基于系统调用依赖的 Linux 内核模糊测试技术研究

◆杨　鑫[1,2]　张　超[2]　李　贺[1,2]　单　征[1]

（1.数学工程与先进计算国家重点实验室　河南　450002）

（2.清华大学 网络科学与网络空间研究院　北京　100083）

摘要：对 Linux 内核系统调用接口进行模糊测试是发现 Linux 内核漏洞的重要方法。现有测试方案生成系统调用测试例的质量和效率存在瓶颈，其中一个重要原因是忽视了系统调用之间的依赖关系。本文提出并实现了一个原型系统 Dependkaller，通过动静结合的方式分析系统调用之间基于共享内核数据的依赖信息，并有效利用多种依赖信息指导 Linux 内核模糊测试，持续高效生成和变异系统调用测试例。实验结果表明 相比于现有方案 MoonShine，我们的方案 Dependkaller 静态依赖分析结果更为全面 相比于现有方案 syzkaller，我们的方案 Dependkaller 动态测试的代码覆盖率提升了 16.89%，多发现了 51.22%（即 21 个）的漏洞。

关键词：系统调用依赖；操作系统内核测试；模糊测试；静态分析；动态分析

Linux 操作系统被应用在绝大部分服务器和包括 Android 在内的许多场景中，支撑着互联网关键基础设施的运转和信息技术产业的发展，占据了大量的操作系统市场份额。而操作系统的核心是内核，提供了操作系统最基本的功能和安全保障。如果在内核中发生安全问题，将给整个操作系统的稳定和安全带来巨大的威胁。因此，对内核进行安全测试，提前发现未知安全问题，意义极其重大。目前，模糊测试是发掘内核缺陷的主要手段，即通过向内核提供的接口发送大量精心构造的输入，然后观测内核是否异常来发现内核的问题。一般来说，内核模糊测试的主要接口是系统调用，因为它是用户态程序与操作系统内核交互的主要接口。而且，系统调用实现中的问题可能导致非特权的普通用户态程序危害整个操作系统[1]。

系统调用接口模糊测试工具的输入通常为多个系统调用组成的系统调用序列，构造系统调用序列的效率和质量是制约模糊测试效果的关键。Linux 内核目前提供了 300 多个系统调用，如果随机组合这些系统调用构成测试输入，其输入空间无比巨大。而且，系统调用之间存在着依赖关系，系统调用之间顺序并不能随意组合。部分系统调用的行为依赖于之前一些系统调用创建的共享内核状态，如果没有正确的内核状态，后续的系统调用只能触发浅层的错误处理代码，而不能执行到深层的核心功能代码[1]，测试效率将非常低下。

为此，现有的系统调用接口模糊测试工具[2][3]依靠人工不断编写了大量的规则，描述了基于资源参数类型的依赖关系，即使用特定资源作为参数的系统调用前，必须存在产生该资源的系统调用，这些资源包括各种类型的文件描述符等。例如，对于 read 系统调用，其第一个参数为文件描述符资源类型，那么为了使 read 系统调用能够执行到核心功能代码，就必须确保该文件描述符处于“open”状态，即需要在 read 系统调用前使用 open 系统调用来达到该要求。

然而，系统调用之间的依赖不仅仅表现在使用资源作为参数的系统调用与生成资源的系统调用间，内核中还存在着一些对于同一进程共享的数据，对这些共享内核数据进行更改的系统调用，也会影响到其他使用这些内核数据的系统调用的执行。并且，这些内核数据通常并不直接体现到系统调用的参数或返回值上，很难直接通过系统调用的参数和返回值类型来分析得出。

为此，本文通过设计和实现 Dependkaller，以自动提取系统调用间基于共享内核数据的依赖，并有效融合多种依赖关系，为系统调用序列的生成和变异提供高效指导。Dependkaller 首先通过静态分析 Linux 内核各个系统调用的代码，提取出基于共享内核数据的依赖，并分配相应权重。然后，将内核数据依赖与现有的资源数据依赖进行融合，形成系统调用间的静态依赖权重，并用该权重指导随机生成和变异系统调用序列，对内核进行测试。然后，结合测试过程中收集的内核代码覆盖信息反馈，统计分析实际产生新的代码覆盖的系统调用序列，从而提取系统调用间的动态依赖权重。最后，利用动态依赖权重，不断修正静态分析得到的依赖权重，提升依赖权重的合理性，不断提高系统调用序列生成和变异的效率。实验结果表明，Dependkaller 比 MoonShine 的静态分析更全面，比 syzkaller 对内核代码模糊测试的代码覆盖率提升 16.89%，多发现了 51.22%（即 21 个）漏洞。

总体上，本文系统地研究了基于系统调用依赖的 Linux 内核模糊测试技术，主要贡献如下：

（1）通过对内核系统调用接口模糊测试技术的研究，揭示了目前方案在系统调用依赖分析和运用上存在的不足，即静态依赖分析存在漏报和误报，而且测试过程中对依赖信息的运用上存在扩展性差和延续性弱的问题。

（2）提出了基于共享内核数据的新型的系统调用间依赖关系；并通过细致分析内核中大量存在的间接函数调用信息，提高数据流分析的精度，提取了系统调用间基于共享内核数据的依赖信息，降低了依赖信息的漏报。

（3）利用了分析得到的基于共享内核数据的依赖信息，有效融合现有的资源数据依赖信息和根据代码覆盖反馈信息提取的动态依赖信息，为系统调用序列的生成和变异提供高效指导，提高了对内核系统调用接口模糊测试的代码覆盖率和 bug 发现数量，具有良好的应用价值和效果。

本文剩下的部分组织如下：第 1 节主要对 Linux 内核系统调用接口模糊测试技术进行介绍，分析存在的挑战、目前的研究现状和存在的问题；第 2 节介绍本文提出的基于系统调用依赖的 Linux 内核模糊测试技术的设计和实现 第 3 节进行了实验验证；最后，在第 4 节进行了总结。

1 内核系统调用接口模糊测试技术研究现状和分析

针对内核系统调用接口的模糊测试，其测试输入为系统调用序列。模糊测试器通过生成或变异的方式产生系统调用序列，然后在目标操作系统上执行，来调用内核功能，以期最大化代码覆盖率，发现尽可能多的程序缺陷。

测试输入构造的效率是模糊测试效果的关键环节。根据文献[4]，系统调用接口模糊测试输入构造技术大致可以分为基于随机、基于参数类型、基于 hook（钩子）和基于反馈四类。基于随机的技术最为简单，可以追溯到 1991 年，当时 Tin Le 研发了 tsys fuzzer[5]，它简单地循环使用随机生成的参数调用随机选择的 UNIX

System V 系统调用，直到系统崩溃。基于参数类型的模糊测试技术，如 Dave Jones 研发的 Trinity [2]通过为参数添加一些随机性来生成测试用例。而基于 hook 的模糊测试技术试图通过在运行程序时拦截系统调用，改变正在执行的系统调用的合法参数值来对系统调用进行模糊测试，获取更高的测试正确率，这一类主要有文献[6, 7]。基于反馈的技术首先出现在目前最成熟的 Linux 内核模糊测试框架 syzkaller[3]上，它将测试例的代码覆盖率用于反馈，基于遗传算法来保留能够贡献新的代码覆盖率的测试例并变异、生成新的测试例，对提高代码覆盖率具有显著效果，挖掘了 Linux 内核大量漏洞。可见，传统的系统调用接口模糊测试技术主要关注于提高构造系统调用参数的效率上。

然而，系统调用之间也存在着一些影响。系统调用要实际执行内核功能，通常需要一些预先存在的内核状态，否则不能通过状态检查，只能执行浅层的错误处理代码。这些内核状态只能被其他系统调用按照特定顺序进行设置，系统调用必须按照特定顺序执行，系统调用之间的参数等资源存在依赖，这些统称为系统调用依赖。

根据文献[1]，内核状态主要体现在两个方面。一是用户态可见的资源数据（如各种类型的文件描述符等），它是通过一些系统调用以返回值的方式产生，而另一些系统调用依赖它作为参数，才能正常执行。比如系统调用 read 和 write 必须使用 open 正确执行返回的文件描述符作为第一个参数，才能正常执行。我们称这种依赖为资源数据依赖；二是用户态不可见的共享内核数据，它是通过共享的内核数据传递，一些系统调用会修改一些共享内核数据，而另一些系统调用的执行会受这些共享内核数据的影响，我们称这种依赖为内核数据依赖。举例来说，Linux 系统调用 msync 内核数据依赖于系统调用 mlockall，因为 mlockall 通过修改共享内核数据 struct vma 的 vm_flags 域，而统一进程中的 msync 会根据 vma.vm_flags 的值，执行不同的程序路径，即影响了 msync 的控制流。

为了高效地生成系统调用序列，syzkaller 依靠专家经验编写大量系统调用的描述规则，规定了各个系统调用的参数和返回值信息，包括类型、值范围、传递方向等。其中比较重要的是设定了一些特殊的参数类型，即资源类型，它们主要由一些系统作为返回值生成，可供其他一些系统调用使用。在系统调用生成时，如果某个系统调用需要某种类型资源作为参数，syzkaller 从之前已产生的相同类型资源返回值中选择一个作为参数（大概率）；或者直接在该系统调用前插入一个生成该类型资源的系统调用（小概率，或之前未产生过所需的资源），然后将返回值作为该系统调用的参数。同时，在生成下一个系统调用时，会优先选择具有更多相同资源参数的系统调用。为此，syzkaller 会先分析所有被测系统调用的参数类型，如果两个系统调用都接收相同的资源类型作为参数，则给这两个系统调用赋予更高的相关性，即在有其中一条系统调用存在的情况下，下一条系统调用被选择的权重。syzkaller 这样的生成和变异系统调用序列的方式，基本确保了系统调用序列符合资源数据依赖，且具有较强的资源相关性。然而，对于共享内核数据依赖，该方案没有进行考虑和应用。

MoonShine[1]首次提出了基于共享内核数据的隐含依赖概念，但它是用 Smatch[8]分析 Linux 内核源码的抽象语法树进行提取的，只得到系统调用间是否存在依赖的信息，未考虑内核中大量存在的函数指针，存在较多漏报，并且存在静态分析固有的误报问题。而且，它只是基于真实用户态程序动态执行产生的系统调用 trace（执行记录），提取包含资源数据依赖和内核数据依赖的系统调用序列，存在一些局限：(1) 依赖于 trace 多样性，一般的 trace 包含的系统调用种类和组合有限，较难扩展提升覆盖率；(2) trace 一般比较正常，需要基于依赖信息大量变异，但 MoonShine 没有进行有效指导，后续生成和变异系统调用序列的效率依然很低。

综合分析当前内核系统调用接口模糊测试技术，发现当前对构造有效系统调用接口测试输入的要求上已有较全面的认识，但对系统调用间依赖的分析手段和有效运用还存在一些不足。主要为：(1) 静态分析内核数据依赖的漏报和误报问题；(2) 测试中对系统调用依赖运用存在的扩展性差和延续性弱的问题。

2 基于系统调用依赖的 Linux 内核模糊测试技术设计和实现

为了解决目前存在的不足，我们设计和实现了 Dependkaller。首先，通过尽可能全面的静态分析，获取低漏报的内核数据依赖信息，并分配相应权重。然后，融合内核数据依赖权重和资源数据依赖权重生成静态依赖权重，指导带权重地随机生成和变异初始的系统调用序列。为了减少人工和静态分析提取的静态依赖权重的漏报和误报，我们在充分基于静态依赖权重生成和变异系统调用序列并执行后，根据内核代码覆盖信息记录工具 KCOV[9]的反馈，分析覆盖了新的内核代码的系统调用序列，形成动态的系统调用依赖权重。最后，融合静态依赖权重和动态依赖权重，共同指导后续的系统调用序列生成和变异。

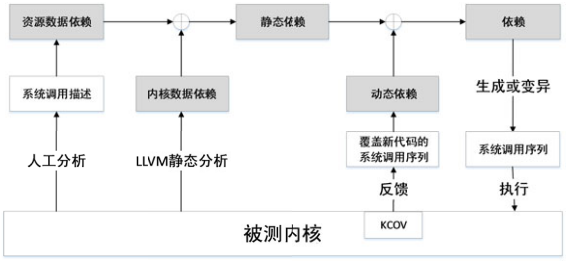

图 1 Dependkaller 架构图

2.1 静态分析内核数据依赖信息

根据文献[1]对内核数据依赖（隐含依赖）的定义：前面系统调用的执行会修改共享内核数据，从而影响后面与该共享内核数据有关的系统调用的执行。更具体的定义：如果系统调用 A 在条件语句中使用了共享内核数据 v，则 v 是 A 的读依赖项；如果系统调用 B 会对共享内核数据 v 进行写入，则 v 是 B 的写依赖项；如果 A 的写依赖项与 B 的读依赖项存在相同的共享内核数据，则 A 对该共享内核数据的写入，会影响 B 的条件判断，即影响 B 的控制流，则称系统调用 B 内核数据依赖于系统调用 A。

为了使静态分析尽可能减少漏报，针对内核代码的特点，我们主要采用了细致的数据流分析和全面的函数指针分析。针对写依赖的共享内核数据，只需分析系统调用会写入的共享内核数据。针对读依赖的共享内核数据，需分析系统调用会读入的共享内核数据，且通过数据流分析，读入的值还会影响条件语句的值。因为系统调用由一系列内核函数进行实现，系统调用对于共享内核数据的依赖也主要体现在内核函数的实现代码中。因此，需要先基于函数调用信息分析系统调用相关的内核函数，然后分析这些内核函数对共享内核数据的依赖，最后得到各个系统调用对共享内核数据的依赖。

Dependkaller 的静态分析实现主要基于 LLVM 编译器框架。首先使用 LLVM 中的 Clang 编译器编译 Linux 内核源码为 IR（Intermediate Representation，中间表示），然后编写 LLVM 静态分析程序，对 IR 进行分析，提取内核数据依赖信息。因为 Linux 内核源码编译生成为多个 IR 文件，每个 IR 文件对应于内核的单个源码文件，且每个 IR 源码中包含了内核函数的具体实现。因此，总体分析步骤为：(1) 逐个分析内核源码 IR，提取函数调用信息和内核函数对共享内核数据的依赖信息；(2) 根据函数调用信息提取系统调用实现相关的内核函数，分析得到系统调用对共享内核数据的依赖信息。具体实现如下。

2.1.1 分析内核函数对共享内核数据的依赖信息

因为 Linux 内核中主要使用复杂数据类型结构体（struct）来组织共享数据，用结构体中简单数据类型域（field）来存储共享数据。因此，本文主要分析基于结构体中域（struct.field）数据的依赖信息，并以 struct 的类型名和 field 的变量名作为依赖项的标识。算法流程如图 2 所示，对于一个内核函数的 IR，若其中存在 Store 指令的目标操作数为 struct.field 类型，则该 struct.filed 为该内核函数的一个写依赖项（第 3-6 行）；若其中存在 Load 指令的源操作数为 struct.filed 类型，且该 Load 指令的目的操作数会影响 Branch 指令（对应于 C 语言的 if、while、for 等语句）或 Switch 指令（对应于 C 语言的 switch 语句）的操作数，则该 struct.filed 为该内核函数的一个读依赖项（第 8-12 行）。因为 Load 指令的目的操作数只需影响 Branch 指令或 Switch 指令的操作数，可以直接也可以间接影响，所以需要对 Load 指令的目的操作数进行数据流分析，看是否有流入 Branch 指令或 Switch 指令的操作数。在基于 LLVM 实现时，需要以源操作数为 struct.filed 类型的 Load 指令的目的操作数为起点 Value，递归地分析其 User（LLVM IR 中，一个 Value 的 User 是将该 Value 作为操作数的指令或表达式），如果存在某个 User 为 Branch 指令或 Switch 指令，则该 struct.filed 为该函数的读依赖项。

算法 1：分析内核函数对共享内核数据的依赖信息

```
1：  AnalyzeFuncDep（Function）  {
2：    for  （Instruction in Function）  {
3：      if  （IsStoreInst（Instruction））  {
4：        StoreId = GetStoreId（Instruction）;
5：        if  （IsStructType（StoreId））
6：          FuncWriteDepSet.insert（StoreId）;//写依赖
7：      }
8：      else if  （IsLoadInst（Instruction））  {
9：        LoadId = GetLoadId（Instruction）;
10：       if  （IsStructType（LoadId））
11：         if  （WillFlowToCondition（Instruction））//数据流分析读依赖
12：           FuncReadDepSet.insert（LoadId）;//读依赖
13：    }
14：  }
15：  FuncWriteDepMap[Function] = FuncWriteDepSet;//写依赖
16：  FuncReadDepMap[Function] = FuncReadDepSet;//读依赖
17：}
18：
19：bool WillFlowToCondition（Instruction）  {
20：  AnalyzedUser.insert（Instruction）;
21：  for  （User in Users（Instruction））  {
22：    if  （NotInAnalyzedUser（User））  {//避免递归造成的死循环
23：      AnalyzedUser.insert（User）;
24：      if  （IsBranchInst（User）  || IsSwithInst（User））
25：        return true;
26：      if  （WillFlowToCondition（User））  //递归分析 User
27：        return true;
28：    }
29：  }
30：  return false;
31：}
```

图 2 算法流程

2.1.2 分析系统调用对共享内核数据的依赖信息

通过 2.1.1，我们得到了各个内核函数对共享内核数据的依赖信息。然而，我们需要的是系统调用对共享内核数据的依赖信息，可通过分析提取各个系统调用实现相关的内核函数，然后将相关内核函数对内核数据的依赖信息整合到对应的系统调用中即可。为了提取各个系统调用实现相关的内核函数，需要分析内核中的函数调用关系。因为内核中存在大量的通过函数指针进行的间接函数调用，而 LLVM 内置的函数调用分析程序不支持分析间接函数调用，为了减少分析漏报造成依赖信息的缺失，我们采用基于类型分析[10]的方法来保守地找出所有的间接函数调用。具体为：首先收集所有被取地址的函数（即函数指针的目标函数），只要其参数类型和和数量与间接调用函数的相同，则认为这个被取地址的函数是被间接调用的目标，从而构建两者的调用关系。通过这样，可以获取整个内核的函数调用信息，然后以各个系统调用为起始节点，逐层分析被调函数，可得到系统调用实现相关的内核函数。然后，将相关内核函数对共享内核数据的依赖信息整合到对应的系统调用即得到系统调用对内核数据的读写依赖信息。最后，交叉对比各个系统调用的读、写依赖的 struct.field，如果系统调用 A 写依赖的 struct.field 与系统调用 B 读依赖的 struct.field 相同，则系统调用 B 依赖于 A，而相应的 struct.field 则为系统调用 A 和 B 的依赖项。最终可得到各个系统调用之间依赖的所有 struct.field。

2.2 融合生成静态依赖权重，指导初步模糊测试

为了充分利用分析所得的系统调用知识，深入持续指导模糊测试进程，我们将分析得到的系统调用对共享内核数据的依赖信息转化为系统调用选择的权重，即存在某条系统调用的前提下，下一条系统调用被选择的权重，权重值为系统调用之间依赖的 struct.field 数量的归一化值。并与 syzkaller 提供的资源依赖权重（同样地归一化后）相加，得到系统调用状态转移的静态权重。最后，基于该静态权重，影响模糊测试生成和变异系统调用序列时，选择下一条系统调用的权重，从而在生成和变异系统调用序列时高效地反映系统调用的资源和内核数据依赖。

2.3 融入动态依赖权重，指导后续模糊测试

由于人工分析的资源数据依赖和静态分析得到的内核数据依赖存在漏报和误报的可能，单纯依靠静态依赖权重指导模糊测试还存在效率上的提升空间。为此，先充分利用静态权重指导生成和变异形成大量系统调用序列，经内核执行后，根据 KCOV 反馈的内核代码覆盖信息，分析能覆盖新的内核代码的系统调用序列，得到当前实际的系统调用组合情况。因为系统调用依赖主要表现为前面系统调用对后面系统调用的影响，所以通过统计分析前后系统调用对的频次，即可得到当前的动态依赖权重。动态依赖权重反映了实际存在依赖的系统调用对，可用于修正静态依赖权重存在的漏报、误报或权重值的偏差。具体方法是将动态依赖权重（同样地归一化后）与静态依赖权重相加，得到实际依赖权重，指导后续生成和变异系统调用序列。随着越来越多系统调用的产生，动态依赖权重将越来越准确，实际依赖权重也将越来越准确，指导模糊测试的效率也将会越来越高。

3 实验

Dependkaller 主要代码实现包括两个部分，第一部分为基于 LLVM 的静态分析模块，实现对 Linux 内核源码中系统调用内核数据依赖信息的提取；第二部分为基于 Syzkaller 的内核模糊测试模块，实现对目标内核的模糊测试。为了验证 Dependkaller 的效果，本部分主要回答两个问题：（1）Dependkaller 分析的内核数据依赖信息是否更全面？（2）Dependkaller 能否提高代码覆盖率和 bug 发现数量？

3.1 Dependkaller 分析的内核数据依赖信息是否更全面

3.1.1 实验方法

实验对象为 MoonShine 分析所采用的 Linux 4.19 内核。主要将 Dependkaller 对 Linux 内核静态分析提取的内核数据依赖信息，与 MoonShine 通过 Smatch 获取的内核数据依赖信息[1]进行对比。

3.1.2 实验结果

Linux 4.19 内核包含的系统调用数目为 366 个。而 Dependkaller 和 MoonShine 对 Linux 4.19 内核静态分析提取的依赖组数（会影响其他系统调用数据流和控制流的系统调用数量）分别为 294 和 228 组，Dependkaller 比 MoonShine 多找出 64 组（28%）存在内核数据依赖的系统调用，如表 1。

表 1 静态分析结果对比

	MoonShine	Dependkaller	增长率
依赖组数	228	294	28.07%

3.1.3 结果分析

通过实验结果可以看出，Dependkaller 提取的系统调用间内核数据依赖信息更全面。

3.2 Dependkaller 能否提高代码覆盖率和 bug 发现数量

3.2.1 实验环境

实验环境操作系统为 Ubuntu Server 16.04，配置有 384GB 内存，72 个 Intel CPU，可满足实验对内存和 CPU 的需求。

3.2.2 实验方法

为了评估 Dependkaller 的设计方案对代码覆盖率的影响，我们分别实现了只加入静态依赖权重的 Dependkaller（以下简称 Statickaller），以及在后期（语料库达到 2 万时）融入了动态依赖权重的 Dependkaller，并与原始的 syzkaller 进行对比测试。均提供空的语料库，对所有系统调用进行测试，不开启崩溃复现功能。为了尽可能消除实验的随机性和资源不足的影响，每个实验组配置 8 个 QEMU 虚拟机，每个虚拟机配置 4GB，运行 4 个模糊测试器进程，同时生成、变异和执行系统调用序列。测试目标为当前最新的 Linux 5.20 内核。测试直到三组实验程序的代码覆盖率均无明显提升为止。

3.2.3 实验结果

因为内核代码空间巨大，模糊测试随机性较大，实验持续了 10 天，三组实验程序的内核代码覆盖率才没有较明显增长。因为在记录代码覆盖信息时，采用基本块边覆盖数较为可行且准确，所以记录了最终 syzkaller、Statickaller 和 Dependkaller 的基本块边覆盖数量，以及 Statickaller 和 Dependkaller 相对 syzkaller 的增长率。同时，记录造成 crash 的 bug 类型数量。如表 2。

表 2 模糊测试结果对比

	边覆盖数（增长率）	bug 类型数（增长率）
syzkaller	312146	41
Statickaller	347838（11.43%）	55（34.15%）
Dependkaller	364854（16.89%）	62（51.22%）

3.2.4 结果分析

通过实验结果可以看出，在充分进行模糊测试后，融入了内核数据依赖指导的 Statickaller 比 syzkaller 在基本块边覆盖方面，有了 11.43%的增长，多发现 34.15%的 bug。而同时融入了动态依赖的 Dependkaller，又有了 4.89%的增长，最终获得 16.89%的代码覆盖率增长和 51.22%（21 个）的 bug 数量增长。可以看出，先基于静态依赖进行充分测试，后融入动态依赖提升测试效率的方案具有一定效果。

4 结论

本文研究发现了 Linux 内核系统调用接口模糊测试技术在系统调用依赖的分析和运用上存在的不足，设计和实现了基于系统调用依赖的 Linux 内核模糊测试工具 Dependkaller，通过较全面的动静结合方式分析和运用依赖信息，持续高效生成和变异系统调用序列，Dependkaller 的静态依赖分析结果更为全面，对 Linux 内核的模糊测试比 syzkaller 在代码覆盖率方面提升了 16.89%，多发现 51.22%（21 个）的 bug。结果表明 Dependkaller 对于提高 Linux 内核系统调用接口模糊测试具有一定的应用价值。

参考文献：

[1]Pailoor S，Aday A，Jana S.MoonShine：Optimizing OS Fuzzer Seed Selection with Trace Distillation.In the 27th USENIX Security Symposium. 2018.

[2]Dave Jones. Trinity [EB/OL]. https://github.com/kernelslacker/trinity.

[3]Dmitry Vyukov.Syzkaller [EB/OL]. https://github.com/google/syzkaller.

[4]H.Han and S.K.Cha. IMF：Inferred Model-based Fuzzer. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security（CCS）.2017.

[5]Tin Le.tsys [EB/OL].https://groups.google.com/forum/?hl=+en#!msg/alt.sources/V_B37EtnWKQ/NztsljVYV84J.

[6]Ian Beer.pwn4fun Spring 2014-Safari-Part II [EB/OL]. http://googleprojectzero. blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html.

[7]Dmytro Oleksiuk.IOCTL fuzzer [EB/OL].https://github.com/Cr4sh/ioctlfuzzer.

[8]N.Brown.Smatch：pluggable static analysis for C [EB/OL].https://lwn.net/Articles/691882/.

[9]Dmitry Vyukov. Kernel：add kcov code coverage [EB/OL].https://lwn.net/Articles/671640/.

[10]Lu K，Song C，Kim T，et al.UniSan：Proactive kernel memory initialization to eliminate data leakages[C]. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security（CCS）.2016.

基金项目：国家自然科学基金联合基金项目（U1736209）资助。

一种基于高速交换缓存的网络隔离 SoC 芯片结构研究

◆谷会涛[1]　孙懿峰[2]

（1.91977 部队　北京　100841；2.91991 部队　浙江　316000）

摘要：随着网络技术的快速发展，网络安全威胁日益严重。网络隔离技术广泛应用于工业控制等类型内部网络的安全防护应用中。本文介绍了网络隔离技术基本原理和技术发展，给出了当前典型网络隔离技术的组成和实现方式，提出了一种基于高速交换缓存的网络隔离 SoC 芯片结构，最后分析评估了该硬件结构的设计约束和性能。

关键词：网络隔离；SoC 芯片；高速缓存；芯片结构