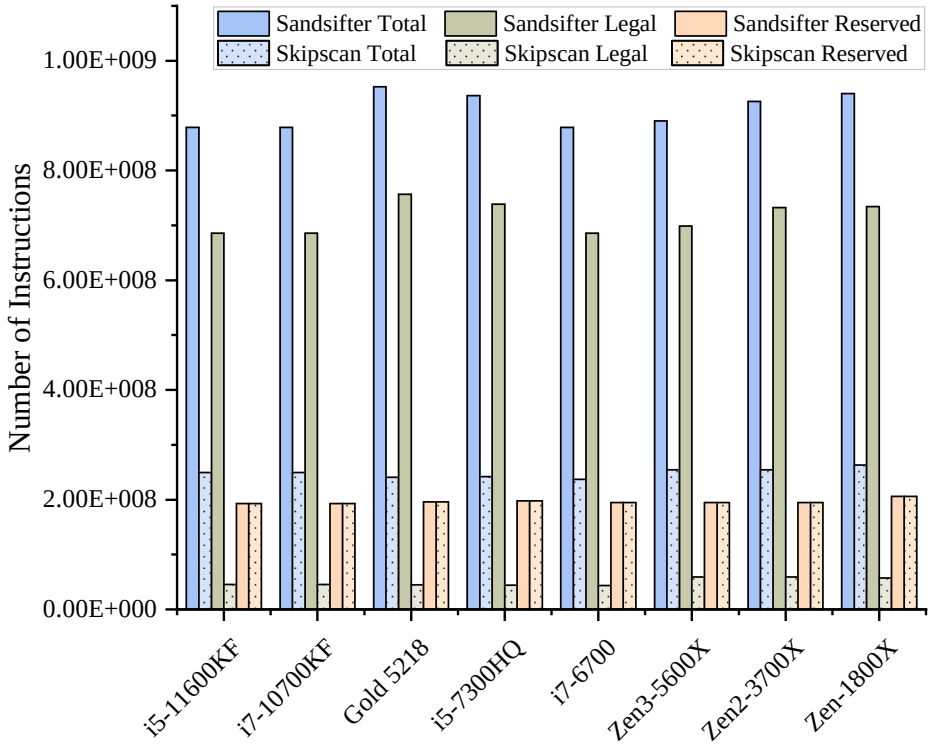

图 4-3　Skipscan 和 Sandsifter 所测试的指令数量的对比

Figure 4-3　Comparison of the number of instructions tested by Skipscan and Sandsifter

在第3章已经讨论过，在指令前缀长度为 1 的时候，Skipscan 能够保证与 Sandsifter 测试同样数量的预留指令，同时提升了对 x86 指令空间中预留指令的搜索效率。我们将第3章的数据通过图的形式进行表述，如图4-3中的 “Reserved” 柱所示，Skipscan 测试的预留指令数量与 Sandsifter 一样。这为我们接下来比较 Skipscan 测试的预留指令数量大于 Sandsifter 提供了一个依据。

我们将 Skipscan 在 8 个处理器上面的测试结果与 Sandsifter 进行比较。通过对表4-4和表4-5的分析，我们总结出了表4-6，它主要展示了指令前缀长度为 1 和指令前缀长度为 4 的时候的预留指令数量、隐藏指令数量和隐藏指令在预留指令中的占比这些指标。表4-4展示的是 Skipscan 在指令前缀长度为 1 的情况下所测试的指令条数，在 8 个所测试的处理器上测试的预留指令数量（NRI）的平均值为 198,984,837。值得注意的是，Skipscan 在指令前缀长度为 1 的时候，所测试的预留指令的数量与 Sandsifter 是一致的。然后，从表4-5中，我们可以发现，Skipscan 在 8 个处理器上测试的预留指令的数量（NRI）的平均值为 51,645,959,589，是指令前缀长度为 1 的时候 Skipscan 测试指令条数的 260 倍。所以，我们可以得到，Skipscan 测试的预留指令数量是 Sandsifter 的 260 倍。很显然，随着指令前缀长

表 4-4　当指令前缀长度为 1 的时候 Skipscan 所测试的指令数量

Table 4-4　The number of instructions tested by Skipscan, when prefix length is one

处理器	合法指令数量	预留指令数量（NRI）	隐藏指令数量（NHI）
i5-11600KF	45,477,719	204,100,708	3,351,847
i7-10700KF	45,477,719	204,100,708	3,347,748
Gold 5218	44,858,193	196,228,010	3,045,384
i5-7300HQ	44,288,021	197,814,409	3,961,284
i7-6700	43,779,413	193,556,383	3,347,748
Zen3-5600X	59,283,642	195,040,790	3,346,148
Zen2-3700X	59,283,642	195,040,790	3,346,148
Zen-1800X	57,408,877	205,996,896	3,959,684
平均值	49,982,153	198,984,837	3,463,249

注：指令前缀长度为 1 的时候，Skipscan 测试的预留指令数量与 Sandsifter 是一致的。

表 4-5　当指令前缀长度为 4 的时候 Skipscan 所测试的指令数量

Table 4-5　The number of instructions tested by Skipscan, when prefix length is four

处理器	合法指令的数量	预留指令的数量（NRI）	隐藏指令的数量（NHI）
i5-11600KF	5,157,582,654	48,675,472,707	1,208,843,721
i7-10700KF	5,447,429,928	52,919,230,073	1,248,932,821
Gold 5218	6,117,550,815	50,729,318,825	1,285,688,183
i5-7300HQ	5,404,241,355	52,324,277,633	1,292,718,193
i7-6700	4,293,856,886	50,303,692,315	1,248,247,661
Zen3-5600X	7,192,100,379	53,661,195,090	1,281,276,849
Zen2-3700X	7,131,777,208	51,940,189,912	1,252,170,755
Zen-1800X	7,797,925,554	52,614,300,156	1,471,817,097
平均值	6,067,808,097	51,645,959,589	1,286,211,910

度的增加，Skipscan 测试的预留操作码和预留操作数的种类和数量也会增加。通过表4-2中的数据：当指令前缀长度为 1 的时候，指令前缀数量为 27；当指令前缀长度为 4 的时候，指令前缀组合的数量总共有 4,273，是指令前缀长度为 1 的时候的 158 倍。如果测试指令的数量与指令前缀长度的增加成正比的话，只从指令前缀数量增加的数量上看，指令前缀长度为 4 的时候所测试的预留指令数量应该是指令前缀长度为 1 的时候的 158 倍，但是实验结果展示为 260 倍。经过对指令集的分析，我们发现，当指令前缀为 1 的时候，其预留操作码和预留的操作数要小于指令前缀长度为 2 的时候的情况。也就是说，随着指令前缀长度的增加会有更多的预留操作码和预留操作数类型。这又近一步的证明了，对指令前缀长度的增加能够增加对预留指令的测试覆盖率。

表 4-6　Skipcan 实验结果的度量指标比较

Table 4-6　Comparison of metrics of experimental results of Skipscan.

指标	指令前缀长度为 1	指令前缀长度为 4	倍数
指令前缀组合数量	27	4,273	158
预留指令数量（NRI）	198,984,837	51,645,959,589	260
隐藏指令数量（NHI）	3,463,249	1,286,211,910	371
隐藏指令/预留指令（PHIRI）	1.74%	2.49%	1.43

而且，如表4-6中数据所示，当指令前缀长度增加的时候，所得到的隐藏指令的数量也会随之增加。当指令前缀长度为 1 的时候，在 8 种处理器平台上发现的隐藏指令数量的平均值为 3,463,249；当指令前缀长度为 4 的时候，发现的隐藏指令数量的平均值为 1,286,211,910。指令前缀长度为 4 的时候发现的隐藏指令的数量为指令前缀长度为 1 的时候的隐藏指令数量的 371 倍。隐藏指令的倍数增加比预留指令的倍数增加要大的多，这说明，随着指令前缀长度的增加，隐藏指令在预留指令中的占比也会增加。指令前缀长度为 1 的时候，所测试的总的预留指令数量的平均值为 198,984,837，对应隐藏指令数量的平均值为 3,463,249，隐藏指令在预留指令中占的比例为 1.74%。指令前缀长度为 4 的时候，所测试的总的预留指令数量的平均值为 51,645,959,589，对应隐藏指令数量的平均值为 1,286,211,910，隐藏指令在预留指令中占的比例为 2.49%。隐藏指令的占比增加又近一步证明了，指令前缀长度增加是有意义的，因为它能发现更多的潜在的隐藏指令。

4.7 讨论

本章节使用基于组合优化的指令前缀生成方法产生了大量的指令前缀组合，相比于现有的指令空间测试方法，例如基于随机的方法，基于全遍历的方法，以及基于深度优先遍历的方法（Sandsifter[24] 和 UISFuzz[25]），Skipscan 对 x86 指令前缀域的覆盖率较高。因为 Skipscan 生成了大量的指令前缀组合，这些指令前缀组合能够与更多的预留指令操作码和操作数结合，从而组合成更多的预留指令。我们将本章节使用的方法与现有的方法进行比较和讨论。

Skipscan 能保证对指令前缀域的覆盖，现有的随机的方法 [56, 57] 只重点关注了对操作码域的覆盖。基于组合优化的指令前缀生成方法能够产生现有的指令前缀使用规则之下的所有的合规的指令前缀的组合。而基于随机指令生成的方法对前缀域的测试是随机的，这不仅会产生大量的非前缀的字节（不在表4-1中的字节值），还会产生大量的冗余的指令前缀组合。总的来看，基于随机的方法很难在可接受的时间范围内对 x86 指令前缀域进行高覆盖率的测试。

对指令域进行全遍历在可接受的时间之内是无法完成测试的。因为基于全遍历的方法会测试到大量的非前缀字节和大量的冗余的指令前缀组合。我们采用基于组合优化的方法，能够在现有的指令使用规则的引导之下，巧妙的绕过非前缀字节和冗余的指令前缀组合，从而使得在可接受的时间内完成对所有的合规的指令前缀组合的测试变得可行。

与现有的方法 Sandsifter 和 UISFuzz 相比，Skipscan 系统的测试了指令前缀域。Sandsifter 和 UISFuzz 只测试了指令前缀长度为 1 的时候的情况，相比之下，Skipscan 测试了指令前缀长度为 1、2、3 和 4 个字节的情况。增加对指令前缀域的测试不仅能增加指令前缀组合与现已测试到的预留指令操作码和操作数的覆盖，还能发现更多的预留指令操作码和操作数。因为，随着指令前缀长度的增加，有些操作码的功能会发生改变。从对实验测试的结果分析可以得出，当指令前缀长度为 4 的时候所测试的预留指令的数量是指令前缀长度为 1 的时候的 260 倍，而指令前缀组合的数量只是指令前缀长度为 1 的时候的 158 倍；这说明随着指令前缀长度的增加，预留指令操作码和操作数的数量也增加了。

需要注意的是，本章节提出的基于组合优化的指令前缀生成方法只适合于对 x86 指令集相关应用的测试，例如处理器、反汇编器、ISA 模拟器和 CPU 模拟器等。不适合于对 RISC 指令集相关应用的测试，因为我们的方法是根据分析 x86 指令格式的前缀域得到的，然而 RISC 的指令格式中没有指令前缀域。

4.8　本章小结

本章节我们首先分析了对指令前缀域测试的动机和必要性，针对现有的指令前缀域覆盖率不高的问题，提出了基于组合优化的前缀组合生成方法。在分析了现有的排列组合方法的基础之上，我们根据现有的指令前缀使用规则抽象出了限定指令前缀生成的约束条件，将约束条件应用在基于全排列的组合生成方法之上形成了我们提出的基于组合优化的指令前缀生成方法。然后，我们将基于组合优化的指令前缀生成方法应用在了指令差分测试的框架中。并在 8 种处理器平台上对我们的算法进行了评估，实验结果表明，基于组合优化的前缀生成方法能够使 Skipscan 测试更多的预留指令，同时也发现更多的隐藏指令。与 Sandsifter 和 UISFuzz 相比较，Skipscan 增加了对前缀域的系统性的测试，具有对 x86 指令空间的高覆盖率的优点。

第 5 章　基于必要搜索深度实时检测的 DFS 方法

x86 指令空间的搜索深度直接决定了对指令操作码和其操作数的覆盖率。过度的增加搜索深度会增加对冗余的立即数、偏移量和无效指令空间测试。然而搜索深度不足会丢失对部分操作码、ModR/M、SIB 和寄存器操作数的覆盖，这不仅影响到了合法指令的搜索，而且也能影响到预留指令的搜索和测试。现有的方法 [24, 25] 都采用识别指令长度的变化这个单一的机制来增加 DFS 算法的搜索深度，不仅会导致错过对一些合法的指令操作码和操作数的测试，而且还错过对预留的指令操作码和操作数的测试。针对 DFS 算法搜索深度不足的问题，我们提出了基于必要搜索深度实时检测的 DFS 算法，方法通过实时的对比搜索深度与必要搜索深度的值，当搜索深度小于必要搜索深度的时候，就及时的增加搜索深度，从而保证对预留操作码和预留操作数的覆盖。

本章节，我们首先通过几个例子分析 x86 指令空间搜索深度不足的问题；然后，我们详细阐述了现有的 x86 指令集中的指令格式，并总结了每一种指令格式对应的必要搜索深度；紧接着，我们展示了基于必要搜索深度实时检测的 DFS 算法；最后，我们将提出的方法在 4 种 x86 处理器平台上进行了实验评估，分析了实验结果，并与相关的工作进行了比较和讨论。

5.1　研究动机

深度优先遍历算法（Depth First Search, DFS）只通过检测指令长度变化这一种机制来增加对 x86 指令空间的搜索深度是不够的。在第3章我们阐述了控制 DFS 算法的搜索深度的两种基本的机制：第一种机制（机制 1）是通过检测指令长度的变化，当指令长度由短变长的时候，将 DFS 算法的搜索深度增加 1；第二种机制（机制 2）是通过检测深度字节值，当深度字节值为 0xFF 的时候，则算法认为搜索深度已经足够，将搜索深度减少 1。然而，只通过机制 1 来增加 DFS 的搜索深度是不够的。例如，图5-1中所示，0E 指令的长度为 1，指令 0F 01 00 的长度为 3，当 DFS 算法从指令 0E 测试到指令 0F 01 00 的时候，指令长度由 1 变为了 3，根据机制 1 测试深度应该由原来的 1 变为 2。但是，此时的测试深度为 2，这个测试深度是不够的，因为 0F 01 00 的第三个字节也需要被测试，因为它是 ModR/M 字节，可被用来扩展新的指令，例如操作码的编码有一部分可能存在于 ModR/M 中；另外，ModR/M 代表了不同的寄存器类型和寻址类型，决定

了操作码调用的寄存器资源。

我们根据现有的 x86 指令格式，总结出每种格式对应的搜索深度不足的情况，如表5-1所示。需要注意的是，表5-1中只是列举了几个例子，指令空间中有更多的搜索深度不足的情况，我们无法一一列举。表5-1中的第二列是现有的方法 Sandsifter 的实际搜索深度，第三列是我们通过观察指令格式总结的对应的必要搜索深度。我们在第四列举出了一些反例，这些反例是为了说明第一列中的 xx[1] 是需要被遍历到的，因为它们都是一些具备功能的合法指令，其周围有对应的预留指令。

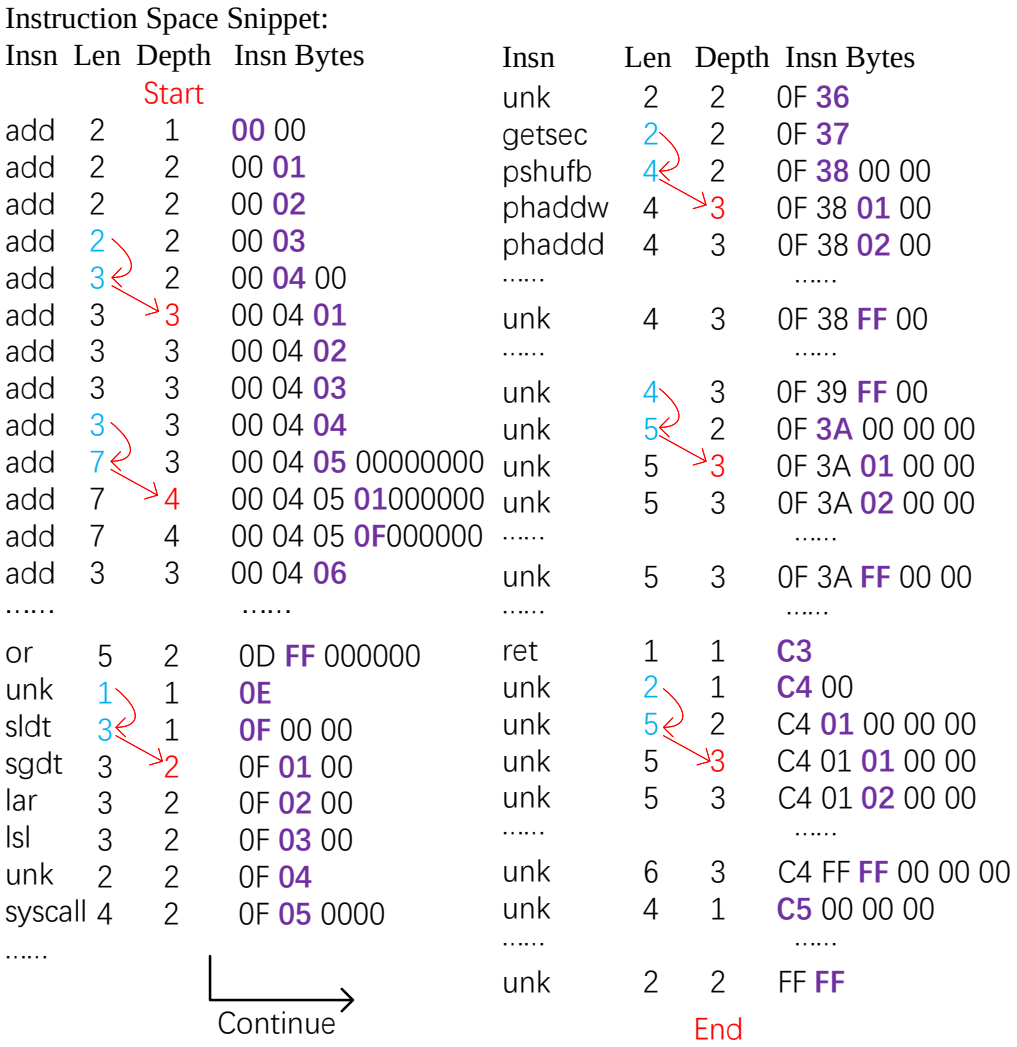

图 5-1　现有的方法中表明测试深度不足的指令流片段

Figure 5-1　Fragments of instruction stream that indicate insufficient test depth in existing methods

通过上面的分析，我们面临的问题是：在现有的 DFS 算法中机制 1 和机制

[1]xx 表示 0x00 到 0xFF 之间的所有可能的值。

2 都被使用的前提下，如何能够有效的增加指令的搜索深度。我们需要一种能够检测出指令搜索深度不足的方法，而且当指令搜索深度不足的时候，方法能及时的增加 DFS 的搜索深度，以保证对预留指令操作码和操作数的覆盖。

表 5-1　现有方法的搜索深度不足的例子

Table 5-1　The search depth of existing DFS algorithm is insufficient

指令类型	实际搜索深度	必要搜索深度	反例
0F 01 xx xx	2	3	0F 01 20 (smsw), 0F 01 28 (reserved)
0F 38 01 xx xx	3	5	0F 38 01 10 (phaddw)
0F 3A 01 xx xx	3	5	0F 3A 01 01 10 (reserved)
C4 03 FD xx xx xx	3	6	C4 03 FD 0F 01 01 (vpalignr)
C5 7D xx xx xx	2	5	C5 7D 6F DA (vmovdqa)

5.2　指令的必要搜索深度

为了提升对 x86 指令空间的搜索深度，我们首先需要分析 x86 指令集中指令的格式，然后得到每种指令格式对应的必要搜索深度。

我们将指令格式按照标识字节来分类，例如 0x0F、0x0F 38、0x0F 3A、0xC4、0xC5。x86 指令集中指令操作码可以有一个字节、两个字节和三个字节。其中一个字节操作码的指令类型是 x86 指令集中最古老的操作码格式，也就是 0x00 到 0xFF 的 256 种可能的值。在指令发展之初，由于处理器功能较为单一，对应的指令数量有限，使用 256 种操作码已经能够表示所有的指令类型。但是，随着处理器功能的不断丰富，同时伴随着指令集的扩展，指令的数量也在不断的增加并且超过了 256 种，所以就不得不对指令的操作码字节进行扩展，从原来的一个字节变为两个字节。于是，指令编码设计者选择了 0x0F 这个字节，将其作为一个 Escape 前缀来扩展两个字节的操作码，其操作码格式为 0x0F xx，显然，两个字节的操作码又可以多表示 256 条指令。同样的，后来为了扩展新的指令格式，又推出了三个字节操作码的格式，形如 0x0F 3A xx 和 0x0F 38 xx，它们分别又可以表示 256 条指令。随着指令集中向量指令的拓展，Intel 最开始的方法是通过前缀 0x66 和 0xF3 等强制前缀来扩展新的指令空间，但是这会使得指令解码变得极其复杂，且容易引起歧义，因此处理器厂商设计出了新的向量指令格式，例如利用 0xC4 和 0xC5 作为标识字节扩展新的指令空间。注意，当处理器的解码器

遇到这些标识字节的时候都会自动的识别对应的指令类型。我们的算法也需要以这些标识字节为依据来识别不同的指令类型，从而判断其搜索深度是否达到了必要搜索深度。

表 5-2　不同指令格式需要遵守的必要搜索深度

Table 5-2　The essential search depth that different types of instruction should obey

指令类型	指令格式标识	必要搜索深度	需要遍历的指令域
一个字节操作码	None	3	Opcode + ModR/M + SIB
两个字节操作码	0F	4	0F + Opcode + ModR/M + SIB
三个字节操作码	0F 38	5	0F 38 + Opcode + ModR/M + SIB
三个字节操作码	0F 3A	5	0F 3A + Opcode + ModR/M + SIB
向量指令	C4	6	C4 + VEX + VEX + Opcode + ModR/M + SIB
向量指令	C5	5	C5 + VEX + Opcode + ModR/M + SIB

本章节重点讨论的是保证对操作码、ModR/M、SIB 字节的覆盖率的问题。通过保证对指令空间的搜索深度能保证对这些字节的覆盖。我们首先总结现有的指令格式对应的必要搜索深度。不同指令的必要搜索深度与其自身的操作码长度和前缀长度等有关系。如表5-2所示，第一列中 None 表示不需要标识字节，代表操作码为一个字节的情况，其必要搜索深度是 3 个字节，也即操作码、ModR/M 和 SIB。同样的，两个字节操作码（0x0F xx）的必要搜索深度为 4 个字节，三个字节的操作码（0x0F 38 xx 和 0x0F 3A xx）的必要搜索深度为 5 个字节。向量指令 0xC4 xx xx 只是代表了前缀，其后面紧跟着操作码、ModR/M 和 SIB 等字节，因此 0xC4 标识字节所代表的指令的必要搜索深度是 6，包括前缀的 3 个字节、操作码和操作数的 3 个字节。同样的，向量指令 0xC5 xx 也代表前缀，其后面紧跟着操作码、ModR/M 和 SIB 等字节，因此 0xC5 代表的指令的必要搜索深度是 5，包括前缀的 2 个字节、操作码和操作数的 3 个字节。

5.3　基于必要搜索深度实时检测的 DFS 算法

实现基于必要搜索深度实时检测的 DFS 算法需要分步骤进行。第一步，识别出指令搜索深度不足的指令。第二步，将指令长度与对应指令格式的必要搜索深度进行对比，如果指令长度小于必要搜索深度的值，将指令搜索深度设置为指

令长度值；如果指令长度值大于必要搜索深度值，则利用表5-2中对应的必要搜索深度将 DFS 的搜索深度增加到指定的值。

第一步：判定指令搜索深度不足首先要识别到指令长度的变化，当指令长度由短变为长的时候，对比正在测试指令的搜索深度的值与表5-2中的必要搜索深度的大小。算法识别指令格式是通过检查指令中的标识字节来实现的，当算法检测到指令中有指令格式对应的标识字节时，就能够唯一的确定所检测指令的格式，同时就能知道其对应的必要搜索深度。如果指令搜索深度值小于必要搜索深度的值，则正在测试的指令的搜索深度不足。

第二步：当检测到搜索深度不足的指令之后，需要比较指令长度与必要搜索深度的值。如果指令长度小于必要搜索深度的值，那么直接将指令的搜索深度设置为指令长度的值。因为指令长度以外的字节都是无效指令，所以这种情况依然遍历到必要搜索深度是没有意义的。如果指令长度大于必要搜索深度的值，算法直接将搜索深度设置为必要搜索深度。如果检测到指令中有立即数或者偏移量操作数，则还需要在必要搜索深度的基础之上增加一个字节的搜索深度。

我们以指令 0F 01 00 为例，简单阐述增加指令搜索深度的过程。在第5.1小节已经讨论过了指令 0F 01 00 搜索深度不足的的案例，接下来我们分析如何使用基于必要搜索深度实时检测的 DFS 算法增加对 x86 指令空间的搜索深度。首先，当指令 0E 被执行以后紧跟着就执行指令 0F 01 00，此时的指令长度由 1 变为了 3，则根据机制 1（见5.1），指令的搜索深度由 1 变为了 2。然而，此时的指令搜索深度的值 2 小于必要搜索深度值 3，且指令长度值 3 小于必要搜索深度值 4，所以应该将指令的搜索深度变为指令长度值 3，然后从新对指令 0F 01 00 进行测试。当重新测试指令 0F 01 00 的时候，紧随着指令 0F 01 00 的指令应该为 0F 01 01，相比于图5-1中所示的 0F 02 00，我们的方法增加了搜索深度。

我们将基于必要搜索深度实时检测的 DFS 算法的伪代码展示在算法3中。首先，我们在算法的第 1 行定义了指令的生成函数 Instruction_Generation()，方法需要存储最近执行的三条指令，以及他们的指令长度和对应的搜索深度值，在算法第 4 行到第 8 行展示。指令的长度需要通过处理器执行之后才能得到，所以算法的第 2 行定义了指令的执行过程的函数 Instruction_Execution()。算法通过第 9 行来检测指令长度的变化，当指令长度发生变化的时候，算法确定指令的格式，对比搜索深度与必要搜索深度的值，如算法第 12 行到 57 行所示。值得注意的是，算法针对每种指令格式都要首先判断指令长度与必要搜索深度的值，如果指令长度小于必要搜索深度值，则将搜索深度设置为指令长度的值，如算法的

第 16、25、34、43、52 行所示；如果指令长度大于必要搜索深度值，则将搜索深度设置为必要搜索深度的值，算法判断搜索深度与必要搜索深度的值的时候，需要参考表5-2，如算法的第 19、28、37、46、55 行所示。另外，如果检测到所测试的指令中含有立即数和偏移量操作数，那么算法还要额外的增加搜索深度 1 个字节，如算法第 58 到 63 行所示，这是为了保证对立即数和偏移量的边界的测试，从而不漏掉其中可能夹杂的预留指令。

算法 3 基于必要搜索深度实时检测的方法

Require: $0 \leq Search\_Depth \leq Instruction\_Length$
Require: $0 \leq Search\_Depth \leq Essential\_Search\_Depth$
Ensure: $0 \leq Instruction\_Length$
1: G ← Instruction_Generation()
2: E ← Instruction_Execution()
3: D ← Instruction_Disassembly()
4: **for** i = 0 to 3 **do** ▷ 缓存最近测试的两条指令，以及其对应的指令长度和指令的测试深度；
5: Instruction Buffers ← G.Instruction ▷ 最近测试的两条指令；
6: Length Buffers ← E.Length ▷ 最近测试的两条指令的指令长度；
7: Depth Buffers ← Instruction Search Depth ▷ 最近测试的两条指令的测试深度；
8: **end for**
9: Instruction_Length_Changes ← Instruction_Length_1 < Instruction_Length_2
10: Has_Imme ← D.Instruction_2.Imme=True ▷ Instruction_Length_2 为搜索深度不足的指令；
11: Has_Disp ← D.Instruction_2.Disp=True
12: **if** Instruction_Length_Changes **then**
13: **if** (Instruction has None) **then**
14: **if** (Instruction_Length_2 ≤ 3) **then**
15: *Depth_Byte* ← 0*x*00
16: Search_Depth ←Instruction_Length_2 ▷ 搜索深度值为指令长度；
17: **else if** (3 < Instruction_Length_2) **then**
18: *Depth_Byte* ← 0*x*00
19: Search_Depth ←3 ▷ 参考表5-2的第二行；
20: **end if**
21: **end if**
22: **if** (Instruction has 0F) **then**
23: **if** (Instruction_Length_2 ≤ 4) **then**
24: *Depth_Byte* ← 0*x*00
25: Search_Depth ←Instruction_Length_2 ▷ 搜索深度值为指令长度；
26: **else if** (4 < Instruction_Length_2) **then**
27: *Depth_Byte* ← 0*x*00
28: Search_Depth ←4 ▷ 参考表5-2的第三行；
29: **end if**
30: **end if**
31: **if** (Instruction has 0F 38 or 0F 3A) **then**

```
32:         if (Instruction_Length_2 ≤ 5) then
33:             Depth_Byte ← 0x00
34:             Search_Depth ←Instruction_Length_2          ▷ 搜索深度值为指令长度;
35:         else if (5 < Instruction_Length_2) then
36:             Depth_Byte ← 0x00
37:             Search_Depth ←5                             ▷ 参考表5-2的第四、五行;
38:         end if
39:     end if
40:     if (Instruction has C4) then
41:         if (Instruction_Length_2 ≤ 6) then
42:             Depth_Byte ← 0x00
43:             Search_Depth ←Instruction_Length_2          ▷ 搜索深度值为指令长度;
44:         else if (6 < Instruction_Length_2) then
45:             Depth_Byte ← 0x00
46:             Search_Depth ←6                             ▷ 参考表5-2的第六行;
47:         end if
48:     end if
49:     if (Instruction has C5) then
50:         if (Instruction_Length_2 ≤ 5) then
51:             Depth_Byte ← 0x00
52:             Search_Depth ←Instruction_Length_2          ▷ 搜索深度值为指令长度;
53:         else if (5 < Instruction_Length_2) then
54:             Depth_Byte ← 0x00
55:             Search_Depth ←5                             ▷ 参考表5-2的第七行;
56:         end if
57:     end if
58:     if (Has_Imme or Has_Disp) then      ▷ 如果发现搜索深度不足的指令中有立即数或者偏移量，则在
59:         if Depth_Byte is not Imme or Disp then ▷ 满足必要搜索深度的基础之上，还要再将搜索深度加 1。
60:             Depth_Byte ← 0x00
61:             Search_Depth ← Search_Depth + 1
62:         end if
63:     end if
64: end if
```

5.4　测试框架

本章节使用的测试框架与图3-6和图4-2所使用的基本测试框架都是基于指令的差分测试的方法，区别在于图5-2需要实时的检测指令格式、指令长度和 DFS 的搜索深度。本章节使用的基于必要搜索深度实时检测的 DFS 算法，通过检测指令长度的变化的同时要分析现在正在测试的指令的格式以及其对应的搜索深度。如果检测到指令长度变化且通过机制 1 增加搜索深度后，搜索深度仍然

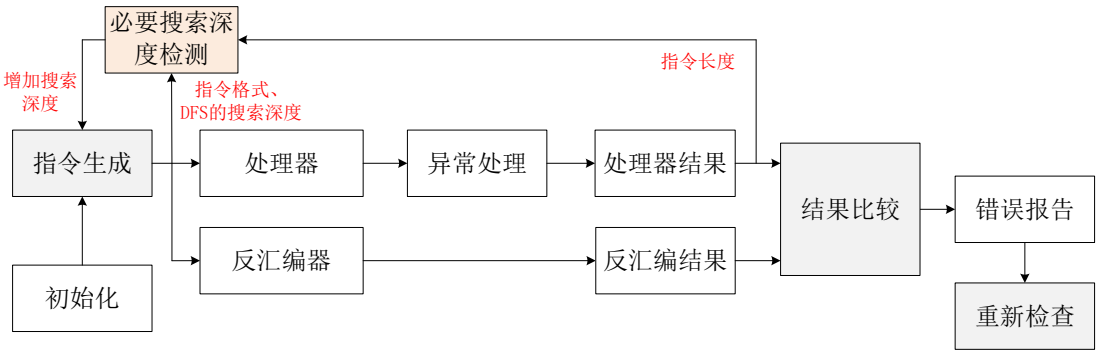

图 5-2　基于必要搜索深度实时检测的 DFS 算法测试框架

Figure 5-2　DFS algorithm testing framework based on real-time detection of essential search depth

小于必要搜索深度，则必要搜索深度检测模块就及时的增加搜索深度，具体的增加方法我们已经展示在算法3中。

图5-2中的初始化、处理器、反汇编器、异常处理、处理器结果、反汇编结果、结果比较、错误报告、重新检查等模块的功能与第3章的图3-6对应的功能描述一致。

我们对检测出的隐藏指令、对隐藏指令重新检查以及隐藏指令可能存在的功能等都放到第6章节讨论。

5.5　实验评估

我们在如表5-3中所示的 4 种不同的 x86 处理器平台上对我们的基于必要搜索深度实时检测的 DFS 算法进行了评估。实验评估所使用的操作系统是 Ubuntu 16.04 64-bit，内核版本为 4.15.0；处理器运行在 64-bit 的长模式；Capstone 的版本是 4.0.2。

5.5.1　度量指标

我们在保证指令前缀长度为 1 的条件下，展示我们本章节提出的基于必要搜索深度实时检测的 DFS 算法的测试结果。我们主要从测试的预留指令数量和隐藏指令数量分析测试覆盖率的增加。

覆盖率的增加主要表现在所测试的预留指令数量 (Number of Reserved Instructions，NRI) 的增加，所以我们提出使用预留指令数量的多少来衡量对 x86 指令空间的覆盖率的提升效果；另一个衡量指标是隐藏指令的数量 (Number of Hidden Instructions，NHI) 的增加；还有一个辅助的指标是隐藏指令在所测试的预

表 5-3　评估基于必要搜索深度实时检测的 DFS 算法所用的处理器

Table 5-3　The processors used to evaluate the DFS algorithm based on real-time detection of the essential search depth

处理器	微架构类型	发布时间
i5-11600KF	Rocket Lake	March 16, 2021
Xeon-W3175X	Skylake	January 30, 2019
Ryzen-5600X	Zen 3	November 5, 2020
Ryzen-3700X	Zen 2	July 7, 2019

留指令的数量中的占比（Proportion of Hidden Instructions in Reserved Instructions，PHIRI）。

$$PHIRI = \frac{Number\ of\ Hidden\ instructions}{Number\ of\ Reserved\ Instructions} \times 100\% \tag{5-1}$$

5.5.2　实验结果

表 5-4　实验测试的指令数量

Table 5-4　The number of instructions tested in the experiment

处理器	总共指令数量	NRI	NHI	PHIRI	时间（分钟）
i5-11600KF	91,525,882,473	88,853,170,655	7,163,228	0.0080%	19,396
Xeon-W3175X	86,325,093,801	83,965,570,251	7,164,092	0.0085%	18,319
Ryzen-5600X	56,002,616,802	6,175,877,839	3,041,596	0.049%	6,312
Ryzen-3700X	32,582,160,498	6,175,877,839	3,041,286	0.049%	4,710
平均值	66,608,938,394	46,292,624,146	5,102,551	0.011%	12,184

如表5-4中所示，由于增加了对指令空间的搜索深度，我们的方法测试了更多的指令，包括合法指令和预留指令，同时也花费更多的时间。

接下来，我们展示了基于必要搜索深度实时检测的方法所测试的总的指令数量、预留指令数量、隐藏指令数量。我们以指令数量为度量标准来衡量我们的方法确实增加了对 x86 指令空间中预留指令的覆盖率。如第4章表4-4中所示，

在增加搜索深度之前总的指令数量的平均值为 248,966,990，预留指令的数量为 198,984,837，隐藏指令的数量为 3,463,249。根据表5-4中的平均值行的数据，我们得到增加搜索深度后所测试的总的指令数量为 66,608,938,394，是增加搜索深度之前的 268 倍；预留指令的数量（NRI）为 46,292,624,146，是增加搜索深度之前的 233 倍；隐藏指令数量（NHI）为 5,102,551，是增加搜索深度之前的 1.5 倍。

我们发现，当增加了搜索深度之后，对预留指令空间的覆盖率增加了，但是发现的隐藏指令的数量并没有成正比的增加。隐藏指令在所测试的预留指令中的占比（PHIRI）从增加搜索深度之前的 1.74% 降低到了 0.011%。其根本原因是，当指令空间的搜索深度增加之后，大量的预留指令空间是分布在 0xC4 和 0xC5 所在的指令空间，而这些指令空间发现的隐藏指令较少。因为，在向量指令空间，合法指令的编码只占据了较少的指令空间。

由于现有的方法 [24, 25] 都没有保证对 x86 指令空间的搜索深度，从而它们都错过了对 x86 指令集中部分操作码和操作数的测试。我们提出的基于必要搜索深度实时检测的 DFS 算法保证了对 x86 指令空间的搜索深度。我们的方法保证了对形如 0F xx xx xx、0F 38/3A xx xx xx、C4 xx xx xx xx xx、C5 xx xx xx xx 等指令格式中 xx 字节的测试。相比于现有的方法，我们测试了更多的预留指令空间，主要是预留操作码和预留操作数。测试更多的指令需要更多的测试时间，如表5-4中的最后一列所示。但幸运的是，与此同时，我们也得到了更多的隐藏指令，这些隐藏指令是现有的方法都无法发现的。

5.6 讨论

我将基于必要搜索深度实时检测的 DFS 算法与现有的基于随机的指令空间搜索方法、基于全空间遍历的指令空间搜索方法和基于深度优先遍历的指令空间搜索方法进行对比，主要是为了说明增加搜索深度对提升 x86 指令空间覆盖率的优势。

针对指令操作码和操作数域，采用基于随机的方法不能有效的保证对操作码和操作数的覆盖，同时基于随机的方法还会测试大量的重复的操作码和操作数，此外，还会测试到大量的冗余的立即数和偏移量操作数。而基于必要搜索深度实时检测的 DFS 算法是基于第3章的提升 x86 指令空间搜索效率的基础之上又增加了对 x86 指令空间的搜索深度。采用基于全遍历的方法对 x86 指令空间中的指令操作码和操作数域进行遍历是不可取的一种方法，因为它会遍历到大量的无效指令和冗余的立即数和偏移量操作数，这使得全遍历的方法在可接受

的时间范围内无法完成测试。相比较而言，本章节的方法不仅能保证绕过大量的冗余的立即数和偏移量，更重要的是，它能保证对现有的指令格式中操作码和操作数种类的覆盖率。

与现有的 DFS 算法相比较，例如 Sandsifter 和 UISFuzz，我们的方法最为突出的贡献是增加了对指令格式的分析，从而获得指令的必要搜索深度。通过将指令的搜索深度与必要搜索深度进行对比，从而找到 x86 指令空间搜索深度不足的时候的场景，从而我们的方法能够及时的增加 DFS 算法的搜索深度，保证了对 x86 指令空间中操作码和操作数的覆盖率。

需要注意的是，本章节提出的基于必要搜索深度实时检测的 DFS 算法只适合于对 x86 指令集相关应用的测试，例如处理器、反汇编器、ISA 模拟器和 CPU 模拟器等。不适合于对 RISC 指令集相关应用的测试，因为我们使用的必要搜索深度的值是对 x86 指令格式的分析得到的，然而 RISC 的指令格式与 x86 的指令格式之间存在较大的差异。

5.7　本章小结

本章节首先根据现有方法的指令流分析了对 x86 指令空间搜索深度不足的情况。然后针对 x86 指令空间搜索深度不足的问题，我们提出了基于必要搜索深度实时检测的 DFS 算法。为了实现我们的方法，我们针对现有的指令格式进行了分析和总结，得到了对应的每种指令格式的必要搜索深度；然后依据每种指令格式的必要搜索深度设计算法。我们的算法实时的检测指令空间的搜索深度，当检测到指令长度变化的时候，按照现有的方法 DFS 的搜索深度会增加 1；当增加 1 之后，算法的搜索深度如果仍然小于必要搜索深度值，则继续增加搜索深度。增加搜索深度两种情况：第一种是指令长度小于对应指令格式的必要搜索深度值，则将搜索深度值设置为指令长度；第二种是指令长度大于对应指令格式的必要搜索深度值，则将搜索深度值设置为必要搜索深度。我们将提出的算法在 4 种 x86 处理器平台上进行评估，测试了更多的预留指令空间，同时也发现了更多的隐藏指令。

第 6 章　基于多种反汇编器的交叉检查方法

对隐藏指令的检测需要依据指令集规范（Specification），测试人员在对处理器进行测试的时候都会使用反汇编器（Disassemblers）作为规范，然而使用的指令集规范的准确度直接影响到了隐藏指令的测试精度。现有的方法在测试的过程中使用的反汇编器是 Capstone[95]，但是它对二进制的指令序列进行解码的精度相对较低 [56, 57]。为了提升对 x86 处理器中隐藏指令的测试精度，我们提出了基于多种反汇编器的交叉检查方法。交叉检查的方法依赖于多种反汇编器，当一条指令序列被多种反汇编器同时确定为预留指令的时候，同时此条指令序列能够被处理器执行且不报非法指令异常，则指令为隐藏指令。相比于使用单个反汇编器，我们的交叉检查方法能够一定程度上提升对隐藏指令的测试精度。

本章节我们首先举例说明现有的方法对隐藏指令的搜索精度较低的问题。然后针对这个问题展开阐述了基于多种反汇编器的交叉检查方法。同时我们详细分析了现有的隐藏指令的判决条件，然后根据判决条件我们分析了第3、4、5章节检测出的隐藏指令，并使用我们提出的基于多种反汇编器的交叉检查方法进行重新检查，目的是过滤掉在测试阶段被误判的隐藏指令，提高整体测试的精度。最后，经过对隐藏指令的重新检查，我们分析了现有的各个处理器中存在的隐藏指令类型，并分析了他们的功能和潜在的威胁。

6.1　研究动机

在检测 x86 处理器中存在的隐藏指令的时候，现有的研究方法 [24, 25] 使用反汇编器 Capstone 作为指令集规范，这提升了测试的便利性，但是由于 Capstone 的精度不高，所以影响到了隐藏指令测试结果的精度。在测试隐藏指令的过程中，我们使用差分测试的方法将生成的预留指令分别放入处理器和反汇编器，然后比较处理器和反汇编器的输出结果，如第3章图3-6、4章图4-2、5章图5-2中测试框架所示。对于同一条指令，当处理器的输出结果没有非法指令异常，同时反汇编器输出的结果没有语义的时候（无法识别），那么指令被判定为隐藏指令。

然而，通过对 Sandsifter 测试结果的分析我们发现，有些指令是在指令集中已定义的，但是反汇编器 Capstone 不能识别。反汇编器不能识别指令有多种可能：第一，反汇编器未能及时的升级其指令库，例如，新推出的处理器实现了某指令，但是反汇编器没能及时的跟进实现；第二，反汇编器中的缺陷，由于对指

令集定义理解出现偏差，导致对二进制的解码出现了与指令集手册不一致的情况。对于第一种情况，类似的例子有 0F 01 E8、0F 01 EE、0F 01 EF、0F 38 F9 BF FF 等；对于第二种情况，类似的例子有 0F 0D 00。Capstone 不能对这些指令进行反汇编，但是它们在处理器上都能被执行，经过查找指令集手册 [96, 97]，我们发现它们都已经被定义。

我们需要一种自动化的提升处理器中隐藏指令测试精度的方法。很显然，使用多种反汇编器的交叉检查能够提升对二进制代码的解码精度，但是这种方法同样也面临着问题需要解决。例如，自动化的对比不同的反汇编器的输出结果，需要解决指令格式不统一的问题；所有的反汇编器都可能存在缺陷和不足，如果遇上难以判定的指令，需要手动的对指令进行分析、观察指令的执行行为，并配合查找指令集手册等方法。

6.2 指令语义的差异

围绕指令集规范有多种不同的指令执行的具体应用，例如处理器、CPU 模拟器、反汇编器等，同样的指令在不同的应用中有可能存在语义差异。这些语义差异会对同样的程序在不同的应用中执行的兼容性产生影响，同时影响对程序二进制分析的正确性。如表6-1所示，我们根据指令集应用之间的关系将指令的语义差异分为三大类：第一，执行差异；第二，反汇编差异；第三，执行-反汇编差异。接下来我们对这三种差异进行简单的阐述和分析。

表 6-1 指令语义的差异

Table 6-1 Differences between instruction semantics

语义差异类型	简单描述	子类	执行载体
执行差异	同样的指令在不同的处理器、CPU 模拟器上执行的行为差异	无	处理器、CPU 模拟器
反汇编差异	同样的指令在不同的反汇编器上解码的结果差异	无	反汇编器
执行-反汇编差异	同样的指令分别在处理器（CPU 模拟器）和解码器上执行（解码）的结果差异	可执行但不可解码、可解码但不可执行	处理器、CPU 模拟器、反汇编器

6.2.1　指令语义的执行差异

指令语义的执行差异指的是同样的指令在不同的处理器或者 CPU 模拟器上执行的时候其指令执行的行为之间的差异，其更加关注的是指令执行的时候对处理器或 CPU 模拟器状态的影响，例如对寄存器、程序计数器（Program Counter，PC）、输入输出端口、内存等。

指令语义的执行差异能够被攻击者利用识别特定的处理器平台。因为这种指令在不同的处理器平台上能够产生特定的有差异的行为，攻击者能够通过设计和运行精巧设计的指令序列找到特定的处理器平台，从而发起有针对性的攻击。例如曾经比较出名的震网病毒（Stuxnet）就是有针对性的攻击。这种有针对性的攻击难以被常规的恶意代码检测工具检测到。

6.2.2　指令语义的反汇编差异

指令语义的反汇编差异指的是同样的指令经过不同的反汇编器反汇编之后其输出的结果存在差异，例如指令助记符的差异、寄存器尺寸的差异、指令长度差异等。

反汇编器之间的差异有的被认定是反汇编器中的缺陷，例如现有的工作 [56, 57, 98] 等使用差分测试的方法检测反汇编器中的缺陷。反汇编器中的缺陷一般可以被分为三种：第一，不支持解码（Under Decoding），这种缺陷是反汇编器不能识别或解码有效指令；第二，错误解码（Incorrect Decoding），这种缺陷是反汇编器不能正确的解码合法指令；第三，过度解码（Over Decoding），这种缺陷是反汇编器能够成功的对某一个二进制指令进行解码，但是指令集没有对这条指令进行定义。攻击者可以利用这三种缺陷发起不同的攻击。

第一，不支持解码类型的攻击。在这种情况下，反汇编器不能识别一条合法指令。如果攻击者使用具备这种特点的合法指令发起攻击，当防御者使用反汇编器对攻击代码进行分析的时候，就不能对其中的特定指令反汇编，从而不能理解整个代码的攻击意图。

第二，错误解码类型的攻击。合法指令被反汇编解码的时候，其行为或者结果与指令集手册不一致（与被攻击的处理器的行为不一致）。当攻击者使用这些指令发起攻击的时候，防御者对攻击代码进行反汇编的过程中就会得到一个错误的反汇编结果，从而误解攻击代码的功能。

第三，过度解码类型的攻击。反汇编器将无效指令或者预留指令的二进制编码反汇编为合法指令的语义。攻击者可以用这种类型的指令误导攻击代码的静

态分析结果，使得防御者得到一个与实际的攻击代码执行行为不一致的控制流图。

通过上面的分析，我们发现反汇编器错误的解码和过度的解码都会导致对二进制形式的攻击代码的理解出现偏差。

6.2.3 指令语义的执行-反汇编差异

指令语义的执行-反汇编差异指的是同样的指令分别让处理器（CPU 模拟器）执行与让反汇编器解码，两种载体的输出结果存在的差异。执行-反汇编差异又可以细分为两种：第一，可执行但是不可解码（隐藏指令）；第二，可解码但不可执行。

处理器中存在的隐藏指令可以看作是处理器实现的指令语义与反汇编器之间的执行-反汇编差异，也就是可执行但是不可解码这种类型的指令。隐藏指令在处理器中可以被执行且不报非法指令异常，然而指令反汇编器不能对指令进行解码。将隐藏指令放入恶意代码发起的攻击，其攻击类型与第6.2.2小节中的不支持解码类型对应的攻击相似。攻击者将隐藏指令注入到恶意代码中，使得防御者使用反汇编工具对二进制形式的恶意代码进行分析的时候无法识别其中某些指令的语义。值得注意的是，攻击者只需要利用二进制重写的技术，将目标指令中的部分比特进行改写，就能改变指令的功能。

对于可解码但是不可执行的指令，用它发起攻击的类型与利用第6.2.2小节中的过度解码指令发起的攻击类型相似。用它发起攻击可以迷惑防御者，当他们使用反汇编器对二进制的攻击代码进行分析的时候，能够得到一个完整的控制流图，但是在实际执行的过程中处理器会报异常信号。

6.3 测试结果的判定条件

在检测处理器中的隐藏指令的时候，我们本质上是在检测处理器和反汇编器之间的执行-反汇编差异。在测试反汇编器中的缺陷的工作中，研究人员将处理器看作是一个“黄金模型”，当处理器的执行结果与反汇编器的输出不一致的时候，一般会认为是反汇编器中存在缺陷。而在测试处理器中的隐藏指令的时候，我们不认为处理器是一个完全可信的黑盒子，当处理器的执行结果与反汇编器的解码结果出现差异的时候，我们会判断结果到底是处理器中的隐藏指令还是反汇编器中的缺陷。

在测试阶段，我们使用反汇编器 Capstone 的反汇编结果与处理器的执行结

果进行对比，如果一条指令不能被 Capstone 解码，但是能够在处理器中正常执行且不报非法指令异常，那么初步认为这是一条隐藏指令。然而，在第6.2小节我们已经讨论过，当一条指令能被处理器成功执行但是不能被反汇编器成功解码的时候，这个指令有可能是隐藏指令，还有可能是反汇编器的不支持，如表6-2中第 3 行所示。除此之外，在表6-2中我们还总结了现有的判决指令测试结果的条件。当一条指令不能被处理器执行，同时也不能被反汇编器解码的时候，这条指令为无效指令或者是预留指令。当一条指令不能被处理器执行，但是可以被反汇编器解码的时候，这条指令的判决结果有两种可能：第一，处理器没有实现这个指令，例如处理器比较老旧；第二，反汇编器中存在缺陷，错误的将无效指令或者预留指令的二进制编码解码成了合法指令。当一条指令能够被处理器执行的同时能够被反汇编器解码的时候，这条指令一般情况下是一条合法指令；但如果在比较了指令的语义或者长度等信息之后，发现反汇编器输出的信息（例如指令长度）与处理器的输出不一致，那么这条指令就是一条反汇编器的指令缺陷。

表 6-2　测试结果的判决条件

Table 6-2　Conditions for the determination of test results

序号	处理器	反汇编器	可能一	可能二
1	*X*	*X*	无效指令	预留指令
2	*X*	✓	可解码但不可执行（第6.2.3小节）	过度解码（第6.2.2小节）
3	✓	*X*	隐藏指令（第6.2.3小节）	不支持解码（第6.2.2小节）
4	✓	✓	合法指令	错误解码（第6.2.2小节）

注：“*X*”**表示指令不能被执行或者反汇编**，“✓”**表示指令能够被执行或者反汇编**。

我们将表6-2中所示的 4 种判决条件及其对应的可能情况当做判定所检测指令的具体类型的依据。当然，真正决定一条指令是否为隐藏指令，还需要借助于多种反汇编器，甚至还需要借助于指令集手册。在测试阶段只使用一种反汇编器势必会使得测试结果的误报率较高，我们接下来讨论基于多种反汇编器的交叉检查方法来对检测出的结果进行重新的检查，目的就是为了降低检测结果的误报率。

6.4 基于多种反汇编器的交叉检查方法

理想的反汇编器对一条指令的反汇编结果应该与指令集规范中定义的指令行为一致。反汇编器在对二进制代码进行分析的过程中是必不可少的工具，在对处理器中的隐藏指令的检测过程中也有较多的应用。反汇编器能够将二进制的指令反汇编为能表示其功能的助记符、寄存器、立即数和偏移量等信息，同时还能得到指令的长度信息。然而，反汇编器在对二进制的指令进行解码的时候，会产生误判，例如第6.2小节描述的不支持解码、错误解码和过度解码等情况。很显然，使用多种反汇编器对同一条指令进行解码，然后对结果进行分析，能够一定程度上减少对其解码结果的误判。

基于多种反汇编器的交叉检查方法是对差分测试思想的应用和拓展。一般而言，差分测试的方法用在比较两种应用的输出之间的差异，从而判断所测试的应用或者设计是否符合预期（设计标准或者“黄金模型”）。当只使用一种反汇编器与处理器的执行结果进行对比的时候，就是常见的差分测试结构，一般用来检测反汇编器中的缺陷，或者检测处理器中的隐藏指令。图6-1所示的是我们基于多种反汇编器的交叉检查方法的测试框架。对于给定的指令序列，我们将其分别放入多个反汇编器中进行反汇编（D_1、D_2、D_3、D_4），同时将指令放入处理器（D_{CPU}）中执行，然后比较它们的结果（r_1、r_2、r_3、r_4、r_{cpu}）。因为并不是所有的反汇编器都能够提供精确分析二进制指令格式的调用接口（API），我们统一采用助记符类型的汇编语言表示指令语义。理想情况下，对于同一个二进制形式的指令，所有的反汇编器应该有同样的语义输出，但是在实际使用的过程中，它们的汇编语言格式或者内容会有微小的差异。所以，图6-1中我们对输出结果进行了规范化，例如将 AT&T 格式的汇编语言与 Intel 格式统一输出为 Intel 格式。本论文中，我们更加关注的是二进制形式的指令能否被反汇编器识别，反汇编器对于不能识别的指令会返回“Invalid Instruction”或者“Unknown”等类似的信息，我们将这些信息统一规范输出为预留指令。如果二进制指令能够被反汇编器识别并输出对应的汇编指令，则我们的方法认为指令都是合法的，将指令统一规范化输出为合法指令；同时，我们的方法还比较指令的汇编语言，如果汇编语言之间存在差异，则使用一致性系数对其进行标记，一般的，汇编语言之间的差异是某个反汇编器中的缺陷。我们将反汇编器的输出结果进行规范化之后对结果进行评估。

将规范化的输出结果进行评估，目的是为了找到可信的反汇编器的结果，然后结合处理器的执行结果判断指令的类型。评估结果需要将不同的反汇编器的

结果进行对比和分组，并计算其一致性系数。我们用一组集合表示各个反汇编器的规范化输出的结果：$R=\{r_1, r_2, ..., r_{n-1}\}$，其中 r_i 对应反汇编器 D_i 的输出，同时我们定义 r_p 是处理器（D_p）的输出结果。我们定义 $\bar{R} \in R$ 为有同样类型输出结果的反汇编器集合，然后给出其中元素对应的一致性系数（Coefficient of Agreement），$c(r_i)=|[r_i]|/|\bar{R}|$，其中 $c(r_i) \in [0,1]$，它表示输出 r_i 正确性的占比，$\bar{R}$ 中一致性系数较高的反汇编器的结果被认为是可信的。如果反汇编器的输出结果 $r_j \in R$，但是 $r_j \notin \bar{R}$，则其对应的一致性系数为 $c(r_j)=0$。然后，我们将 $\bar{R}$ 与 r_p 进行比较，从而判定可疑指令的类型。

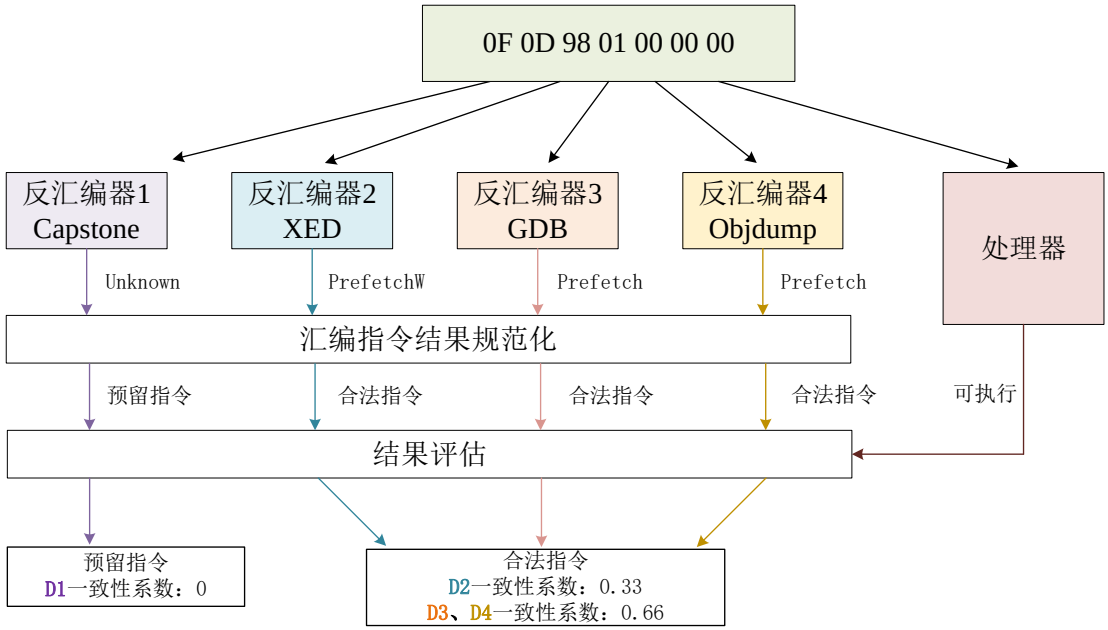

图 6-1　基于多种反汇编器的交叉检查方法的测试框架

Figure 6-1　The test framework for cross-checking methods based on multiple disassemblers

如图6-1所示，我们展示了指令 0F 0D 98 01 00 00 00 的交叉检查过程。图中我们使用了四种反汇编器 Capstone、XED、GDB 和 Objdump，其中 Capstone 不能对指令进行识别，其输出为 “Unknown”，我们将其规范化输出为预留指令；XED 能对指令进行反汇编，结果为 PrefetchW，为指令集中的预取指令；GDB 和 Objdump 的输出都为 Prefetch，也为预取指令；我们统一将 XED、GDB 和 Objdump 的结果标记为合法指令，并将它们放入 $\bar{R}$，同时用一致性系数来表示指令助记符的差异。将反汇编器的结果进行分组，由于 r_1 的输出结果与其他的三个反汇编器都不一样，它不被放入 $\bar{R}$，所以它的一致性系数为 0；反汇编器 D_2、D_3 和 D_4 的规范化输出为合法指令，则有 $\bar{R}=\{r_2, r_3, r_4\}$，其中 r_2 的一致性系数为 0.33，r_3 和 r_4 的一致性系数为 0.66。一致性系数越高，我们的方法认为其可信度越高。

所以，此条指令被反汇编器的组合判定为合法指令 Prefetch。处理器输出的信息有指令的长度、指令是否可被执行、指令执行过程中报的异常等。根据处理器对指令的执行结果，我们得知指令可被正常执行且不报非法指令异常，从而我们根据一致性系数和处理器的输出结果判断指令 0F 0D 98 01 00 00 00 为合法指令 Prefetch。如果只参考 Capstone 的反汇编结果，那么指令 0F 0D 98 01 00 00 00 会被判定为隐藏指令，经过上面的分析，我们证实了它是一条合法指令。

6.5 实验评估

我们在 Intel 的 Xeon W3175X 处理器上对我们提出的方法进行了评估。其具体的实验配置如表6-3所示。我们将第3章、第4章和第5章中发现的隐藏指令都使用基于多种反汇编器的交叉检查方法进行了重新的检查，过滤掉了大量的合法指令，提升了整体的测试精度。

表 6-3 评估交叉检测方法所需要的实验环境

Table 6-3 The experimental environment required to evaluate the method of cross-checking mechanism

类型	详细信息	版本
处理器	Xeon-W3175X	微架构 Skylake
操作系统	Ubuntu 16.04	内核 4.15.0
内存	192G	DDR4 2666
Capstone[95]	开源	4.0.2
XED[99]	Intel 官方	12.0.1
GDB[100]	GDB	8.1.1
Objdump[101]	GNU	2.30

6.5.1 度量指标

本节的度量标准是误报率（False Positive Rate, FPR），我们定义误报率是被错误认定为隐藏指令的数量与总共发现的隐藏指令数量的比值：

$$FPR = \frac{Number\ of\ \ false\ reported\ instructions}{Number\ of\ total\ reported\ instructions} \times 100\% \tag{6-1}$$

6.5.2　测试精度的提升

我们将第3章发现的隐藏指令使用基于多种反汇编器的交叉检查方法进行重新检查，平均将隐藏指令的数量从 3,463,249 降低到了 1,150,733，如表6-4所示。当设置指令前缀长度为 1 的时候，通过误报率的度量指标公式（6-1），我们可以计算得到平均的误报率降低了 66.77%。

表 6-4　交叉检查方法实施前后的隐藏指令数量对比（前缀长度为 1）

Table 6-4　Comparison of the number of hidden instructions before and after the implementation of cross-checking method

处理器	交叉检查之前的隐藏指令数量	交叉检查过滤掉的合法指令数量	交叉检查过后的隐藏指令数量	降低误报率
i5-11600KF	3,351,847	2,083,122	1,268,725	62.15%
i7-10700KF	3,347,748	2,083,122	1,264,626	62.22%
Gold 5218	3,045,384	2,696,386	348,998	88.54%
i5-7300HQ	3,961,284	2,696,658	1,264,626	68.08%
i7-6700	3,347,748	2,083,122	1,264,626	62.22%
Zen3-5600X	3,346,148	2,081,394	1,264,754	62.20%
Zen2-3700X	3,346,148	2,081,394	1,264,754	62.20%
Zen-1800X	3,959,684	2,694,930	1,264,754	68.06%
平均值	3,463,249	2,312,516	1,150,733	66.77%

如表6-5所示，当指令前缀长度为 4 的时候，使用基于多种反汇编器的交叉检查的方法将隐藏指令的误报率平均降低了 95.55%。可见当指令前缀长度为 4 的时候其误报率要比指令长度为 1 的时候要大，其背后的原因是，随着指令前缀长度的增加，反汇编器 Capstone 不支持解码类型的指令会增加。有必要仔细审计 Capstone 的代码，从而增加其对二进制指令反汇编的可信度，尤其是当指令前缀长度增加的时候。

6.5.3　隐藏指令的类型

经过对检测出来的隐藏指令的交叉检查，隐藏指令的种类可以确定的是 0F 0D、0F 18、0F 20、0F 22、C4Y37D 等类型[1]。在 Intel 和 AMD 中都存在的指令

[1]"Y" 可以是 {0,2,4,6,8,A,C,E}。

表 6-5　交叉检查方法实施前后的隐藏指令数量对比（前缀长度为 4）

Table 6-5　The number of instructions tested by Skipscan, when prefix length is four

处理器	交叉检查之前的隐藏指令数量	交叉检查过滤掉的合法指令数量	交叉检查过后的隐藏指令数量	误报率降低
i5-11600KF	1,208,843,721	1146357317	62,486,404	94.83%
i7-10700KF	1,248,932,821	1186486180	62,446,641	95%
Gold 5218	1,285,688,183	1269289463	16,398,720	98.72%
i5-7300HQ	1,292,718,193	1229911411	62,806,782	95.14%
i7-6700	1,248,247,661	1185338783	62,908,878	94.96%
Zen3-5600X	1,281,276,849	1217515038	63,761,811	95.02%
Zen2-3700X	1,252,170,755	1188457542	63,713,213	94.91%
Zen-1800X	1,471,817,097	1408303499	63,513,598	95.68%
平均值	1,286,211,910	1,228,957,404	57,254,506	95.55%

类型有 0F 0D 和 0F 18，只有 AMD 处理器中存在的指令类型有 0F 20、0F 22 和 C4Y37D。

经过分析，我们发现这些指令的功能与其临近的合法指令的功能相似，我们将其称为合法指令的衍生品。就是在设计和实现处理器的过程中有些预留指令编码位没有得到较好的优化，其具备了某种合法指令类似的功能。

表 6-6　在 x86 处理器中发现的隐藏指令的操作码

Table 6-6　The opcodes of hidden instructions on different processors

指令	i5-11600KF	i7-10700KF	Gold 5218	i5-7300HQ	i7-6700	Zen3	Zen 2	Zen
0F0D	✓	✓	✓	✓	✓	✓	✓	✓
0F18	✓	✓	✓	✓	✓	✓	✓	✓
0F{20, 22}	*X*	*X*	*X*	*X*	*X*	✓	✓	✓
C4Y37D	*X*	*X*	*X*	*X*	*X*	✓	✓	✓

注：表中指令类型 C4Y37D 的“Y”可以是 {0,2,4,6,8,A,C,E}。“*X*”表示指令不能在对应的处理器中被执行，“✓”表示指令能够在对应的处理器中被执行。

6.6　讨论

诚然，在指令测试阶段直接使用基于多种反汇编器的交叉检查的方法能够更加直接的增加对隐藏指令的检测精度。然而，与现有的方法一样，我们在测试阶段也使用了反汇编器 Capstone[95]，其原因有两个：第一，在测试阶段使用与现有的方法一致的反汇编器能够最大限度的减少实验条件的变量，使得我们的方法与现有的方法开展效率和覆盖率的横向对比更加的准确；第二，Capstone 有着较好的效率和可兼容性，例如可以兼容 Python 编程语言的函数，从而使得结果能较好的以图形界面的形式输出。

任何反汇编器都可能存在缺陷，通过多种反汇编器对同样的指令进行反汇编，会增加其结果的可信度。但是，也不排除所有的反汇编器都同时出现错误的情况，虽然这种情况出现的概率比较低。我们在第6.4小节使用的一致性系数，是为了衡量多种反汇编器之间一致性的指标。一般而言，这个指标为我们提供了反汇编器可信度的一种参考。当然，也不排除一致性系数较高的反汇编器出现错误的情况，这就要具体问题具体分析，对特殊的指令进行“人工介入”，需要对指令功能进行单独的测试和查询指令集手册，从而在证实指令功能和语义之后再判定其是否为隐藏指令。

需要注意的是，本章节提出的基于多种反汇编器的检查方法也适用于对 RSIC 处理器中隐藏指令或者指令缺陷的测试，可以提升检测的精度。

6.6.1　隐藏指令的威胁

隐藏指令的威胁可以分为两类：第一，直接影响处理器的执行行为，干扰处理器处理信息的可信性；第二，注入恶意代码中（包括侧信道攻击代码），替换有同样功能的合法指令，使得防御人员在分析二进制的恶意代码的时候难以理解其真实的攻击意图。

对于第一种威胁类型的隐藏指令，其发现难度较大，而且发现之后需要对其功能进行详尽的分析，才能找到较好的攻击场景。而对于第二种威胁，根据现有的隐藏指令，其可实施性较高，如果现有的恶意代码中有与现有的隐藏指令功能类似的指令，则可以用隐藏指令替换，如使用二进制重写技术将隐藏指令放入某代码中。例如，使用预取指令（Prefetch）能够发起侧信道攻击，能够使得低优先级的攻击者获得较高优先级的内存地址信息。假如恶意代码检测工具能够检测类似的攻击类型的代码和指令，如果将预取指令替换成具备预取功能的隐藏指令，那么就可以绕过恶意代码攻击检测工具的检测，从而使得预取侧信道攻击

（Prefetch Side-Channel Attack）[102] 仍然有效。使用有同样功能的隐藏指令替换合法指令之后，恶意代码的功能不变，但是如果防御者通过反汇编器对恶意代码进行二进制分析，就会难以理解其真实的攻击意图。

另外，有现有的工作 [18, 29, 103] 研究合法指令的组合所能构成的隐蔽信道（Covert Side-Channel），从而能够构建攻击者窃取传输私密信息的通道。我们在下一个章节探索类似的攻击类型。

6.7 本章小结

本章节我们针对处理器中隐藏指令搜索精度不高的问题提出了基于多种反汇编器的交叉检查方法。我们首先辩证的讨论了同样的指令在不同的指令集实现载体之间的语义差异的类型，总结出指令类型的判定规则。然后我们设计了基于多种反汇编器的交叉检查方法的检测框架，并依据指令的判定规则对指令进行分类；检测框架能够对第3章、第4章和第5章中发现的隐藏指令进行系统的检查，然后筛选掉合法指令，从而提升我们方法的检测精度。经过实验评估，结果证实了我们的方法能够将现有工作 Sandsifter 测试的误报率降低 66.77%（指令前缀长度为 1）和 95.55%（指令前缀长度为 4），证明了我们方法的有效性。

第 7 章　基于指令操作码分类的时间侧信道检测方法

计算机系统中的侧信道攻击能在窃取计算机中的私密信息的同时较难被发现。为了提升处理器的性能，处理器研发人员不断的对微架构进行优化，例如为了实现指令级并行 (Instruction Level Parallelism, ILP)，推出了多发射、乱序执行和分支预测等机制，为了实现线程级并行 (Thread Level Parallelism, TLP)，推出了超线程和多核心处理器等机制。与此同时，为了使开发者更好的理解这些机制，处理器研发人员将处理器微架构模型进行抽象，发布了处理器的数据通路架构和控制逻辑图示，这也为侧信道攻击者提供了便利。一般的，攻击者可以通过分析微架构中的数据通路和控制逻辑对应的处理器计算资源模块，并利用资源模块的功能特点巧妙的设计侧信道攻击代码，同时防御者也通过观察微架构和分析攻击者的代码提出对应的防御策略。但是，现存的处理器有多种微架构设计，通过分析微架构中计算资源模块的方式发现微架构脆弱性的方法效率较低，从而也使得防御机制的提出更加滞后，延长了计算机系统遭受安全威胁的时间窗口。幸运的是，所有的微架构功能都能在指令层得到抽象和体现，我们可以在指令层面通过执行不同的指令序列，并检测处理器状态的变化，从而分析所测试的指令序列是否构成侧信道攻击的脆弱性风险。这种方法能够通过自动化的测试方法实现，且在不同的微架构处理器上兼容，因为不同的 Intel 和 AMD 微架构（Micro-architecture）遵循同样的 x86 指令集架构（Instruction Set Architecture, ISA）。

本章节我们主要研究自动化的检测处理器中基于时间的侧信道的方法。我们首先分析了现有的侧信道的检测方法的优缺点。然后，我们提出了基于指令操作码分类的检测方法，将使用同样的计算资源的指令进行分类，然后加强同类指令的交叉测试，目的是为了在指令层面尽可能的发现处理器微架构中可能存在的时间侧信道，其中也包括隐藏指令可能触发的隐蔽信道。最后，我们将我们提出的方法在 4 种来自 Intel 和 AMD 的 x86 处理器上进行了评估，并发现了一些指令序列触发的时间侧信道，同时我们与现有的工作进行了比较和讨论。

7.1　研究动机

检测特定类型的微架构脆弱性，需要基于微架构脆弱性相关的攻击模型（威胁模型）构建检测框架。例如，Transynther[65] 是针对 Meltdown 类型的瞬态执

行攻击而构建的测试框架，经过分析现有的 Meltdown 类型攻击的代码，总结出代码对应的攻击模型，然后替换代码中的一些指令，从而发现新的 Meltdown 类型的攻击代码。ABSynthe[19] 是针对基于竞争的侧信道 (Contention-based Side Channel) 类型，目的是为了发现特定的加密软件在某硬件上运行的时候对应的侧信道风险（同样的软件在不同的处理器上运行可能面临不同的侧信道风险）。Covert Shotgun[18] 是针对同时多线程中（Simultaneous Multi-threading, SMT）存在的隐蔽信道 (主要针对的是 Time-based Side Channel，检测不同指令执行时间延迟的变化)，但是它只遍历测量了 12 种指令类型，对 x86 指令集中的指令类型的覆盖率较低。Osiris[29] 相比于 Covert Shotgun 使用了更多的 x86 指令，但是它对指令组合的测试效率不高。

本章节主要针对处理器中的共享资源引起的时间侧信道问题，研究自动化的检测它们的方法。指令执行最终是要依赖对应的计算模块，同时也会使用对应的共享资源。如果某条被测试的指令在执行时候正巧碰上所需要的计算模块或者共享的数据通路被其他的指令所占用的情况，那么此条被测试的指令的等待时间就会增加。所以，在指令层面，通过观察某条被测试指令执行的等待时间是否会被另一条指令调制（干扰），是检测所测试的指令组合是否具备构建侧信道风险的比较直接的一种方法。从指令集中的指令出发，展开自动化的测试处理器中微架构侧信道风险的方法，能够避免分析复杂的、种类多样的处理器微架构（Intel 和 AMD 目前总共有十几种以上的微架构，但都遵循 x86 指令集），同时也能够避免为了理解处理器微架构的功能而进行大量的反向工程工作。

然而，现有的基于单条指令执行时间的变化检测处理器中的侧信道风险的工作 [18, 29] 对 x86 指令集中指令的覆盖率较低，而且对指令组合的测试效率较低。我们需要一种方法能尽可能的覆盖所有的 x86 指令空间中的具备功能的指令（包括合法指令和隐藏指令），同时在测试指令组合所能引起的侧信道的时候，能保证对指令组合的测试有较高的效率。

7.2 侧信道

任何侧信道同时也都可以被利用成隐蔽信道 [29]，隐蔽信道强调的是其行为更难被发现。可信计算系统评估标准（Trusted Computer System Evaluation Criteria, TCSEC）[104] 将隐蔽通道（Covert Channel）定义为任何可以被进程利用以违反系统安全策略的方式传输信息的通信通道。时间侧信道是允许一个进程通过调制它自己对系统资源的使用来向另一个进程发送信息的通道，这个通道使得第

二个进程通过观察到的对系统资源响应时间的变化获得所传输的信息（通常一次传输一个比特，例如，响应时间大于某个阈值为 1，小于某个阈值为 0）。

通常侧信道传输信息的传输率与使用的共享资源的基本访问延迟有关系，内存的访问延迟要大于高速缓存（Cache），那么基于高速缓存的侧信道的传输率就要比基于内存的侧信道要快。基于内存的侧信道的传输率为比特每秒的数量级，例如 Okamura 等 [105] 构建的基于内存的侧信道带宽为 0.49 比特每秒；Ristenpart 等 [106] 构建的基于内存的侧信道带宽为 0.2 比特每秒。而基于 Cache 的侧信道的传输率为千比特每秒甚至兆比特每秒的数量级，例如针对三级缓存 [107] 的 (Last Level Cache, LLC) 侧信道传输率能超过 1 兆比特每秒。TCSEC 指出侧信道的带宽超过 100 比特每秒就可以被认为是高带宽的信道，如果带宽小于 0.1 比特每秒则被认为不是有效的侧信道，并不是说带宽低不能构成威胁，而是较低的带宽对于敌手窃取信息的代价和困难都是较大的。这为我们自动化的检测处理器中的基于时间的侧信道提供了参考：如果某指令的等待时间过于长，那么就会导致使用此条指令构建的侧信道攻击带宽较低；如果某指令的等待时间过于短，则会导致难以形成稳定的侧信道攻击。通过对现有的使用指令执行时间差异发起攻击的分析，我们发现一般指令的差异时间在 70 到 250 个时钟周期。为了囊括尽可能多的时间侧信道，我们的方法目前关注了指令执行时间差异在 50-300 个时钟周期之内的指令。另外，我们的方法目前只是测试了每次触发条件和检测条件都是单条指令的情况。

7.3　攻击模型

本论文中我们重点关注的是时间侧信道的检测。如图7-1所示，我们的攻击模型是 Trojan 已经得到了私密信息，而且要将此私密信息通过隐蔽信道传输给 Spy。Trojan 和 Spy 约定好使用的共享资源（本章节对应使用特定的指令），而且在传送私密信息之前会先传输一个特定的握手信号，例如先发送 8 个比特的 10101010，以确保隐蔽信道可用。

基于共享资源的时间侧信道会受到噪声的干扰，例如 Trojan 和 Spy 使用的共享资源，在传输隐秘信息的时候被其他进程占用。在本论文对共享资源引起的时间隐蔽信道检测的过程中会将测试的程序绑定到特定的处理器核心上，从而使测试程序避免受到噪声的干扰。

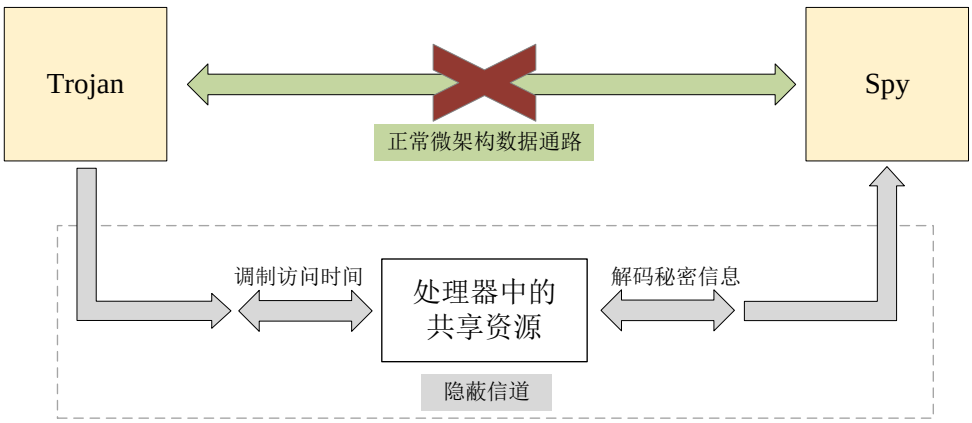

图 7-1　基于共享资源的时间侧信道攻击模型

Figure 7-1　Timing-based side channel attack model based on shared resource

7.4　基于指令操作码分类的时间侧信道检测方法

为了提升测试方法的效率和覆盖率，我们提出了基于指令操作码分类的时间侧信道检测方法。为了实现对 x86 指令空间的高覆盖率，我们对指令空间进行深度优先遍历，然后筛选出具备功能的指令，其中包括合法指令和隐藏指令。为了提高对指令组合的测试效率，我们将具备功能的指令通过反汇编工具进行反汇编，并根据指令的操作码进行分类，隐藏指令单独归为一类；从而增加同类指令组合执行的过程中引起共享资源争用的概率，提高检测时间侧信道的效率。

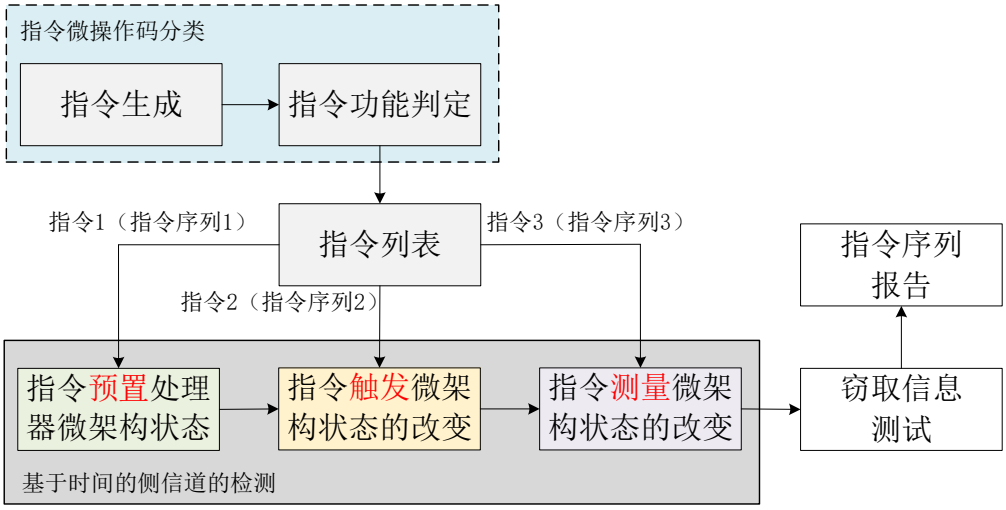

图 7-2　检测时间侧信道的测试框架

Figure 7-2　Test framework for detecting timing-based side channels

图7-2是我们提出的基于指令操作码分类的时间侧信道检测方法的测试框架，我们将我们的方法命名为 TSCMiner（Time-based Side Channel Miner）。TSCMiner

是为了实现自动化的发现新的基于时间的侧信道，其测试框架分为两个过程：第一，对指令列表的预处理；第二，对指令的组合和测试。第一个过程是为了通过遍历指令空间，获得尽可能全的合法指令和隐藏指令列表，并将指令列表中的指令进行分类。第二个过程是为了优先将同类的指令进行测试，发现能够构建时间侧信道的指令组合。对指令测试的过程分为三个阶段：预置处理器微架构的状态、触发微架构状态以及测量微架构状态的改变。然后检测这三个指令序列是否能够形成一个侧信道，如果检测出来指令组合对应的侧信道能够窃取到预置的私密信息，则将指令组合放入测试报告文件。接下来，我们将各个模块的功能进行详细的描述。

- 指令生成：通过深度优先遍历算法对 x86 指令空间遍历，主要是对前缀、操作码、ModR/M、SIB 域进行遍历，尽可能全面的覆盖所测试处理器中的具备功能的指令类型，包括合法指令和隐藏指令。
- 指令功能判定：将生成的指令在所要测试的处理器上执行并筛选出具备功能的指令。注意，这一步比较重要，因为不同的处理器中实现的指令会存在差异，包括合法指令和隐藏指令类型，通过对指令的执行，我们能够判断指令的行为，过滤掉报异常的指令。将筛选出的指令用通过反汇编器识别指令的操作码（助记符），将具备同样类型助记符的指令进行分类。如果指令的助记符一致，而寄存器类型和操作码类型不一致，则这些指令具备同样功能。需要注意的是，我们将改变程序控制流的指令排除在外，例如 JMP、RET、CALL 等类型，因为这些指令可能会使程序异常退出。
- 指令列表：将分类后的指令放入指令列表文件。
- 指令预置处理器微架构状态：通过指令的执行将微架构的状态预置到一个初始的已知的的状态。例如清除或者驱逐 Cache 的行（Flush+Reload 类型侧信道）、或者关掉 AVX2 单元、或者预置随机数生成器为未使用的状态。一般的，对于同样的共享资源，在测试之前其预置状态的指令可以是一样的。
- 指令触发微架构状态的改变：通过指令的执行将第一步预置的状态改变，对微架构状态的改变是为了调制（干扰）后续的测量指令使用的共享资源的访问时间。例如，内存 Cache 行访问指令、或者执行一条 AVX2 指令、或者执行一条 RdSeed 指令。注意，在触发指令执行之前，测试指令会先得到一个指令执行所需要的时间 T_0。
- 指令测量微架构状态的改变：在指令已经触发微架构状态改变的基础之上，通过指令的执行测量再次访问同样的资源所需要的时间 T_1，如果访问时间增加，即

$T_0 < T_1$，且时间差异在 50-300 个时钟周期之内，则具备构成隐蔽信道的条件。例如测量再次访问同样 Cache 行的时间，如果时间变长且超过设定的预置则可作为传输 1 的信道；再次执行 AVX2 类型的指令，则指令等待的时间增加。注意，我们将所测试的程序绑定在特定的处理器核上，目的是为了避免其他程序的运行对共享资源占用带来的噪声干扰。

- 窃取信息测试：将获得的指令组合对应的侧信道进行真实的攻击场景的测试。使用图7-1中的攻击模型窃取私密信息，如果所发现的侧信道能够将信息稳定的传送出去，且带宽大于 0.1 比特每秒，则认为构成了有效的侧信道攻击，将指令组合存入报告文件。
- 指令序列报告：将确实能稳定的实现窃取预置的私密信息的指令序列放入结果输出文件。然后对结果进行分类，如果多种指令组合中有同样的预置指令或者触发指令，则将这些指令序列归结为同样的侧信道类型。

```
Trigger:
    <trigger_instruction>
    <trigger_instruction>
    <trigger_instruction>
    <trigger_instruction>
    <trigger_instruction>
    <trigger_instruction>
Jmp Trigger:
```

Listing 1 触发微架构状态改变的指令执行方式

```
Cpuid
Mfence
Rdtscp
Mov rdi, rax
<measure_instruction>
Rdtscp
Sub rax, rdi
Ret
```

Listing 2 测量微架构状态改变的指令执行环境

以检测 SMT 中存在的执行端口共享的时间侧信道的检测为例，TSCMiner 需要不断的执行触发指令，以达到占用指令对应的执行端口的目的，如代码1中所示为触发微架构状态改变的指令执行例子，使用循环的方式不断的执行触发指令。相对应的，测量指令需要借助 Rdtscp 指令来计算访问共享端口的时间。如代码2所示，第 3 行为第一个 Rdtscp 指令，是为了获得测量指令执行之前的时间戳，默认的时间戳的值放置在 rdi 寄存器中，第 4 行的 Mov 指令将其放到 rax 寄存器中；第 6 行为第二个 Rdtscp 指令，获得的第二个时间戳的值被放置到了 rdi

寄存器中。第 7 行通过减法指令 Sub 计算寄存器 rax 和 rdi 之间的差值，就得到了测量指令的执行时间。

7.5　实验评估

我们将 TSCMiner 在如表7-1中所示的 4 种 x86 处理器上进行测试评估。处理器运行的操作系统为 Ubuntu 18.04 内核版本为 4.15.0，处理器运行在 64-bit 的长模式下。

表 7-1　评估检测时间侧信道脆弱性所用的处理器

Table 7-1　The processors used to evaluate the detection of timing-based side channel vulnerabilities

处理器	微架构类型	发布时间
i5-11600KF	Rocket Lake	March 16, 2021
i7-6700	Sky Lake	September 27, 2015
Ryzen-5600X	Zen 3	November 5, 2020
Ryzen-3700X	Zen 2	July 7, 2019

7.5.1　实验结果

在对表7-1中 4 种处理器上的测试测试了将近一个月的时间，经过对实验结果的分析和分类，TSCMiner 不仅发现了已有的攻击类型，例如 Flush+Reload，而且还发现了新的攻击类型，例如 AVX2 以及 RdRand 等指令，如表7-2中所示。从发现的侧信道攻击类型的结果来看，证明了我们在指令层面自动化的检测处理器微架构侧信道脆弱性的有效性。

值得注意的是能够形成 Flush+Reload 类型的攻击的指令组合有超过 2 万多种，但由于都使用了例如 CLFLUSH 和 CLFLUSHOPT 等类型的预置指令，而且后续的触发和测量指令类型都是内存加载之类的指令类型，所以，在分类的过程中，我们依据预置指令的类型将其归结为一类。在瞬态攻击类型中，例如 Meltdown，Flush+Reload 类型的侧信道能够将微架构状态的信息泄露到攻击者可见的处理器架构层面。

对于 AVX2 指令类型形成的侧信道攻击，TSCMiner 发现了将近 600 种指令组合。向量指令的延迟要比整数计算指令的大，如果有大量的向量指令同时发

射，则后续指令的等待时间会增加，从而较容易形成可被利用的时间侧信道。

表 7-2　TSCMiner 发现的时间侧信道

Table 7-2　Timing-based side channel discovered by TSCMiner

类型	预置指令	触发指令	测量指令	时间差异
Flush+Reload	CLFLUSH	Mov al, [rcx]	Mov al, [rxc]	186 cycles
RdRand	Sleep	RdRand	RdRand	210 cycles
RdSeed	Sleep	RdSeed	RdSeed	207 cycles
MMX	FLDLN2	PHADD MM2, [R9]	PHADD MM1, [R8]	88 cycles
AVX2	FISTP [R9]	VDMADD1 YMM2, YMM3, [R9]	VDMADD YMM2, YMM3, [R9]	157 cycles
XSAVE	LAR ECX, EDX	XSAVE [R9]	XSAVE [R9]	152 cycles

RdRand 和 RdSeed 类型的指令构建的侧信道。由于这两个指令调用的是核间共享的资源，所以，它们能够被用来发起核间交叉的隐蔽信道（Cross-core Covert Channel）。

XSAVE 类型是在指令组合中含有 XSAVE 和 XSAVE64 这两种类型的指令。在这种情况下，预置指令可以是任何的指令。因为，它们是对浮点数计算单元（Floating-point Unit，FPU）中资源的调用。同样的，MMX 类型的指令也是调用 FPU 中的计算资源。

综上所述，指令对核间共享计算资源和核内共享计算资源的调用，如果指令本身执行所需要的时钟周期就较长，则容易引起后续的需要同样计算资源的指令发生等待，从而更容易引发可被利用的侧信道脆弱性风险。相比较而言，对于整型计算资源，由于现有的处理器对并行计算的优化，会增加程序中占比较高的指令对应的计算资源，例如 Add、Sub、Load、Store 等指令类型，从而使得这些常用的指令的等待时间较短，一般都在几个到几十个时钟周期，这小于我们预定的 50-300 个时钟周期。

7.6　讨论

前面章节研究的提升 x86 处理器中隐藏指令测试覆盖率的方法，使得我们能够获得更多类型的隐藏指令。这也为本章节测试隐藏指令与合法指令组成的指令序列所触发的基于时间的侧信道攻击提供了一个基础的隐藏指令库。我们

对指令进行分组是根据操作码的功能展开的，一般而言，隐藏指令的功能与其临近的合法指令的功能比较接近，所以，我们将每种类型的隐藏指令划分到与其临近的合法指令的分组当中。

从指令层面出发，结合现有的攻击模型，构建自动化的检测方法无疑是一种发现微架构脆弱性的较好的方法。对于处理器厂商公布的微架构图示中没有标识的计算资源，通常研究人员需要通过反向工程的方法发现其中存在的可被利用发起侧信道攻击的功能。但从反向工程本身而言，就需要研究人员对微架构的理解和对应的软件编程技能有较高的水平。当然，针对不同的攻击模型需要单独构建不同的测试框架也需要花费一定的时间来分析和总结威胁模型。

不同的处理器微架构支持的侧信道会存在区别，同样的指令在不同的处理器中表现的侧信道风险也会有偏差。那些执行端口中计算资源比较单一的指令更加容易出现拥堵和等待。支持更好的并行执行的处理器，其中可能存在的隐蔽信道就越少。所以，同样的侧信道攻击代码，在不同的处理器中表现的性能和攻击效果会有区别。这更证明研究兼容性较好的测试处理器中微架构脆弱性方法的重要性。

处理器相关的共享资源可以分为核心内部共享和核心之间共享，例如对于 ALU 中的执行端口属于核心内部共享的资源，而对于随机数生成器属于核心之间共享的资源。Covert Shotgun 的出发点是为了检测由于 SMT 共享计算资源引起的隐蔽信道，直观的来看它是属于对核心内部的共享资源的检测。Covert Shotgun 目前不支持对内存子系统的测试，而且，作者排除了对寄存器的使用，因为在 SMT 中两个线程所使用的寄存器是独立的。同时，Covert Shotgun 只能发现与指令执行时间相关的隐蔽信道，受到威胁模型的限制，它无法通过测量指令的执行时间发现例如 Flush+Reload 和 Flush+Flush 类型的侧信道攻击。我们的方法 TSCMiner 更具备一般性，它从指令层面检测处理器中共享资源引起的侧信道风险，并能无差别的同时兼顾核心内的共享资源和核心之间的共享资源。

TSCMiner 主要关注的是基于时间的侧信道。当然，由于检测的框架与攻击模型有直接的关系，我们的方法在检测微架构脆弱性方面还有一定的局限性。通过对现有的使用指令执行时间差异发起攻击的分析，我们发现一般指令的差异时间在 70 到 250 个时钟周期，而且，我们发现，时钟周期差异太小的话则攻击很难被识别到，时间周期差异太大的话，又会直接影响到信息传输的带宽，使得攻击的效果可以忽略不计。因此，我们的方法目前只关注了指令执行时间差异在 50-300 个时钟周期附近的指令。另外，我们的方法目前只是测试了每次触发

条件和检测条件都是单条指令的情况。

TSCMiner 对检测指令执行时间较长的指令组合引起的侧信道有较好的效果，它通过测试指令执行时间和被阻塞时候的执行时间的差异判断指令组合是否能够构建有效的时间侧信道。由于我们发现的隐藏指令的执行时间都较短，目前我们还没有发现可以有效的构建时间侧信道攻击的案例。但对于其他的威胁模型，隐藏指令是否能够有效的构建对应的攻击，我们将其作为以后的工作继续研究。

TSCMiner 在指令层面发现基于时间的侧信道，能够在处理器上较快的发现可能存在的单条指令引起的微架构脆弱性。而且，目前我们的方法还是只针对 x86 平台，但我们的测试框架可以直接移植到 RSIC 处理器平台的测试中，例如对 ARM 和 RISC-V 处理器的测试，唯一的区别是指令生成方法部分。

7.7 本章小结

本章节提出的方法为自动化的检测处理器中存在的微架构脆弱性提供了一种解决方案。处理器微架构脆弱性的检测需要基于一定的威胁模型展开，我们提出了基于指令操作码分类的时间侧信道的自动化搜索方法。首先通过深度优先遍历方法将 x86 指令空间中的指令进行搜索和生成，并将生成的指令在所需要测试的处理器中执行，筛选出可以被执行的指令；紧接着将指令进行反汇编并根据助记符进行分类。然后，我们将分类的指令放入基于时间的侧信道的自动化测试框架进行检测。我们将提出的方法在 4 种 x86 处理器中进行了测试，不仅发现了现有的一些侧信道攻击方法，而且也发现了一些新的侧信道攻击方法。总的来看，我们的方法和检测出的结果为积极有效的展开微架构脆弱性防御提供了参考依据。

第 8 章　总结与展望

处理器中存在的指令安全脆弱性会对计算机系统的信息安全带来威胁。攻击者利用处理器中的指令安全脆弱性对计算机系统发起攻击，能达到窃取计算机中私密信息、使计算机系统宕机、改变程序执行行为等目的。然而，与处理器硬件相关的攻击，其隐蔽性较高，很难被现有的恶意代码检测工具检测到，而且由于这些脆弱性是处理器硬件本身的问题，很难通过软件补丁的方法对相关的攻击展开防御。本文我们提出了在指令层面检测处理器中脆弱性风险的方法 Skipscan 和 TSCMiner，其中 Skipscan 主要是为了发现处理器中的隐藏指令，具有高效率、高覆盖率和高精度的特点；TSCMiner 主要是为了高效的检测处理器中存在的时间侧信道，而且，能够同时保证对隐藏指令和合法指令类型的覆盖，这为保证处理器的安全性提供了参考。

值得注意的是，第3、4、5章描述的增加 x86 处理器中隐藏指令的测试效率和测试覆盖率的方法都被集成到了 Skipscan 的测试工具中，所述方法分别都能达到各个章节描述的实验效果，且各个章节的方法可以叠加起来综合使用，第6章描述的方法也被集成到了工具 Skipscan 中，目的是为了提升第3、4、5章检测出来隐藏指令的测试精度。TSCMiner 是相对于 Skipscan 工具而独立开发的测试指令序列触发的微架构侧信道的测试工具，它用到了 Skipscan 工具的测试结果，目的是为了探索隐藏指令和合法指令形成的组合可能触发的处理器微架构脆弱性。目前，TSCMiner 只支持基于时间的侧信道，后续我们会继续探索自动化的检测其他的微架构脆弱性相关的侧信道攻击的方法。

8.1　总结

本论文面向处理器中存在的指令安全脆弱性展开研究，针对处理器中存在的隐藏指令，具体研究高效率、高覆盖率和高精度的测试方法；针对处理器中的指令序列触发的微架构脆弱性，具体研究了自动化的高效的测试指令序列的方法。接下来，我们从五个方面阐述本论文具体的贡献：

- 第一，针对 x86 处理器中隐藏指令的测试效率低的问题，我们提出了基于合法指令中操作数的最小测试集的方法，减少了对合法指令空间中立即数和偏移量操作数的测试，从而达到了提高预留指令在所测试的所有的指令中的占比的效果。与现有的方法进行了对比，我们的方法将现有的预留指令的占比从 21.89%

提升到了 79.78%。

• 第二，针对 x86 处理器中隐藏指令的测试覆盖率低的问题，我们分析了 x86 的指令前缀域，提出了基于组合优化的指令前缀生成方法，增加对合规的指令前缀组合的覆盖，从而覆盖了更多的指令前缀组合与预留操作码、预留操作数共同构建而成的预留指令。与现有的方法进行对比，我们的方法增加了对指令前缀域的覆盖率，所测试的预留指令数量是现有的研究工作的 260 倍。

• 第三，针对 x86 处理器中隐藏指令的测试覆盖率低的问题，我们分析了 x86 的指令操作码和操作数域，提出了基于必要搜索深度实时检测的 DFS 算法，通过对现有的合法指令的指令格式进行分析，我们得到了每种指令格式对应的必要的搜索深度，当我们的方法检测到 DFS 算法的搜索深度小于我们总结的必要搜索深度的时候，会及时的增加搜索深度。与现有的方法相比，我们的方法保证了对 x86 指令空间的搜索深度，从而增加了对 x86 指令空间中指令操作码和操作数的覆盖率，所测试的预留指令数量是现有方法的 268 倍。

• 第四，针对 x86 处理器中隐藏指令的测试精度低的问题，我们提出了基于多种反汇编器的交叉检查方法，通过使用多种反汇编器对测试阶段检测出的隐藏指令进行重新测试，从而提升测试的精度，降低误报率，在指令前缀长度为 1 和 4 个字节的时候，误报率分别降低了 66.77% 和 95.55%。

• 第五，针对 x86 处理器中存在的微架构脆弱性，我们提出了基于指令微操作码分类的时间侧信道检测方法，方法首先对合法指令和隐藏指令的操作码进行分类处理，然后对于有相同的操作码的指令进行交叉测试，从而提升了检测出指令时间侧信道的效率。我们将方法在 4 中 x86 处理器上进行评估，我们不仅发现了已有的侧信道攻击类型，而且发现了新的侧信道类型。

8.2 展望

诚然，本论文还有一些需要继续深入探索的科学和工程问题：

• 第一，第4章使用的增加覆盖率的方法针对的是 x86 指令的前缀域，第5章使用的增加覆盖率的方法针对的是 x86 指令的操作码和操作数域，它们都是基于第3章提出的高效率的测试方法的基础之上又提升了覆盖率。需要注意的是，对前缀域、操作码域和操作数域增加覆盖率的方法综合运用能够更进一步的增加测试的覆盖率，也就是将第4章和第5章的方法同时使用。本论文并没有给出实验结果和讨论，因为综合应用的问题只是需要时间去测试，并不牵扯到理论的创新，所以我们此工程任务留作未来的工作。

- 第二，我们目前只将 Skipscan 运行在 64-bit 长模式，值得注意的是 x86 处理器支持多种执行模式，这也是一个工程类型的问题，我们将对其他模式中是否有隐藏指令这个问题留作以后的工作。
- 第三，我们系统的研究了对 x86 处理器中隐藏指令的搜索方法，我们的指令空间搜索方法同样适合于对 x86 指令集相关应用的测试，例如反汇编器、CPU 模拟器等。
- 第四，由于处理器架构的具体硬件实现极其复杂，而且处理器的执行状态还会受到电压、温度、电磁等外部因素的干扰，所以研究处理器中的微架构脆弱性相关的脆弱性是一个多变量复杂的科学问题，需要进一步深入的研究。
- 第五，处理器中的微架构脆弱性风险的检测需要特定的威胁模型。我们将对瞬态执行攻击和基于争用的侧信道攻击等的检测方法留作以后的研究内容，特别是针对隐藏指令可能引起的侧信道攻击。

参考文献

[1] 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.

[2] Kocher P, Horn J, Fogh A, et al. Spectre attacks: Exploiting speculative execution [C/OL]// 2019 IEEE Symposium on Security and Privacy (SP). 2019: 1-19. DOI: 10.1109/SP.2019.00002.

[3] Evtyushkin D, Ponomarev D. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations [C]//Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. 2016: 843-857.

[4] Union E. The definition of vulnerability [EB/OL]. July 2017. https://www.enisa.europa.eu/topics/risk-management/current-risk/risk-management-inventory/glossary#G52.

[5] Wang Y, Liu P, Wang W, et al. On a consistency testing model and strategy for revealing risc processor's dark instructions and vulnerabilities [J]. IEEE Transactions on Computers, 2021.

[6] Bhunia S, Hsiao M S, Banga M, et al. Hardware trojan attacks: Threat analysis and countermeasures [J/OL]. Proceedings of the IEEE, 2014, 102(8): 1229-1247. DOI: 10.1109/JPROC.2014.2334493.

[7] Elnaggar R, Chakrabarty K, Tahoori M B. Hardware trojan detection using changepoint-based anomaly detection techniques [J/OL]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2019, 27(12): 2706-2719. DOI: 10.1109/TVLSI.2019.2925807.

[8] Cassano L, Iamundo M, Lopez T A, et al. Deton: Defeating hardware trojan horses in microprocessors through software obfuscation [J]. Journal of Systems Architecture, 2022: 102592.

[9] Baumgarten A, Steffen M, Clausman M, et al. A case study in hardware trojan design and implementation [J]. International Journal of Information Security, 2011, 10(1): 1-14.

[10] Forte D, Bao C, Srivastava A. Temperature tracking: An innovative run-time approach for hardware trojan detection [C/OL]//2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 2013: 532-539. DOI: 10.1109/ICCAD.2013.6691167.

[11] Marcelli A, Sanchez E, Sasselli L, et al. On the mitigation of hardware trojan attacks in embedded processors by exploiting a hardware-based obfuscator [C/OL]//2018 IEEE 3rd International Verification and Security Workshop (IVSW). 2018: 31-37. DOI: 10.1109/IVSW.2018.8494850.

[12] Chakraborty R S, Narasimhan S, Bhunia S. Hardware trojan: Threats and emerging solutions [C]//2009 IEEE International high level design validation and test workshop. IEEE, 2009: 166-171.

[13] Banga M, Hsiao M S. A novel sustained vector technique for the detection of hardware trojans [C]//2009 22nd international conference on VLSI design. IEEE, 2009: 327-332.

[14] Banga M, Hsiao M S. A region based approach for the identification of hardware trojans [C]// 2008 IEEE International Workshop on Hardware-Oriented Security and Trust. IEEE, 2008: 40-47.

[15] Collins R R. The pentium f00f bug [EB/OL]. https://www.drdobbs.com/embedded-systems/the-pentium-f00f-bug/184410555.

[16] Mark E, Dmitry S, Maxim G. Undocumented x86 instructions to control the cpu at the microarchitecture level in modern intel processors. [EB/OL]. 2021. https://raw.githubusercontent.com/chip-red-pill/udbgInstr/main/paper/undocumented_x86_insts_for_uarch_control.pdf.

[17] Qin S, Zhang C, Chen K, et al. idev: exploring and exploiting semantic deviations in arm instruction processing [C]//Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021: 580-592.

[18] Fogh A. Covert shotgun: Automatically finding smt covert channels [EB/OL]. Sep 2016. https://cyber.wtf/2016/09/27/covert-shotgun/.

[19] Gras B, Giuffrida C, Kurth M, et al. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. [C]//NDSS. 2020.

[20] 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.

[21] Sanders J. Spectre and meltdown explained: A comprehensive guide for professionals [EB/OL]. May 2019. https://www.techrepublic.com/article/spectre-and-meltdown-explained-a-comprehensive-guide-for-professionals/.

[22] Aldaya A C, Brumley B B, ul Hassan S, et al. Port contention for fun and profit [C/OL]// 2019 IEEE Symposium on Security and Privacy (SP). 2019: 870-887. DOI: 10.1109/SP.2019.00066.

[23] Gras B, Razavi K, Bos H, et al. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks [C/OL]//27th USENIX Security Symposium (USENIX Security 18). Baltimore, MD: USENIX Association, 2018: 955-972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[24] Domas C. Breaking the x86 isa [C]//Black Hat. 2017.

[25] Li X, Wu Z, Wei Q, et al. Uisfuzz: An efficient fuzzing method for cpu undocumented instruction searching [J/OL]. IEEE Access, 2019, 7: 149224-149236. DOI: 10.1109/ACCESS.2019.2946444.

[26] Dofferhoff R, Göebel M, Rietveld K, et al. iscanu: A portable scanner for undocumented instructions on risc processors [C/OL]//2020 50th Annual IEEE/IFIP International Conference

on Dependable Systems and Networks (DSN). 2020: 306-317. DOI: 10.1109/DSN48063.2020.00047.

[27] Jiang M, Xu T, Zhou Y, et al. Examiner: automatically locating inconsistent instructions between real devices and cpu emulators for arm [C]//Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022: 846-858.

[28] Xiao Y, Zhang Y, Teodorescu R. Speechminer: A framework for investigating and measuring speculative execution vulnerabilities [Z]. 2019.

[29] Weber D, Ibrahim A, Nemati H, et al. Osiris: Automated discovery of microarchitectural side channels [J]. arXiv preprint arXiv:2106.03470, 2021.

[30] Oleksenko O, Fetzer C, Köpf B, et al. Revizor: Testing black-box cpus against speculation contracts [Z]. 2021.

[31] Guarnieri M, Köpf B, Reineke J, et al. Hardware-software contracts for secure speculation [C]//2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021: 1868-1883.

[32] Wikipedia. Vulnerability (computing) [EB/OL]. December 2022. https://en.wikipedia.org/wiki/Vulnerability_(computing)#cite_ref-rfc4949_5-1.

[33] of Standards N I, Technology. Guidelines on active content and mobile code [EB/OL]. December 2022. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-28ver2.pdf.

[34] Wikipedia. Meltdown (security vulnerability) [EB/OL]. December 2022. https://en.wikipedia.org/wiki/Meltdown_(security_vulnerability).

[35] Wikipedia. Spectre (security vulnerability) [EB/OL]. December 2022. https://en.wikipedia.org/wiki/Spectre_(security_vulnerability).

[36] Dofferhoff R. A performance evaluation of platform-independent methods to search for hidden instructions on risc processors [J]. Leiden University, 2019.

[37] Easdon C. Undocumented cpu behaviour on x86 and risc-v microarchitectures: A security perspective [D]. University of Bristol, 2019.

[38] Wikipedia. 指令集架构 [EB/OL]. February 2022. https://zh.wikipedia.org/wiki/%E6%8C%87%E4%BB%A4%E9%9B%86%E6%9E%B6%E6%A7%8B.

[39] Wikipedia. 微架构 [EB/OL]. February 2022. https://zh.wikipedia.org/wiki/%E5%BE%AE%E6%9E%B6%E6%A7%8B.

[40] Wu L M, Wang K, Chiu C Y. A bnf-based automatic test program generator for compatible microprocessor verification [J/OL]. ACM Trans. Des. Autom. Electron. Syst., 2004, 9(1): 105–132. https://doi.org/10.1145/966137.966142.

[41] Bentley B. Validating the intel(r) pentium(r) 4 microprocessor [C/OL]//Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232). 2001: 244-248. DOI: 10.1145/378239.378473.

[42] Bentley B. Validating a modern microprocessor [C]//Etessami K, Rajamani S K. Computer Aided Verification. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005: 2-4.

[43] Kaivola R, Ghughal R, Narasimhan N, et al. Replacing testing with formal verification in intel® coretm i7 processor execution engine validation [C/OL]//CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification. Berlin, Heidelberg: Springer-Verlag, 2009: 414–429. https://doi.org/10.1007/978-3-642-02658-4_32.

[44] Kodakara S V, Mathaikutty D A, Dingankar A, et al. Model based test generation for microprocessor architecture validation [C/OL]//20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07). 2007: 465-472. DOI: 10.1109/VLSID.2007.108.

[45] Malandain D, Palmen P, Taylor M, et al. An effective and flexible approach to functional verification of processor families [C/OL]//Seventh IEEE International High-Level Design Validation and Test Workshop, 2002. 2002: 93-98. DOI: 10.1109/HLDVT.2002.1224435.

[46] Aharon A. Test program generation for functional verification of powepc processors in ibm [C/OL]//32nd Design Automation Conference. 1995: 279-285. DOI: 10.1109/DAC.1995.249960.

[47] Reid A, Chen R, Deligiannis A, et al. End-to-end verification of processors with isa-formal [C]//Chaudhuri S, Farzan A. Computer Aided Verification. Cham: Springer International Publishing, 2016: 42-58.

[48] Herdt V, Große D, Le H M, et al. Verifying instruction set simulators using coverage-guided fuzzing* [C/OL]//2019 Design, Automation Test in Europe Conference Exhibition (DATE). 2019: 360-365. DOI: 10.23919/DATE.2019.8714912.

[49] Goel S, Slobodova A, Sumners R, et al. Verifying x86 instruction implementations [C/OL]// CPP 2020: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. New York, NY, USA: Association for Computing Machinery, 2020: 47–60. https://doi.org/10.1145/3372885.3373811.

[50] Heule S, Schkufza E, Sharma R, et al. Stratified synthesis: Automatically learning the x86-64 instruction set [J/OL]. SIGPLAN Not., 2016, 51(6): 237–250. https://doi.org/10.1145/2980983.2908121.

[51] Foutris N, Gizopoulos D, Psarakis M, et al. Accelerating microprocessor silicon validation by exposing isa diversity [C/OL]//MICRO-44: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA: Association for Computing Machinery, 2011: 386–397. https://doi.org/10.1145/2155620.2155666.

[52] Mitra S, Seshia S A, Nicolici N. Post-silicon validation opportunities, challenges and recent advances [C/OL]//Design Automation Conference. 2010: 12-17. DOI: 10.1145/1837274.1837280.

[53] Wagner I, Bertacco V. Reversi: Post-silicon validation system for modern microprocessors

[C/OL]//2008 IEEE International Conference on Computer Design. 2008: 307-314. DOI: 10.1109/ICCD.2008.4751878.

[54] Martignoni L, Paleari R, Reina A, et al. A methodology for testing cpu emulators [J/OL]. ACM Trans. Softw. Eng. Methodol., 2013, 22(4). https://doi.org/10.1145/2522920.2522922.

[55] Martignoni L, Paleari R, Roglia G F, et al. Testing cpu emulators [C]//Proceedings of the eighteenth international symposium on Software testing and analysis. 2009: 261-272.

[56] Paleari R, Martignoni L, Fresi Roglia G, et al. N-version disassembly: Differential testing of x86 disassemblers [C/OL]//ISSTA '10: Proceedings of the 19th International Symposium on Software Testing and Analysis. New York, NY, USA: Association for Computing Machinery, 2010: 265–274. https://doi.org/10.1145/1831708.1831741.

[57] Jay N, Miller B P. Structured random differential testing of instruction decoders [C]//2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018: 84-94.

[58] Woodruff W, Carroll N, Peters S. Differential analysis of x86-64 instruction decoders [M]. EasyChair, 2021.

[59] Martignoni L, Paleari R, Roglia G F, et al. Testing cpu emulators [C/OL]//ISSTA '09: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. New York, NY, USA: Association for Computing Machinery, 2009: 261–272. https://doi.org/10.1145/1572272.1572303.

[60] Martignoni L, Paleari R, Reina A, et al. A methodology for testing cpu emulators [J/OL]. ACM Trans. Softw. Eng. Methodol., 2013, 22(4). https://doi.org/10.1145/2522920.2522922.

[61] Kim T, Shin Y. Reinforcing meltdown attack by using a return stack buffer [J/OL]. IEEE Access, 2019, 7: 186065-186077. DOI: 10.1109/ACCESS.2019.2961158.

[62] Schwarz M, Lipp M, Moghimi D, et al. Zombieload: Cross-privilege-boundary data sampling [C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 753-768.

[63] Minkin M, Moghimi D, Lipp M, et al. Fallout: Reading kernel writes from user space [J]. arXiv preprint arXiv:1905.12701, 2019.

[64] Van Schaik S, Milburn A, Österlund S, et al. Ridl: Rogue in-flight data load [C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 88-105.

[65] Moghimi D, Lipp M, Sunar B, et al. Medusa: Microarchitectural data leakage via automated attack synthesis [C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 1427-1444.

[66] Kiriansky V, Waldspurger C. Speculative buffer overflows: Attacks and defenses [J]. arXiv preprint arXiv:1807.03757, 2018.

[67] Maisuradze G, Rossow C. Ret2spec: Speculative execution using return stack buffers [C/OL]//CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and

Communications Security. New York, NY, USA: Association for Computing Machinery, 2018: 2109–2122. https://doi.org/10.1145/3243734.3243761.

[68] Koruyeh E M, Khasawneh K N, Song C, et al. Spectre returns! speculation attacks using the return stack buffer [C/OL]//12th USENIX Workshop on Offensive Technologies (WOOT 18). Baltimore, MD: USENIX Association, 2018. https://www.usenix.org/conference/woot18/presentation/koruyeh.

[69] Kiriansky V, Waldspurger C A. Speculative buffer overflows: Attacks and defenses [J/OL]. CoRR, 2018, abs/1807.03757. http://arxiv.org/abs/1807.03757.

[70] Bhattacharyya A, Sandulescu A, Neugschwandtner M, et al. Smotherspectre: Exploiting speculative execution through port contention [C/OL]//CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2019: 785–800. https://doi.org/10.1145/3319535.3363194.

[71] 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]//27th USENIX Security Symposium (USENIX Security 18). 2018: 991-1008.

[72] Ahmad B A. Real time detection of spectre and meltdown attacks using machine learning [J/OL]. CoRR, 2020, abs/2006.01442. https://arxiv.org/abs/2006.01442.

[73] Fadiheh M R, Stoffel D, Barrett C, et al. Processor hardware security vulnerabilities and their detection by unique program execution checking [C]//2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2019: 994-999.

[74] Zhang R, Deutschbein C, Huang P, et al. End-to-end automated exploit generation for validating the security of processor designs [C]//2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018: 815-827.

[75] Tol M C, Gulmezoglu B, Yurtseven K, et al. Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings [C]//2021 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2021: 616-632.

[76] Nilizadeh S, Noller Y, Pasareanu C S. Diffuzz: differential fuzzing for side-channel analysis [C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 176-187.

[77] Pan Z, Mishra P. Automated detection of spectre and meltdown attacks using explainable machine learning [C/OL]//2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). 2021: 24-34. DOI: 10.1109/HOST49136.2021.9702278.

[78] Van Bulck J, Moghimi D, Schwarz M, et al. Lvi: Hijacking transient execution through microarchitectural load value injection [C/OL]//2020 IEEE Symposium on Security and Privacy (SP). 2020: 54-72. DOI: 10.1109/SP40000.2020.00089.

[79] Müller L. Kpti a mitigation method against meltdown [J]. Advanced Microkernel Operating Systems, 2018: 41.

[80] Hill M D, Masters J, Ranganathan P, et al. On the spectre and meltdown processor security vulnerabilities [J/OL]. IEEE Micro, 2019, 39(2): 9-19. DOI: 10.1109/MM.2019.2897677.

[81] Canella C, Schwarz M, Haubenwallner M, et al. Kaslr: Break it, fix it, repeat [C/OL]//ASIA CCS '20: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2020: 481–493. https://doi.org/10.1145/3320269.3384747.

[82] Khasawneh K N, Koruyeh E M, Song C, et al. Safespec: Banishing the spectre of a meltdown with leakage-free speculation [C]//2019 56th ACM/IEEE Design Automation Conference (DAC). 2019: 1-6.

[83] Herzog B, Reif S, Preis J, et al. The price of meltdown and spectre: Energy overhead of mitigations at operating system level [C/OL]//EuroSec '21: Proceedings of the 14th European Workshop on Systems Security. New York, NY, USA: Association for Computing Machinery, 2021: 8–14. https://doi.org/10.1145/3447852.3458721.

[84] Kim S, Mahmud F, Huang J, et al. Revice: Reusing victim cache to prevent speculative cache leakage [C/OL]//2020 IEEE Secure Development (SecDev). 2020: 96-107. DOI: 10.1109/SecDev45635.2020.00029.

[85] Canella C, Pudukotai Dinakarrao S M, Gruss D, et al. Evolution of defenses against transient-execution attacks [C/OL]//GLSVLSI '20: Proceedings of the 2020 on Great Lakes Symposium on VLSI. New York, NY, USA: Association for Computing Machinery, 2020: 169–174. https://doi.org/10.1145/3386263.3407584.

[86] Cauligi S, Disselkoen C, Moghimi D, et al. Sok: Practical foundations for spectre defenses [J/OL]. CoRR, 2021, abs/2105.05801. https://arxiv.org/abs/2105.05801.

[87] Fustos J, Farshchi F, Yun H. Spectreguard: An efficient data-centric defense mechanism against spectre attacks [C/OL]//DAC '19: Proceedings of the 56th Annual Design Automation Conference 2019. New York, NY, USA: Association for Computing Machinery, 2019. https://doi.org/10.1145/3316781.3317914.

[88] Wang G, Chattopadhyay S, Gotovchits I, et al. oo7: Low-overhead defense against spectre attacks via program analysis [J/OL]. IEEE Transactions on Software Engineering, 2021, 47 (11): 2504-2519. DOI: 10.1109/TSE.2019.2953709.

[89] Genkin D, Yarom Y. Whack-a-meltdown: Microarchitectural security games [systems attacks and defenses] [J/OL]. IEEE Security Privacy, 2021, 19(1): 95-98. DOI: 10.1109/MSEC.2020.3036146.

[90] Mushtaq M, Novo D, Bruguier F, et al. Transit-guard: An os-based defense mechanism against transient execution attacks [C/OL]//2021 IEEE European Test Symposium (ETS). 2021: 1-2. DOI: 10.1109/ETS50041.2021.9465429.

[91] Kollenda B, Koppe P, Fyrbiak M, et al. An exploratory analysis of microcode as a building block for system defenses [C/OL]//CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2018: 1649–1666. https://doi.org/10.1145/3243734.3243861.

[92] Mambretti A, Neugschwandtner M, Sorniotti A, et al. Speculator: A tool to analyze speculative execution attacks and mitigations [C/OL]//ACSAC '19: Proceedings of the 35th Annual Computer Security Applications Conference. New York, NY, USA: Association for Computing Machinery, 2019: 747–761. https://doi.org/10.1145/3359789.3359837.

[93] Domas C. Sandsifter's source code [EB/OL]. July 2017. https://github.com/xoreaxeaxeax/sandsifter.

[94] 百度百科. Boundary value analysis [EB/OL]. https://baike.baidu.com/item/%E8%BE%B9%E7%95%8C%E5%80%BC%E5%88%86%E6%9E%90%E6%B3%95/4137943.

[95] aquynh. Capstone [EB/OL]. 2021. https://www.capstone-engine.org.

[96] Intel. Intel 64 and ia-32 architectures software developer's manual [EB/OL]. https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[97] AMD. Amd64 architecture programmer's manual: Volumes 1-5 [EB/OL]. https://www.amd.com/system/files/TechDocs/40332.pdf.

[98] Wang G, Zhu Z, Li S, et al. Differential testing of x86 instruction decoders with instruction operand inferring algorithm [C]//2021 IEEE 39th International Conference on Computer Design (ICCD). IEEE, 2021: 196-203.

[99] Intel. Xed [EB/OL]. 2021. https://github.com/intelxed/xed.

[100] GDB. Gdb disassembler [EB/OL]. 2022. https://www.sourceware.org/gdb/current/.

[101] GNU. Gnu objdump [EB/OL]. 2022. https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_6.html.

[102] Gruss D, Maurice C, Fogh A, et al. Prefetch side-channel attacks: Bypassing smap and kernel aslr [C]//Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. 2016: 368-379.

[103] Evtyushkin D, Ponomarev D. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations [C]//Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. 2016: 843-857.

[104] Department of defense trusted computer system evaluation criteria [M/OL]. London: Palgrave Macmillan UK, 1985: 1-129. https://doi.org/10.1007/978-1-349-12020-8_1.

[105] Okamura K, Oyama Y. Load-based covert channels between xen virtual machines [C]//Proceedings of the 2010 ACM Symposium on Applied Computing. 2010: 173-180.

[106] Ristenpart T, Tromer E, Shacham H, et al. Hey, you, get off of my cloud: exploring informa-

tion leakage in third-party compute clouds [C]//Proceedings of the 16th ACM conference on Computer and communications security. 2009: 199-212.

[107] Liu F, Yarom Y, Ge Q, et al. Last-level cache side-channel attacks are practical [C]//2015 IEEE symposium on security and privacy. IEEE, 2015: 605-622.