高可靠信息系统非相似冗余架构中的执行体同步技术

于 洪* 刘勤让 魏 帅 兰巨龙
(中国人民解放军战略支援部队信息工程大学信息技术研究所 郑州 450000)

摘 要：非相似冗余架构被广泛使用到信息系统中，提高系统的安全性和可靠性。非相似冗余架构中的执行体之间存在差异，当系统正常工作时，执行体表现一致，但在面对恶意攻击行为时，执行体会表现出不一致。架构通过比较执行体的表现监控系统、感知威胁，从而提升系统安全可靠性。执行体的同步监控，是所有非相似冗余架构都需要解决的难题。目前没有针对同步技术比较系统性的描述和总结。该文首先对执行体同步问题进行了抽象建模，然后提出基于同步点的同步技术分类方法，并分别对每种技术的基本方式、流行度、优缺点进行了总结。该文还提出了影响同步效果的3个重要指标：同步点、误报率和性能，同时给出了同步技术的数学模型，可用于同步技术的设计评估。最后，结合网络弹性工程领域和软件定义晶上系统领域的发展，指出了同步技术未来的发展潜力和可能的发展方向。

关键词：非相似冗余；多变体；异构冗余；执行体同步
中图分类号：TN915.08; TP309 **文献标识码**：A **文章编号**：1009-5896(2024)05-0001-15
DOI: 10.11999/JEIT231048

Executer Synchronization in Highly Reliable Information System with Dissimilar Redundancy Architecture

YU Hong LIU Qinrang WEI Shuai LAN Julong
(*Institute of Information Technology, Information Engineering University, Zhengzhou* 450000, *China*)

Abstract: Dissimilar redundancy architecture is widely used in information systems to improve their security and reliability. When the system operates normally, the executers behave consistently, but when faced with malicious attacks, the executers exhibit inconsistency. The architecture improves the security and reliability of the system by comparing the performance of the executers to monitor the system and perceive threats. The synchronization of executers is a challenge that all dissimilar redundancy architectures need to address. There is currently no systematic description and summary of synchronization technology. This article is a review of executer synchronization techniques in dissimilar redundancy architectures. First, the importance of synchronization in dissimilar redundancy systems is explained and a standardized description of synchronization is provided. Then, a synchronization technology classification method based on synchronization points is proposed and the basic process, popularity, advantages and disadvantages of each class are summarized separately. This article also proposes three important indicators that affect synchronization performance, namely synchronization point, false alarm rate, and performance, and provides a mathematical model for synchronization technology, which can be used for design evaluation. Finally, this article combines the development of cyber resilience and software defined system on wafer, and points out the potential and possible directions for the future development of synchronous technology.

Key words: Dissimilar redundancy; Multi-variant; Heterogeneous redundancy; Executer synchronization

1 引言

信息系统面临各种漏洞及后门威胁。一般情况下，漏洞/后门存在特定软硬件平台，对它们的利用也要基于特定软硬件平台来实现。因此，漏洞/后门只在相同的平台之间可以快速、广泛传播，在架构和设计不同的计算机上则呈现惰性。但是目前信息系统在部署时倾向采用市面上流行的架构，或者集成现有第三方库，或者大量使用开源代码，这使系统具备相似性。这种情况被称为同质化(mono-

收稿日期：2023-09-27；改回日期：2024-03-16；网络出版：2024-04-03
*通信作者： 于洪 yuhong_3210@163.com
基金项目：国家重点研发计划(2022YFB4401401)
Foundation Item: The National Key R&D Program of China (2022YFB4401401)

culture)，同质化降低了信息系统攻击的成本，给漏洞的传播增加了便利，加剧了漏洞的威胁[1,2]。

非相似冗余架构(Dissimilar Redundancy Architecture, DRA)被广泛使用到信息系统中，用于防范因同质化带来的漏洞/后门问题，其主要解决思路就是在系统中引入非相似冗余，使系统对外呈现不一致，增加攻击者攻破系统的难度。

非相似的来源有多样性和异构性。典型软件多样性技术有指令集随机化技术[3–6]和内存重排技术[7–10]等，从不同粒度上，对段的基地址或段内元素进行随机化操作。比如为运行的程序提供动态的、随机的指令集，使攻击者无法得知当前使用的指令集，或者改变代码段中函数的相对顺序和数据段中数据对象的相对顺序，从而使攻击者无法构造相应的漏洞利用程序。多样性可以体现在软件的源代码层次[11]、中间代码层次[12]或者可执行文件层次[13]。异构性主要来源于设计异构，包括软件和硬件。比如最早在文献[14]中，就在相同的运算部件上运行不同设计团队开发的代码，后来的拟态防御系统[15–22]中，则是从硬件就采用不同设计，上层软件为满足硬件环境会做一定修改进行适配，或者直接重新设计。

同构冗余技术常用来提高系统的可靠性。比如文献[23,24]，采用的软件冗余技术，生成多个程序副本，通过比较副本执行表现，减小偶然原因导致的计算错误对系统的影响，增强系统容错。硬件冗余技术，将硬件作为单位进行复制冗余，主要是二模冗余(DMR)或者三模冗余(TMR)，如文献[25]。硬件实现的同构冗余，可以对系统软件和硬件的偶然错误有很好的容错能力，但是无法抵御针对软硬件共性漏洞和后门的恶意攻击。在分布式系统中，也广泛使用复制冗余体的方式，来达成容错[26,27]。在分布式系统的不同的机器上运行同样的应用程序，处理同样的请求。分布式系统内的同构冗余，同样也无法解决软件可能存在共性漏洞从而遭受共性攻击的问题。

冗余技术结合非相似性，并逐渐发展成熟，形成了较为系统和完善的非相似冗余架构。2006年，Cox等人[28]首次提出了较为完整的非相似冗余架构——N-Variant架构。自此为始，众多研究者都进行了非相似冗余架构方面的探索，提出了包括但不限于TightLip[29], Orchestra[30], CHUMVEE[31], DCL[32], ReMon[33], MvArmor[34], Bunshin[35], DMON[36], HeterSec[37]等在内的不同架构，在维护系统安全方面进行了诸多探索，关键的研究内容包括非相似执行体的生成、执行体同步监控、执行体结果表决等。非相似冗余架构中的执行体在应对正常的工作请求时，都可以有一致的表现；但在面对恶意攻击行为时，会出现不一致。架构基于此监控系统状态，感知威胁。执行体之间的差异越大，则共同的漏洞越少，安全性就越高，但也意味着实现的难度越大。因为要保证执行体在应对正常工作请求时行为一致，就必须对执行体进行同步控制，使执行体可以在指定的观察节点保持一致。执行体的同步监控技术，是非相似冗余架构保证系统安全的前提，也是所有非相似冗余架构都需要解决的难题，但是目前没有针对同步技术比较系统性的描述和总结。

本文总结了目前非相似冗余架构中的执行体同步技术。文章结构安排如下：第1节是引言，介绍了信息系统中存在的同质化问题，以及可以缓解同质化带来的安全问题的技术发展路线，点出了非相似冗余架构及其中重要的执行体同步问题；第2节概要分析了非相似冗余架构的特点，并抽象描述了执行体同步问题；第3节分类介绍了各种执行体同步技术；第4节对同步技术进行了总结归纳；第5节对执行体同步技术的发展进行了展望；第6节是全文的总结。

2 非相似冗余架构及执行体同步问题

2.1 非相似冗余架构特点

非相似冗余架构基本构成如图1所示。

在采用非相似冗余架构的信息系统中，一般存在两个或两个以上冗余执行体节点，每个节点上都运行一份功能等价但有所不同的软件。节点可以是纯软件的，一般称为变体，也可以是硬件实现的，一般称为异构执行体。在统一描述时，本文将冗余执行体节点统一称为执行体。在描述所引用的论文时，将依据原文习惯把执行体称为变体、克隆体、实例或者执行体。

非相似冗余架构中也存在输入管理节点、输出管理节点以及监控器节点。输入管理节点一般用于将接收到的数据复制分发给各冗余执行体，输出管理节点负责结果输出。输入输出管理节点决定了非

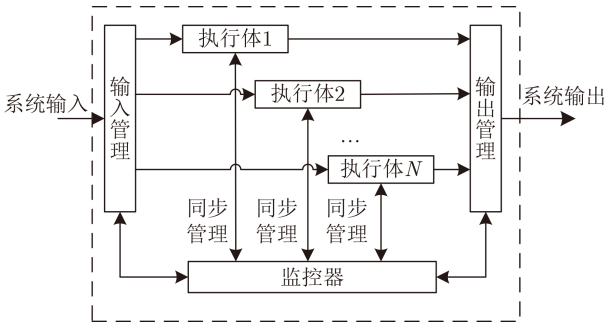

图 1 非相似冗余架构构成示意图

相似冗余的边界。在有些实现中，输入输出管理节点可以是融合的一个节点。监控器节点还可能被叫做控制器[38]、调度器[15]或者表决器等，用于执行体同步监控、执行体输出裁决、执行体异常检测、执行体动态调度等操作，是一个非相似冗余系统中的核心关键节点。

一个具备非相似冗余架构的系统，其典型工作流程一般为

(1) 输入管理节点将接收到的数据复制分发给各冗余执行体；

(2) 执行体在监控器节点的监控下处理数据，产生输出；

(3) 监控器节点对各冗余执行体产生的输出进行比较判决，鉴别出是否有异常节点，再根据情况做出相应处理(一般是异常节点停止服务，或者恢复到前一个确定正常的状态)；

(4) 数据从输出管理节点输出。

2.2 非相似冗余架构前提

从前面的描述中可知，非相似冗余架构存在一个将多个执行体输出的数据进行比较、表决后再输出的过程。这个过程，是监控器节点检测和定位执行体异常行为的关键，也是消除和降低攻击行为对系统不良影响的关键。正是有了这个过程，利用非相似冗余保证安全/可靠的原理才能发挥作用。

而监控器节点可以比较输出的前提是，这些输出在正常情况下，不论从输出内容上还是输出时间上，都要在一定的程度上保持一致。也就是说，冗余执行节点的输出需要同步，才能保证裁决过程具有可行性。

由于冗余执行体存在差异性，以及系统本身的一些不确定性因素影响，致使执行体不能保证完全一致的输出。因此，需要采用一些技术手段，对冗余执行体进行同步(输出内容和输出时间同步)，尽量降低表决失误的概率，以保证系统多模表决过程顺利进行。

但是多样性的设计越多，保持执行体之间功能等效和同步就越困难。这是非相似冗余架构中普遍存在的多样性和等效性/同步之间的矛盾。在系统中设计实现非相似冗余架构的前提是，必须要在多样性设计与等效性/同步之间获得平衡，使系统既可以因为执行体之间的差异性获取安全增益，也可以因为执行体之间的等价性和同步性保证系统平稳运行。

2.3 同步问题描述

为了描述同步问题，首先定义程序的执行过程为一个由状态组成的无限序列：$[S_0,\ S_1,\ \cdots,]$，其中S_i表示在节点i，程序的状态。对于有N个执行体的系统来说，程序的执行过程则是一个包含N元组的无限状态序列：$[<S_{0,0},\ S_{0,1},\cdots,S_{0,N-1}>,\ <S_{1,0},\ S_{1,1},\ \cdots,S_{1,N-1}>,\ \cdots,]$，其中$S_{i,j}$表示在节点$i$，执行体$j$的状态。程序的状态分为3种：正常状态、警告状态、异常状态。正常状态表示符合预期的状态，异常状态表示确定被攻击所破坏的状态，警告状态表示因异常行为被检测到但还未确认为异常状态的状态。

定义M为映射函数。对于没有经过任何变换的原始程序，其运行对应的状态序列定义为一个规范序列，也就是$[S_0,\ S_1,\ \cdots,]$。执行体上运行的程序因为经过了多样化的设计，其运行产生的状态序列和规范序列不一一对应，需要通过一个映射函数M进行转化，转化成规范序列。M_j表示执行体j的规范化函数。例如，如果执行体的多样化设计是根据某些规则修改内存地址，则映射函数需要将执行体更改后地址映射到规范地址。

定义T为状态转变函数。前一个状态到后一个状态的转化，由状态转变函数T加上特定的输入完成。T代表原始程序的状态转变函数，T_j代表执行体j的状态转变函数。状态的转换被分为两类：正常转换和异常转换。正常转换表示从正常状态转换到另一个正常状态，异常转换表示从正常状态转换到异常状态。如果一次攻击，让执行体从正常状态转换到异常状态，却没有被检测到，则认为此次攻击是成功的。

非相似冗余架构保证系统安全的原理就在于，执行体的软硬件在结构设计和实现上存在不同，保证了执行体之间存在相同软硬件漏洞/后门的概率是极低的，也就是攻击者难以同时对多数执行体发起相同的攻击，并产生相同的输出。同时，执行体运行的软件具有功能等价性，在正常工作情况下，每个执行步骤中，所有执行体的规范化状态都与原始程序状态相同。也就是说，任意一个执行体j在某个节点i的状态$S_{i,j}$都可以映射到同一规范状态S_i，也即，$\forall i\geq 0, 0\leq j,k<N$

$$M_j\left(S_{i,j}\right)=M_k\left(S_{i,k}\right)=S_i \tag{1}$$

另外，相同的输入和相同的当前状态，一定可以产生相同的输出和相同的下一状态，也即，

$\forall S_i\in[S_0,S_1,\cdots,], i\geq 0, 0\leq j,k<N$，$S_{i,j}$满足$M_j\left(S_{i,j}\right)=S_i$，$I$为输入，则

$$M_j(T_j(S_{i,j},I))=M_k(T_k(S_{i,k},I))=T(S_i,I)=S_{i+1} \tag{2}$$

因此，当攻击者进行系统攻击并成功攻破其中

一个执行体，会导致该执行体的输出与其他执行体不一致。监控器节点可以感知到这种情况，持续输出符合大多数执行体输出的正确结果，使攻击行为无法达到预期效果。理论分析表明，对于采用择多判决的三模异构冗余架构来说，在单执行体因出错概率低于5%时，整个系统出错的概率至少降低1个数量级[15]。

执行体同步的主要工作，就是保证式(1)和式(2)成立。理想情况下，是在每一次状态转换之后，都使用映射函数将执行体的状态映射到规范状态上进行检查，以判断执行体状态是否一致。但是这样的理想情况，在实际实现中，可能导致巨大的延迟，是真实应用无法承受的。因此，在什么样的粒度上，实现同步，达到同步目标和系统开销的平衡，也是同步研究中需要解决的问题。

3 执行体同步技术的相关研究

通常在实现中，研究者仅会选择一部分其认为重要的状态转换节点进行比较，这些被选取的节点，称为同步点。因为这些节点是所有状态转换节点的子集，因此，现实中的同步方案通常造成的是“有限形式的功能等价”。在本章中，我们以同步点为分类标准，介绍目前的执行体同步技术研究现状。

3.1 系统调用级的同步技术

将系统调用设置为同步点的同步技术，是多样性冗余架构中最常采用的同步技术。

以最早的多变体架构，N-Variant[28]为例，作者将内核的所有系统调用分成三大类：共享型、反射型和危险型。然后修改内核代码，为每种系统调用设计实现不同的封装器(wrapper)，利用封装器实现变体之间的同步。

共享型系统调用是与外部进行交互的系统调用。此类调用的封装器等待所有变体进入系统调用，比较它们的参数，如果一致，让后到的变体进行实际的系统调用。返回的输出保存到缓冲区中，先到的变体将数据拷贝走，再执行后续操作。反射型系统调用，是需要观测或者修改程序某些属性的系统调用。对此类调用，封装器的实现和共享型系统调用的封装器类似。区别就在于，每次都是变体0实施系统调用。危险型系统调用，是指打破变体系统所依赖的假设的系统调用。比如，进程使用execve系统运行一个新的可执行文件，这将脱离N变体系统保护范围。因此对于此类系统调用，封装器禁用系统调用并返回错误代码。

这种方式不足之处在于，它以攻击为导向来设计。设计者需要预先考虑想要防御的特定攻击类型，然后进行多样化方式、检测属性等的设计。变体同步的实现(主要是映射函数M_j)受多样化影响很大，通用性比较差。且不同的变体自动化属性，不能轻易融合并存，因为会带来不可控的不确定性因素，造成系统误报率进一步上升。方法分析处理了所有系统调用，封装器在系统调用时的等待、参数检查等是系统延迟的主要原因，相比普通系统，延迟增加了将近一倍。另外，在处理危险型系统调用时直接禁用，可能会给论文测试之外的其他类型应用程序带来未知问题。不过该篇文章首次提出了较为完整的多变体冗余执行架构[28]，在架构中应用到的同步技术为后续很多研究提供了参考。后续很多类似研究中，研究者在系统调用分类和处理细节方面进行了改进。

比如MvArmor[34]将系统调用分为敏感系统调用和非敏感系统调用，只对敏感系统调用锁步执行，非敏感调用则不需要锁步。同时，制定了3种安全程度不同的安全策略：(1)代码执行(code execution)：对execve和mprotect/mmap执行锁步；(2)信息泄露(information disclosure)：对I/O相关的系统调用执行锁步；(3)全面防范(comprehensive)：对所有系统调用执行锁步。这样，在某些安全模式下，不需要对所有系统调用都执行锁步，可以一定程度上提高系统效率。变体可用的PID、文件描述符等信息则通过变体管理器使用名称空间管理器确保一致。以PID为例，名称空间管理器使用分层结构为每个进程和线程分配虚拟PID和TID：当一个变体快速连续创建线程时，这些线程必须在所有变体中获得相同的虚拟TID，无论它们实际出现的顺序如何。MvArmor架构中所有模块都放置在Dune中，其运行效率高于使用纯软件的N-variant系统。在变体数目为2，安全策略为全面防范的模式下，MvArmor架构的服务器相对于普通服务器，增加的延迟为55.2%。

其他的改进还有TightLip[29]，它为系统调用增加了一个屏障(barrier)，用于实现原始进程和替身进程的同步。屏障的功能和封装器相似，但它只在处理敏感数据时，才生成一个“替身”进程，与原始进程并行执行。以缩小保护范围为代价节省了系统开销。

Orchestra[30]使用ptrace API来对变体的系统调用进行拦截和比对，相比于之前提到的方法，不需要对操作系统进行任何改动。

同样使用ptrace API拦截系统调用的有GHUMVEE[31]和ReMon[33]。ReMon是GHUMVEE的改进版本。GHUMVEE增加了对系统调用

之外的隐含输入问题处理，利用代理模块和共享缓冲区对可逃离监控的输入进行了处理，因此降低了系统的误报率。同时还使用“记录+重放”技术提供了对多线程的支持。ReMon在进程外增设一个单独的监视进程对与安全紧密相关的重点系统调用进行监控，另外使用两种类型的共享缓冲区，一种用于比较系统调用参数、复制系统调用结果，一种用于捕获和重放需要同步的操作，进一步提高了同步效率。文献[39]借鉴了ReMon两种缓冲区的设计，利用同步缓冲区(sync buffer)，控制主变体和从变体统一访问同步变量的顺序。主变体的同步代理(synchronization agent)捕获主变体中访问同步变量的顺序并保存至同步缓冲区中，并强制从变体也严格按照这个顺序访问同步变量，任何可能破坏这个访问顺序的线程都会被暂停执行。这个方法不依赖具体的多变体架构，通过少量修改可以应用到不同架构中，通用性更强。

除了在单个主机上执行多个变体的架构，在多个异构主机上执行多个变体的架构，也可以通过系统调用的同步实现变体同步[36,37,39]。异构主机上执行变体，大大提高了变体之间的“内部差异”级别，包括不同的指令集、端序、调用参数、系统调用接口，以及潜在的硬件安全功能差异。如果变体之间的通信开销不处理好，将造成系统性能损失巨大，如文献[36]。而在文献[39]中，将通信代价高昂的系统调用鉴别出来，避免重复操作，降低了分布式异构冗余变体系统的同步开销。

总结起来，在系统调用级的同步，一般都需要先将系统调用分类，然后根据系统调用的类型在系统调用外部额外增加系统调用管理模块，对变体的系统调用进行同步。只是在具体实现时，对系统调用的分类和处理的范围及方式上有所差别。在系统调用级进行同步，不需要对应用程序进行修改。相比于需要手动修改服务器以实现应用程序的监控的技术，这种技术让应用的部署负担相对较小，可以达成更为广泛的使用。这种方式的不足之处在于，需要对内核提供的几百个系统调用进行分析、分类，分别设计处理组件，工作量较大。另外，对所有(大部分)系统调用进行同步，频繁的等待和交互造成系统性能代价也比较高昂，因此更适用于纯软件实现的非相似冗余架构。同时，对于变体之间没有完全隔离的架构，仅进行系统调用同步是不够达成完全同步的，还需要对地址空间、定时器等进行管理，对变量进行访问控制。为了效率考虑省略这些步骤，就容易造成变体状态相异，使系统误判率较高，这也会影响系统安全效能。

3.2 I/O级的同步技术

针对系统调用级同步的性能问题，有研究者增大了同步粒度，在系统I/O位置进行同步控制。最早是在DieHard[10]架构中，在输出位置用共享内存区域内的独立缓冲区暂时储存每个变体的输出结果，周期性地对变体输出缓冲区的内容投票表决，至少两个缓冲区内容相同的情况下，才将其中结果发送至标准输出。为了使正常变体的输出尽可能等效，DieHard重定向了某些可能产生不同结果的系统调用(比如访问日期和系统时钟的函数)，以便所有变体都返回相同的值。DieHard采用了一种阻塞的表决方式，理论上发生错误的变体可能会进入无限的循环，这将导致整个变体的程序挂起，不产生输出，导致缓冲区永远也填不满，因此永远不会发生同步。如果要解决这个问题，可以设置一个定时器，终止在规定时间内没有填满缓冲区的变体程序，以打破阻塞状态。由于这种方式在标准I/O位置对变体进行监控，如果攻击者攻破某个变体，但不使用其标准I/O，则监控器是没有办法检测到异常行为的。因此其变体异常行为的检测是概率性的，对系统的安全增益也是概率性的。主要贡献还是对系统的可靠性和可用性的增强。

文献[40–42]都在DieHard基础上提出了改进。文献[40,41]提出了改进的DieHard版本(adaptive DieHard)以及DieHarder，在随机化堆对象布局技术上进行了优化，不过同步方式没有变化。Exterminator[42]不光可以在输出表决时检测到错误，还可以精确定位错误，自动生成补丁更正错误。当Exterminator发现错误时，它会转储一个包含堆完整状态的堆映像，然后利用概率错误隔离算法处理一个或多个堆图像，以定位缓冲区溢出和悬挂指针错误的来源和大小，然后生成运行时补丁来纠正错误。

纯I/O的同步方法，虽然效率相比于系统调用级别的同步提高了很多，但是由于其同步粒度增大，导致了系统的误判率比较高，因此，有研究者在BUDDY[38]架构中，提出了将IO同步和系统调用同步结合的方法。在I/O同步高效率的基础上，增加了系统准确性。BUDDY中，目标进程被复制到两个实例中。BUDDY首先通过一个内核空间的协调器(coordinator)拦截涉及I/O写的九个系统调用，重新定义它们的虚拟系统调用接口。然后通过一个用户空间的环形共享缓冲区同步虚拟后的I/O写系统调用。具体方式和ReMon类似。最后在所公开的数据实际离开操作系统时(即仅在文件写和套接字写时)检测是否存在分歧，报告内存泄漏。BUDDY通过这种设计，消除了由数据结构填

充引起的误报问题，原则上可以完全(即没有假阴性)和准确(即没有伪阳性)地检测任何存储器泄漏。同时节省了系统开销，BUDDY的SPEC测试仅增加了2.3%的开销，在Web服务器应用程序上的测试增加了4%的开销。不过BUDDY也呈现了一定局限性，因为系统中的不确定性源除了BUDDY处理的这几种I/O写系统调用，还有映射内存写操作等，出于性能考虑，BUDDY并没有实现。

在另一个架构ShadowExe[43]中，同步点被设置在了变体输入位置。ShadowExe使用影子执行防止系统敏感信息泄露。它利用虚拟机技术，同时运行一个程序的两个实体，一个公共实体，一个私有实体。公共实体的输入包括公共输入和假的私密输入，私有实体的输入包括公共输入和真的私密输入。两个实体的公共输入是相同的，私密输入是不同的，但是都需要保证输入时间的同时性。ShadowExe在输入位置，利用VNC协议[44]实现了公共输入的同步。为私密信息建立文件夹，利用架构实现的剪贴板处理程序，分别将真假私密信息发送给两个实体，保证了敏感数据非对称性地同时输入。由于这个系统的目的主要是防止敏感信息泄露，其设计不保证两个实体的输出完全一致，因此，其在输出位置只对结果进行监控，而不进行同步。

采用与BUDDY较为相似的同步方式的架构还有kMVX[45]，它在同一台机器上同时运行多个不同的内核变体来全面防御内核中的信息泄漏漏洞。kMVX也设置了主变体和从变体，在用户空间边界设置了系统调用同步组件，在内核与硬件的边界设置了I/O同步组件。kMVX结合了I/O同步与系统调用同步，将copy_to_user调用和put_user调用设置为系统调用的同步点。每次主变体到达一个同步点，系统调用被同步组件捕获，主变体将自己的数据复制一份，通过一个快速的内核间通道(用无锁共享内存哈希表实现)传输到从变体。每次传输的数据都被赋予一个独特的ID，这个ID基于当前程序的PID，系统调用ID和消息计数计算得来，让从变体可以准确鉴别数据来源，匹配自己相应的系统调用。而从变体到达一个同步点(比如copy_to_user系统调用)，则从通道中读取数据，并且逐字节比较自己缓冲区中的数据。如果没有检测到不一致，则返回1个确认给主变体；如果检测到不一致的情况，则将冲突字段清零返回给主变体。主变体要在获取从变体的返回后，才进行后续操作。也就是说，系统调用的同步采用了锁步的方式，会导致变体之间交互越频繁，系统的效率越低。

由于硬件只有1份，两个内核不能同时与硬件进行交互。因此，先由主变体与硬件交互，再将结果重放给从变体。与系统调用同步不同的是，I/O同步不使用锁步，主变体不需要等待从变体返回确认。

kMVX的同步方式与BUDDY类似，但效率看起来比BUDDY低得多，主要是因为kMVX的变体单位为内核而不是应用，内核之间的通信开销大于应用之间的通信开销。

总体来说，相对于系统调用级的同步，在I/O级别的同步，最主要的优点就是大大减少了监视器同步比较的次数，使架构性能损失更小，适用于非相似冗余边界大的架构(比如硬件及以上层次的非相似冗余架构)。但是单纯的I/O级别同步，会造成系统误报率较高，更适合部署中间输出较少的应用程序。如果想要降低误报率，则需要辅以其他的同步手段(比如系统调用级的同步)。

3.3 应用级的同步技术

前面介绍的变体同步技术，大多数都是在纯软件的变体架构上实现的，变体们共享硬件，可以用到共享缓冲区等实现变体间通信，可以在系统调用层次实现同步。对于软硬件结合的异构冗余架构来说，硬件隔离，让变体之间的通信代价更加高昂，使用细粒度的同步技术，将使系统开销过高。因此，就有了效率更高、同步粒度更大的应用级同步技术。

比如文献[15–22]等采用了动态异构冗余(Dynamic Heterogeneous Redundancy, DHR)架构设计系统，实现了变体从硬件到软件的异构冗余。

以内生安全交换机[17]为例，在交换机控制管理层实现三模异构冗余，各执行体分别采用了Atom E3930、QorIQ T1042和龙芯2K-1 000作为底层处理器，采用Ubuntu v16.04、VxWorks v6.9和Linus v3.10作为操作系统，其上运行协议栈源自同一源代码，进行了适量适配性的修改。然后利用一个FPGA实现执行体的输入输出代理、调度管理等功能。执行体直接与调度器连接，相互之间不能直接通信。由于不同执行体上系统时间、调度顺序、随机数生成都存在差异，给整个系统带来了较大的不确定性。

内生安全交换机中采用了一种应用级的同步技术，基于相对时间的异构执行体程序状态同步方法[46]，将应用程序中事件作为同步点，统一异构执行体上的事件发生顺序，保证程序状态转换的一致性。作者将引起程序状态变迁的事件分为3种：读事件、定时器事件和由读事件及定时器事件引起的其他事件。因此，统一了读事件和定时器事件的执行顺序，就相当于统一了所有事件的顺序。文献[46]定

义了一个相对时间，这个时间由调度器提供。每当执行体调用I/O复用检测函数(该函数负责检测定时器超时，将定时器事件和读事件等放入可执行队列)，需要向调度器发出申请。调度器在收到申请后，会检测其他执行体是否发出了相同的申请。如果所有有效执行体都发出了相同的申请，则返回一个确认信息，其中包含相对时间。执行体根据这个相对时间，判定自己的定时器是否超时，将超时定时器放入可执行队列，排队等待执行。同时，执行体所发的申请中还包含读事件的标记，调度器也会比较该内容，在返回的确认信息中包含对读事件的确认。执行体收到含有读事件确认的消息，才会将读事件放入可执行队列。通过这样的方式，交换机统一了读事件和定时器事件在不同执行体上的排序，从而统一了不同执行体的状态转换序列。

除此之外，交换机也采用执行体统一向调度器申请的方式，获取随机数种子，解决了随机数种子不一致带来的程序运行不确定性问题。并在执行体输出位置，也设置了数据同步检查，用来判别执行体状态。

和前面提到的很多同步技术不同，内生安全交换机在某个执行体出现异常情况后，不会直接停止对其的使用，而是会对其进行清洗(一般为重启)，然后重新与其他执行体同步，以此保证交换机始终保持一个较高的安全防护水平。在此过程中，其他执行体不会停止正常运行。因此，还涉及清洗完成后的执行体与其他执行体同步的问题。其采用的解决方案，是执行体定期向调度器备份关键状态信息，清洗完成后的执行体会向调度器申请最新的状态信息，对自身状态进行更新，以完成与其他执行体的同步。

该方法无论是相对时间申请，还是随机数申请，都采用了非阻塞的同步方式，偶尔的丢包不会影响程序的正常运行。相比于系统调用级同步方法，增加了同步粒度，大大减少了同步比较次数。但是需要对应用源程序进行修改，并不适用于商用软件，在这一点上，所有基于应用的同步技术都受到限制。

3.4 其他同步技术

在以上提到的同步技术之外，还有一些研究者提出了别的同步方式，只是应用范围并不广泛。比如文献[47]中实现了主从两个执行体在指令级的同步。作者使用了一个监控器，监控指令执行过程，基于文献[48]中提出的(Execution Indexing)技术计算执行点的索引信息，基于文献[49]中提出的离线技术比较索引信息，然后对齐具有相同索引信息的执行点。采用指令级的追踪，其产生的追踪信息量很大。一个小规模的执行程序，就可以产生超过30 GB的追踪信息，存储和处理这些信息对系统来说都是比较大的负担。测试也显示相比于仅仅运行两个程序，追踪和比较增加了2000多倍的延迟。因此，这个技术仅适用于内部的测试和调试。

还有Detile[50]架构，实现了在脚本解释器字节码级实现克隆体和本体的同步。Detile在本体和克隆体之间创建了一个程序内通信桥(IPC)。在字节码解释器循环中解释操作码和调用本机函数的位置安装钩子，将钩子位置设置为同步点。在同步点通过IPC交换信息完成本体和克隆体之间的同步。Detile是针对脚本引擎专门设计的，所以可以实现比系统调用同步更细粒度的同步，且也可以在功耗方面取得较好的平衡(性能损失在17%左右)，但也因此限制了其使用范围。

另外在有些架构设计中，更侧重系统的可靠性，而不是安全性。因此，及时发现变体异常不是关注重点，持续提供稳定的系统服务才是。在这样的系统中，通常采用松散同步的方式，也就是说，不要求变体之间进行严格同步，只需要保存执行日志，后续可重放执行过程，离线检测安全。这种思想也得到了一定应用[51–55]。

4 同步技术总结

本文提到的几种典型同步方式，总结列表如表1所示。

本表统计的系统开销并不是单纯的同步操作的开销，而是整个架构部署的开销，因此表格中数据仅供参考。本表统计的论文中使用到的测试集有以下几种：(1)用于Web服务器性能评测的测试集，包括WebBench 5.0，httperf和SpecWeb99；(2)轻量级的Web服务器测试集，包括thttpd, lighttpd和nginx；(3) 多线程基准测试集PARSEC，可以在模拟器中进行并行工作负载模拟；(4) 嵌入式基准测试集Mibench；(5) 通用CPU性能评估测试集SPEC CPU系列。为方便比较，表中尽量选用了基于相同benchmark(SPEC CPU系列)，和相同数量执行体(2-执行体)的测试结果作为参考。

4.1 同步点设置

同步技术设计最重要的是同步点的选择，直接决定了方法实现的工作量、同步效果和工作效率。从粒度上说，本文介绍到的有系统调用级、I/O级、应用级、指令级和语句级。同步的粒度越大，需要检测的次数越少，比较的工作量越小，方法效率就越高，比如BUDDY[38]，只比较系统最终输出，增加的系统开销仅为2.34%；还有文献[21]

表 1 执行体同步技术总结

序号	架构名称	同步点	严格同步	误报率	测试集	延迟/开销
1	N-Variant[28]	系统调用	是	高	WebBench 5.0	17.8%～93.77%
2	未命名[56]	系统调用	是	高	httperf	1.2%～68.93%
3	未命名[57]	系统调用	是	低	thttpd	31%～6x
4	TightLip[29]	系统调用	是	高	SpecWeb99	5%
5	Orchestra[30]	部分系统调用	是	低	SPEC CPU2000	16%
6	GHUMVEE[31]	部分系统调用	是	低	SPEC CPU2006	15%
7	DCL[32]	部分系统调用	是	高	SPEC CPU2006	6.37%
8	ReMon[33]	规则定义的系统调用	是	低	SPEC CPU2006	3.1%
9	MvArmor[34]	规则定义的系统调用	是	低	SPEC CINT2006	9.10%
10	Bunshin[35]	规则定义的系统调用	是	低	PARSEC	8.50%
11	未命名[58]	规则定义的系统调用	是	低	PARSEC 2.1	<15%
12	DMON[36]	规则定义的系统调用	是	低	lighttpd	443%
13	HeterSec[37]	规则定义的系统调用	是	低	lighttpd	50%
14	dMVX[39]	规则定义的系统调用	可选	低	lighttpd	3.10%
15	I-MVX[59]	规则定义的系统调用 &动态链接的插桩函数	是	低	SPEC CINT2006	2.13%
16	NG-MVEE[60]	部分系统调用	是	低	SPEC CPU2006	7%
17	DieHard[10]	I/O	是	高	SPECint2000	12%
18	ShadowExe[43]	所有输入	是	高	Adobe reader等应用程序	>100%
19	Exterminator[42]	I/O	是	低	SPECint2000	7.20%
20	BUDDY[38]	I/O写&部分系统调用	是	低	SPEC CPU2006	2.34%
21	kMVX[45]	I/O写&部分系统调用	是	低	nginx-1.10.1, lighttpd-1.4.48等	20%～50%
22	DHR Switch[16]	应用程序读事件 &定时器事件	是	低	交换机入网测试集	<3 ms
23	DHR MCU[61]	应用输出(查询响应)	是	低	N/A	N/A
24	DHR Router[21]	应用输出(路由表)	是	低	N/A	N/A
25	MimicBox[62]	输出	是	高	SPECint2006	<13%
26	Detile[50]	解释器字节码	是	低	N/A	17%
27	Dual Execution[47]	指令级	是	低	N/A	>2000x
28	MVX-CFI[13]	间接跳转/ 调用指令&返回指令	是	低	SPEC CINT2006	N/A
29	EXPERTISE[63]	内存写	是	低	MiBench	～5x
30	Varan[52]	所有系统调用	否	低	SPECint2006	14.20%
31	LDX[53]	输出	否	高	SPECint2006	4.70%
32	Respec[54]	程序段(Epoch)	否	低	PARSEC	18%～55%
33	Mx[51]	系统调用	否	N/A	SPEC CPU2006	17.91%

中，只比较路由协议下发的路由表，文献[64]中，只比较响应上位机查询产生的响应报文。反之则效率越低，比如Dual Execution[47]，执行体数目虽然只有两个，但是在每一条指令都执行同步操作，程序执行时间由毫秒级变为秒级，延迟增加了2000多倍。同在系统调用级的同步技术，也有粒度的区别。比如早期的N-Variant[28]，比较所有系统调用，系统延迟在最好情况下17.8%，最坏情况下达到93.77%。后续很多基于系统调用的同步，都设置了一定规则对系统调用进行分类，只同步规则定义的重要的系统调用，由此减少同步次数，提高执行效率。如表1所示，只比较规则定义的系统调用的同步技术，系统开销普遍更小。

同步的粒度越大，也意味着系统监控粒度越大，能监控到的执行点越少。攻击者相对有更大的操作空间来利用系统中的漏洞违反正常的等效属性，从而使原本可以检测到的攻击能够在没有检测到的情况下进行。比如在系统输出进行同步的方

法，没有办法检测到只破坏系统内部数据而不产生输出的攻击行为。通常要辅以更多的功能等效性保障措施，以提高系统安全性。比如文献[20,21]等只在应用程序输出位置进行同步的架构中，就增加了异构执行体的动态组合，来提高系统漏洞利用难度。

4.2 不确定性控制

误报率是同步技术的主要评价指标之一，指监控器节点通过判决检测到执行体遭受攻击(即存在执行体输出不一致的情况)，但实际上执行体并没有遭受攻击的次数，占监控器判决遭受攻击的总次数的百分比。出现误报，一般是由于系统中的不确定因素没有得到有效控制，造成的变体程序在正常运行的情况下，却在同步点呈现不一致。

非相似冗余架构的发展经历了从纯软件的非相似冗余到硬件同构、软件多样性的冗余，再到软硬件均异构的冗余。非相似冗余架构的发展过程，总体来说，是一个逐步扩大非相似冗余边界的过程，也是一个让同步问题变得更加复杂的过程。

纯软件的非相似冗余，在变体同步时，需要考虑的控制范围基本上在程序以内，不确定性因素包括进程号、线程号、随机数、文件描述符等。目前主要的处理手段一是代理，二是选择性忽略。代理是在不确定性因素输入/输出的位置，不使用真实的数值，而是复制主变体的数值，比如N-variant[28]、MvArmor[34]。选择性忽略则是分析出不确定性因素后，在程序编译过程中进行插桩，对插桩的元素，同步比较时予以忽略，比如mimicBox[62]。

软硬件结合实现的异构冗余架构中，不确定性因素更多，除了程序内的不确定性因素，还包括异构环境原因造成的不确定性因素(比如异构环境的程序调用机制不同，异构平台的时钟差异等)、异构实现造成的不确定性因素(比如因为硬件层面的异构导致的在驱动层设计的不同，以及由不同设计路线、不同开发团队实现导致的不同等)，和外部不确定性因素(比如用户输入时间、内容不确定，网络报文到达内容、时间不确定)等。目前在DHR架构中提供了解决这些问题的部分思路。比如，利用调度器提供统一的相对时间，替代异构执行体的系统时间，消除因时间问题造成的不确定性。利用调度器统一随机数种子获取途径，消除随机数问题造成的不确定性。利用调度器对所有外部不确定性因素进行统一代理，消除了外部不确定性因素造成的问题。在软件设计实现层面，细化对软件的要求，保证异构程序在状态转换触发和输出数据内容及大小方面是一致的。驱动层面的异构不可避免，但是驱动层面一般不涉及输出，只是影响系统运行快慢，因此可以不理会实现层次的差异，只是在某些特定节点，对异构执行体进行程序运行状态的同步。

误报率的高低直接决定了多变体架构在检测变体故障方面的有效性，是架构安全能力的重要体现，也是是同步技术的重要评价指标。

4.3 性能开销及其他指标

性能开销是同步技术的另一个重要指标。同步技术的开销主要包括锁步等待的时间开销和信息交换的通信开销。我们定义时间开销为从执行体初次发起同步操作，到同步操作完成所耗费的时间；定义通信开销为用于同步操作所交换的信息量。锁步等待的时间开销和通信开销都受同步点设置的粒度的影响，同步点越多，锁步等待越频繁，时间开销和通信开销越大。要优化锁步等待的时间和通信开销，主要是减少需要锁步等待的同步点数量，通过细致分析，剔除不必要的锁步操作，如MvArmor[34]中，只对规则定义的敏感系统调用采用锁步执行。此外，等待的时间开销还受执行体通信方式影响，目前，执行体之间的通信可以分成3类：(1)在同一硬件平台内部，应用之间的通信，如文[31,33]等；(2) 在同一硬件平台内部，内核之间的通信，如文献[41]；(3)不同硬件平台之间的通信，如文献[15,16,36]等。一般说来，应用之间的通信造成的时间开销小于内核之间的通信造成的时间开销，前两者又小于硬件平台之间通信的时间开销。要优化这部分时间开销，要尽量减少代价高昂的通信。对通信开销的优化，除了要减少同步点数量，还需要根据执行体构成设计轻量级通信方案，减少不必要内容的传输。目前各论文中性能开销的测试数据，绝大部分是系统整体运行结果，而不是单纯的同步技术开销，这给同步技术之间的比较分析造成了一定的困难。只有文献[47]将程序执行、变体同步、结果裁决的时间开销分别作了分析。另外文献[58]也排除了如变体多样化开销在内的其他干扰因素，测试出了同步技术的性能开销。

同步技术还有其他的评判指标，比如运行空间是用户还是内核，方法实现是否要求修改内核代码或者应用程序代码，修改量有多大等。

4.4 同步技术分析模型

本节提出一个数学模型用于描述同步技术，可用于同步技术的设计和初步评估。定义包含q个执行体的集合为$C=(C_1,C_2,\cdots,C_q)$。假设b_k表示执行体在不同层次(应用软件、操作系统、硬件结构等)上的非相似边界，其中$0<k<m$，m表示总的层次数。对于任意b_k，其边界内的待同步元素可表

示为a_{ky}，其中$0<y<n$，n表示待同步元素总数，则$b_k=(a_{k1},\cdots,a_{ky},\cdots,a_{kn})$。执行体$C_q$的非相似边界可表示为

$$B=\sum_{k=1}^{m}b_k=\sum_{k=1}^{m}\sum_{y=1}^{n}a_{ky} \tag{3}$$

这里用$M(a_{ky})$表示a_{ky}单次同步的系统通信开销，则执行体C_i的同步通信开销可以表示为

$$F_{C_i}=\sum_{k=1}^{m}\sum_{y=1}^{n}M(a_{ky})\delta_1(k,y)\delta_2(k,y) \tag{4}$$

其中，二元函数$\delta_1(k,y)=1$表示对a_{ky}元素进行同步，否则$\delta_1(k,y)=0$；$\delta_2(k,y)$表示a_{ky}元素的同步次数占总同步次数的概率，则$0\le\delta_2(k,y)<1$，$\sum_{k=1}^{m}\sum_{y=1}^{n}\delta_2(k,y)=1$。执行体集进行同步的通信开销可计算为

$$\mathrm{Com}=\sum_{i=1}^{q}F_{C_i} \tag{5}$$

用$T_{C_i}(a_{ky})$表示执行体C_i同步a_{ky}元素时间开销，则系统同步a_{ky}元素的时间开销为

$$T(a_{ky})=\max\{T_{C_i}(a_{ky})\},C_i\in C \tag{6}$$

执行体集进行同步的时间开销可表示为

$$\mathrm{Time}=\sum_{k=1}^{m}\sum_{y=1}^{n}T(a_{ky})\delta_1(k,y)\delta_2(k,y) \tag{7}$$

综上所述，系统同步产生的总开销可表示为

$$E=\alpha\mathrm{Com}+(1-\alpha)\mathrm{Time},(0<\alpha<1) \tag{8}$$

系统的同步率可以表示为

$$S=\sum_{k=1}^{m}\sum_{y=1}^{n}\delta_1(k,y)\delta_2(k,y) \tag{9}$$

在非相似冗余系统中，如果半数以上的执行体行为一致，则系统就是安全可靠的。前面定义了执行体集合$C=(C_1,C_2,\cdots,C_q)$，$\lceil q/2\rceil$表示向上取整，则系统安全性可以表示为

$$\begin{aligned}H=&\sum_{i=1}^{C_q^{\lceil q/2\rceil}}\phi_i^{\lceil q/2\rceil}(C_1,C_2,\cdots,C_q)\\&+\sum_{i=1}^{C_q^{\lceil q/2\rceil+1}}\phi_i^{\lceil q/2\rceil+1}(C_1,C_2,\cdots,C_q)+\cdots\\&+\sum_{i=1}^{C_q^{q}}\phi_i^{q}(C_1,C_2,\cdots,C_q)\end{aligned} \tag{10}$$

当从执行体集$C=(C_1,C_2,\cdots,C_q)$中选出$\lceil q/2\rceil$个执行体，且第i个排列组合同步执行体个数为$\lceil q/2\rceil$时，二元函数$\phi_i^{\lceil q/2\rceil}(C_1,C_2,\cdots,C_q)=\lceil q/2\rceil\cdot S$；否则，$\phi_i^{\lceil q/2\rceil}(C_1,C_2,\cdots,C_q)=0$。在这里，$\phi_i^{\lceil q/2\rceil}(C_1,C_2,\cdots,C_q)$的取值仅代表了系统安全性的趋势，也就是越多的执行体达到同步，系统就越安全可靠。

综上分析可知，式(8)表示的系统同步开销函数和式(10)表示的系统安全性函数，是相互矛盾的，当同步执行体数量越多时，系统安全性越高，但同步开销越大；同理，当同步执行体数量越少时，同步开销越低，但系统安全性减小。同步技术设计是一个多目标优化问题，需求目标的帕累托最优解。因此，具体制定什么样的同步方案，一是要看架构能提供的基础条件，二是要看架构想要达成的目标。架构所能提供的基础条件指非相似的边界(主要受成本约束)，包括纯应用软件层次、操作系统及以上层次和硬件及以上层次。非相似的边界越大，引发不同步的因素就越多，同步的通信开销越大，需要减少同步点的数量，牺牲同步效果来获取较高的系统效率。架构的目标包括安全性和实时性，对安全要求更高的系统，往往要求同步准确率更高，这需要增加同步点数或者种类。对实时要求更高的系统，则要求同步的效率更高，需要减少同步点数量及种类。

5 执行体同步技术未来发展

从非相似冗余架构诞生之日起，执行体之间的同步问题就是架构研究中的关键问题。并且随着异构冗余边界的增大，同步问题也变得更加复杂[65]。虽然同步技术的同步粒度、同步效果和同步效率之间存在折衷，但总体上，同步技术还是在沿着一个综合效率更高的路线发展的。这伴随着同步点的选择越来越精细。研究者付出更多的精力鉴别必要和不必要的同步，减少因设置不必要的同步点带来的系统开销的同时，不损失系统的准确性和安全性。另外，同步点的选择也更多呈现多维融合的特点，从设置单纯的系统调用级的同步点或单纯的I/O 级的同步点，到不同维度的同步点相结合，用更少的同步次数，达到了更高的同步精度。

由于同步技术与架构技术常常紧密结合，因此，本章节也结合架构的发展来讨论同步技术的发展。

在系统安全领域，应用非相似冗余架构的研究，依然十分活跃，研究者依然在尝试不同粒度、不同层次、不同维度的非相似冗余，来解决系统面临的不同种类攻击和问题，如入侵检测[66]、拒绝服务攻击[67]、共因失效问题[68,69]、云应用安全[70]等

等。除此之外，一些新的快速发展的研究领域，也为非相似冗余架构提供了研究空间。

网络弹性工程领域。随着网络安全威胁种类越来越多，规模越来越大，难以预测且高速演变，人们逐渐意识到要保证网络空间绝对安全是不现实的。网络安全的重点从阻止网络事故的发生向缓解事故带来的危害发展，从抵御攻击向保障业务连续可用发展，从而催生出了网络弹性(cyber resilience)的概念。网络弹性于2010年由美国MITRE研究所发表的论文[71]提出，文中探讨了网络弹性架构的设想。相比于传统网络安全思想，网络弹性更关注自身的健壮性与可靠性，更倾向与网络攻击共存。实现网络弹性目标的关键技术中就包括了多性样和冗余技术以及其他关联技术。在弹性网络的策略和设计原则中，公共关键资产保护、敏捷性和架构适应性、分层防御、资源位置多样化、改变或破坏攻击表面、预防对手进化等都需要多样性和冗余技术的支持。网络弹性技术在体系架构上没有取得明显突破，现在是一个活跃的研究领域，欧美各国都出台了弹性网络相关法案，要求数字产品制造商实施网络弹性设计，学术界也正在积极探索[72–74]。可以预见，非相似冗余架构在网络弹性领域将会迎来非常广泛的应用。弹性网络设计目标存在优先级，对应了非相似冗余架构和同步技术在安全性、准确率和效率之间的不同折衷方案，会进一步刺激架构和同步技术往更灵活的方向发展。比如，原来的同步技术中，同步粒度通常不可变，但面对弹性网络的多样化需求，可能需要同步粒度可动态调整，为系统提供可变的安全等级。

软件定义晶上系统领域。前面提到非相似冗余架构的发展从软件到硬件，非相似冗余的边界在逐步扩大，硬件更多地参与到架构设计中来。但是硬件的加入，给非相似冗余带来了两大挑战，一是硬件多样性增加了架构实现复杂性，二是更多的硬件资源投入，会导致系统成本大幅增加。软件定义晶上系统采用了通过体系创新来突破工艺局限的路线，用完整的晶圆基板来做互连底座，连接多个、多种芯粒实现一个完整的系统，芯粒的功能和互连支持软件定义。在晶圆上直接集成系统，可以极大减少芯粒间的连接损耗。目前，晶上系统技术并不成熟[75]，理论上，其集成性能可以达到数量级的提升。如果基础性研究取得突破，那么利用晶上系统来实现非相似冗余将获得性能上的优势。无论是同构硬件的软件非相似冗余，还是异构硬件的非相似冗余，晶上系统都可以提供足够的研究空间，刺激非相似冗余关键技术有更多方向的尝试。比如，与深度学习算法结合，使同步判决过程智能化。晶上系统的互连方式和互连结构，会影响执行体同步通信方式。再加上晶上系统的结构可定义，也给非相似冗余架构增加了动态变结构的可能，将给执行体的监控和同步带来新的挑战。

6　结束语

本文对高可靠信息系统非相似冗余架构进行了一般性的描述，并对其中存在的执行体同步技术进行了抽象建模，指出了同步技术在非相似冗余架构中的重要研究价值，是保障非相似冗余架构可行性、安全性的重要前提。然后，本文提出了基于同步点的执行体同步技术分类方法，对执行体同步技术进行了主流种类分析，对每一种类技术的基本方式、流行度、优缺点进行了总结。本文也指出，同步技术的研究重点，在于根据架构基础，选择同步点、控制不确定性因素、提高性能，进行最接近架构目标的折衷设计。同时给出了同步粒度、同步效果和同步效率之间的制约关系。最后，本文结合网络弹性工程领域发展指出，网络安全正从使用侧向设计侧转型，各国都开始将数字产业源头作为平衡网络安全风险与责任的抓手。采用非相似冗余架构，是一种可有效防御攻击，且富有弹性的网络安全手段，符合网络安全技术发展转型的趋势。而软件定义晶上系统通过晶上集成体系创新来突破工艺局限，可以为非相似冗余架构这种需要消耗更多面积、更多能量的安全手段提供的更充分的基础条件。作为非相似冗余架构中的关键技术，同步技术将持续向更精细、更准确、更高效的方向发展并会面临新的挑战。

参 考 文 献

[1] STAMP M. Risks of monoculture[J]. *Communications of the ACM*, 2004, 47(3): 120. doi: 10.1145/971617.971650.

[2] 王刚, 冯云, 陆世伟, 等. 多操作系统异构网络的病毒传播模型和安全性能优化策略[J]. 电子与信息学报, 2020, 42(4): 972–980. doi: 10.11999/JEIT190360.
WANG Gang, FENG Yun, LU Shiwei, *et al*. Virus propagation model and security performance optimization strategy of multi-operating system heterogeneous network[J]. *Journal of Electronics & Information Technology*, 2020, 42(4): 972–980. doi: 10.11999/JEIT190360.

[3] 杜三, 舒辉, 康绯. 基于硬件的动态指令集随机化框架的设计与实现[J]. 网络与信息安全学报, 2017, 3(11): 29–39. doi: 10.11959/j.issn.2096-109x.2017.00216.
DU San, SHU Hui, and KANG Fei. Design and implementation of hardware-based dynamic instruction set

randomization framework[J]. *Chinese Journal of Network and Information Security*, 2017, 3(11): 29–39. doi: 10.11959/j.issn.2096-109x.2017.00216.

[4] CHRISTOU G, VASILIADIS G, PAPAEFSTATHIOU V, *et al*. On architectural support for instruction set randomization[J]. *ACM Transactions on Architecture and Code Optimization*, 2020, 17(4): 36. doi: 10.1145/3419841.

[5] 张贵民, 李清宝, 曾光裕, 等. 运行时代码随机化防御代码复用攻击[J]. 软件学报, 2019, 30(9): 2772–2790. doi: 10.13328/j.cnki.jos.005516.
ZHANG Guimin, LI Qingbao, ZENG Guangyu, *et al*. Defensing code reuse attacks using live code randomization[J]. *Journal of Software*, 2019, 30(9): 2772–2790. doi: 10.13328/j.cnki.jos.005516.

[6] 何红旗, 王奕森, 董卫宇, 等. 基于编译置换的指令随机化系统设计与实现[J]. 计算机应用与软件, 2017, 34(12): 313–320. doi: 10.3969/j.issn.1000-386x.2017.12.059.
HE Hongqi, WANG Yisen, DONG Weiyu, *et al*. Design and implementation of instruction randomization based on compiling substitution[J]. *Computer Applications and Software*, 2017, 34(12): 313–320. doi: 10.3969/j.issn.1000-386x.2017.12.059.

[7] KIL C, JUN J, BOOKHOLT C, *et al*. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software[C]. The 22nd Annual Computer Security Applications Conference, Miami Beach, USA, 2006: 339–348. doi: 10.1109/ACSAC.2006.9.

[8] WANG Ye, LI Qingbao, CHEN Zhifeng, *et al*. Shapeshifter: Intelligence-driven data plane randomization resilient to data-oriented programming attacks[J]. *Computers & Security*, 2020, 89: 101679. doi: 10.1016/j.cose.2019.101679.

[9] 侯尚文, 黄建军, 梁彬, 等. 一种基于实时代码装卸载的代码重用攻击防御方法[J]. 计算机科学, 2022, 49(10): 279–284. doi: 10.11896/jsjkx.220500091.
HOU Shangwen, HUANG Jianjun, LIANG Bin, *et al*. Defense method against code reuse attack based on real-time code loading and unloading[J]. *Computer Science*, 2022, 49(10): 279–284. doi: 10.11896/jsjkx.220500091.

[10] BERGER E D and ZORN B G. DieHard: Probabilistic memory safety for unsafe languages[C]. The 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, Canada, 2006. doi: 10.1145/1133981.1134000.

[11] 马博林, 张铮, 陈源, 等. 基于指令集随机化的抗代码注入攻击方法[J]. 信息安全学报, 2020, 5(4): 30–43. doi: 10.19363/J.cnki.cn10-1380/tn.2020.07.03.
MA Bolin, ZHANG Zheng, CHEN Yuan, *et al*. The defense method for code-injection attacks based on instruction set randomization[J]. *Journal of Cyber Security*, 2020, 5(4): 30–43. doi: 10.19363/J.cnki.cn10-1380/tn.2020.07.03.

[12] 张宇嘉, 庞建民, 张铮, 等. 基于软件多样化的拟态安全防御策略[J]. 计算机科学, 2018, 45(2): 215–221. doi: 10.11896/j.issn.1002-137X.2018.02.037.
ZHANG Yujia, PANG Jianmin, ZHANG Zheng, *et al*. Mimic security defence strategy based on software diversity[J]. *Computer Science*, 2018, 45(2): 215–221. doi: 10.11896/j.issn.1002-137X.2018.02.037.

[13] 姚东, 张铮, 张高斐, 等. MVX-CFI: 一种实用的软件安全主动防御架构[J]. 信息安全学报, 2020, 5(4): 44–54. doi: 10.19363/J.cnki.cn10-1380/tn.2020.07.04.
YAO Dong, ZHANG Zheng, ZHANG Gaofei, *et al*. MVX-CFI: A practical active defense framework for software security[J]. *Journal of Cyber Security*, 2020, 5(4): 44–54. doi: 10.19363/J.cnki.cn10-1380/tn.2020.07.04.

[14] CHEN Liming and AVIZIENIS A. N-Version programminc: A fault-tolerance approach to reliability of software operation[C]. Proceedings of the 25th International Symposium on Fault-Tolerant Computing, Pasadena, USA, 1995. doi: 10.1109/FTCSH.1995.532621.

[15] 魏帅, 于洪, 顾泽宇, 等. 面向工控领域的拟态安全处理机架构[J]. 信息安全学报, 2017, 2(1): 54–73. doi: 10.19363/j.cnki.cn10-1380/tn.2017.01.005.
WEI Shuai, YU Hong, GU Zeyu, *et al*. Architecture of mimic security processor for industry control system[J]. *Journal of Cyber Security*, 2017, 2(1): 54–73. doi: 10.19363/j.cnki.cn10-1380/tn.2017.01.005.

[16] 宋克, 刘勤让, 魏帅, 等. 基于拟态防御的以太网交换机内生安全体系结构[J]. 通信学报, 2020, 41(5): 18–26. doi: 10.11959/j.issn.1000-436x.2020098.
SONG Ke, LIU Qinrang, WEI Shuai, *et al*. Endogenous security architecture of Ethernet switch based on mimic defense[J]. *Journal on Communications*, 2020, 41(5): 18–26. doi: 10.11959/j.issn.1000-436x.2020098.

[17] 王禛鹏, 扈红超, 程国振. 一种基于拟态安全防御的DNS框架设计[J]. 电子学报, 2017, 45(11): 2705–2714. doi: 10.3969/j.issn.0372-2112.2017.11.018.
WANG Zhenpeng, HU Hongchao, and CHENG Guozhen. A DNS architecture based on mimic security defense[J]. *Acta Electronica Sinica*, 2017, 45(11): 2705–2714. doi: 10.3969/j.issn.0372-2112.2017.11.018.

[18] 黄俊贤. 基于拟态防御架构的抗缓存投毒DNS系统研究[D]. [硕士论文], 华南理工大学, 2022. doi: 10.27151/d.cnki.ghnlu.2022.004792.
HUANG Junxian. A research on DNS against cache poisoning based on mimic defense architecture[D]. [Master dissertation], South China University of Technology, 2022. doi: 10.27151/d.cnki.ghnlu.2022.004792.

[19] 任权, 邬江兴, 贺磊. 基于GSPN的拟态DNS构造策略研究[J]. 信息安全学报, 2019, 4(2): 37–52. doi: 10.19363/J.cnki.cn10-1380/tn.2019.03.05.

REN Quan, WU Jiangxing, and HE Lei. Research on mimic DNS architectural strategy based on generalized stochastic petri net[J]. *Journal of Cyber Security*, 2019, 4(2): 37–52. doi: 10.19363/J.cnki.cn10-1380/tn.2019.03.05.

[20] 仝青, 张铮, 张为华, 等. 拟态防御Web服务器设计与实现[J]. 软件学报, 2017, 28(4): 883–897. doi: 10.13328/j.cnki.jos.005192.
TONG Qing, ZHANG Zheng, ZHANG Weihua, *et al*. Design and implementation of mimic defense Web server[J]. *Journal of Software*, 2017, 28(4): 883–897. doi: 10.13328/j.cnki.jos.005192.

[21] 马海龙, 伊鹏, 江逸茗, 等. 基于动态异构冗余机制的路由器拟态防御体系结构[J]. 信息安全学报, 2017, 2(1): 29–42. doi: 10.19363/j.cnki.cn10-1380/tn.2017.01.003.
MA Hailong, YI Peng, JIANG Yiming, *et al*. Dynamic heterogeneous redundancy based router architecture with mimic defenses[J]. *Journal of Cyber Security*, 2017, 2(1): 29–42. doi: 10.19363/j.cnki.cn10-1380/tn.2017.01.003.

[22] 马海龙, 尹梓诺, 胡涛. 面向异构化平台的轻量级程序异常检测方法[J]. 电子与信息学报, 2022, 44(2): 602–610. doi: 10.11999/JEIT210152.
MA Hailong, YIN Zinuo, and HU Tao. A lightweight program anomaly detection method for heterogeneous platform[J]. *Journal of Electronics & Information Technology*, 2022, 44(2): 602–610. doi: 10.11999/JEIT210152.

[23] MILLS B, ZNATI T, and MELHEM R. Shadow computing: An energy-aware fault tolerant computing model[C]. 2014 International Conference on Computing, Networking and Communications, Honolulu, USA, 2014: 73–77,doi: 10.1109/ICCNC.2014.6785308.

[24] SHYE A, BLOMSTEDT J, MOSELEY T, *et al*. PLR: A software approach to transient fault tolerance for multicore architectures[J]. *IEEE Transactions on Dependable and Secure Computing*, 2009, 6(2): 135–148. doi: 10.1109/TDSC.2008.62.

[25] ITURBE X, VENU B, OZER E, *et al*. The arm triple core lock-step (TCLS) processor[J]. *ACM Transactions on Computer Systems*, 2018, 36(3): 7. doi: 10.1145/3323917.

[26] SALAMA A, BINNIG C, KRASKA T, *et al*. Cost-based fault-tolerance for parallel data processing[C]. The 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Australia, 2015: 285–297. doi: 10.1145/2723372.2749437.

[27] JAYASEKARA S, HARWOOD A, and KARUNASEKERA S. A utilization model for optimization of checkpoint intervals in distributed stream processing systems[J]. *Future Generation Computer Systems*, 2020, 110: 68–79. doi: 10.1016/j.future.2020.04.019.

[28] COX B and EVANS D. N-Variant systems: A secretless framework for security through diversity[C]. The 15th USENIX Security Symposium, Vancouver, Canada, 2006: 105–120.

[29] YUMEREFENDI A R, MICKLE B, and COX L P. TightLip: Keeping applications from spilling the beans[C]. The 4th Symposium on Networked Systems Design and Implementation, Cambridge, USA, 2007: 25–31.

[30] SALAMAT B, JACKSON T, GAL A, *et al*. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space[C]. The 4th ACM European Conference on Computer Systems, Nuremberg, Germany, 2009. doi: 10.1145/1519065.1519071.

[31] VOLCKAERT S, DE SUTTER B, DE BAETS T, *et al*. GHUMVEE: Efficient, effective, and flexible replication[C]. The 15th International Symposium on Foundations and Practice of Security, Montreal, Canada, 2013. doi: 10.1007/978-3-642-37119-6_17.

[32] VOLCKAERT S, COPPENS B, and DE SUTTER B. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution[J]. *IEEE Transactions on Dependable and Secure Computing*, 2016, 13(4): 437–450. doi: 10.1109/TDSC.2015.2411254.

[33] VOLCKAERT S, COPPENS B, VOULIMENEAS A, *et al*. Secure and efficient application monitoring and replication[C]. The 2016 USENIX Conference on Usenix Annual Technical Conference, Denver, USA, 2016: 167–179.

[34] KONING K, BOS H, and GIUFFRIDA C. Secure and efficient multi-variant execution using hardware-assisted process virtualization[C]. The 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Toulouse, France, 2016: 431–442. doi: 10.1109/DSN.2016.46.

[35] XU Meng, LU Kangjie, KIM T, *et al*. Bunshin: Compositing security mechanisms through diversification[C]. The 2017 USENIX Conference on Usenix Annual Technical Conference, Santa Clara, United States, 2017.

[36] VOULIMENEAS A, SONG D, PARZEFALL F, *et al*. DMON: A distributed heterogeneous n-variant system[EB/OL]. https://arxiv.org/abs/1903.03643, 2019.

[37] WANG Xiaoguang, YEOH S, LYERLY R, *et al*. A framework to secure applications with ISA heterogeneity[C]. The SFMA Workshop'19, Dresden, Germany, 2019.

[38] LU Kangjie, XU Meng, SONG Chengyu, *et al*. Stopping memory disclosures via diversification and replicated execution[J]. *IEEE Transactions on Dependable and Secure Computing*, 2021, 18(1): 160–173. doi: 10.1109/TDSC.2018.2878234.

[39] VOULIMENEAS A, SONG D, LARSEN P, *et al*. dMVX: Secure and efficient multi-variant execution in a distributed setting[C]. Proceedings of the 14th European Workshop on

Systems Security, United Kingdom, 2021. doi: 10.1145/3447852.3458714.

[40] BERGER E D and ZORN B G. DieHard: Efficient probabilistic memory safety[J]. *ACM Transactions on Computers*, 2007.

[41] NOVARK G and BERGER E D. DieHarder: Securing the heap[C]. The 17th ACM Conference on Computer and Communications Security, Chicago, USA, 2010. doi: 10.1145/1866307.1866371.

[42] NOVARK G, BERGER E D, and ZORN B G. Exterminator: Automatically correcting memory errors with high probability[J]. *Communications of the ACM*, 2008, 51(12): 87–95. doi: 10.1145/1409360.1409382.

[43] CAPIZZI R, LONGO A, VENKATAKRISHNAN V N, *et al*. Preventing information leaks through shadow executions[C]. 2008 Annual Computer Security Applications Conference, Anaheim, USA, 2008: 102–110. doi: 10.1109/ACSAC.2008.50.

[44] RICHARDSON T, STAFFORD-FRASER Q, WOOD K R, *et al*. Virtual network computing[J]. *IEEE Internet Computing*, 1998, 2(1): 33–38. doi: 10.1109/4236.656066.

[45] ÖSTERLUND S, KONING K, OLIVIER P, *et al*. kMVX: Detecting kernel information leaks with multi-variant execution[C]. The Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Providence, USA, 2019: 559–572. doi: 10.1145/3297858.3304054.

[46] 苏野, 魏帅, 姚领彦, 等. 基于相对时间的异构执行体程序状态同步方法[C]. 第三届“先进计算与内生安全”学术会议论文集, 南京, 中国, 2020: 716–725.
SU Ye, WEI Shuai, YAO Lingyan, *et al*. Program state synchronization method of heterogeneous executors based on relative time[C]. The 3rd Conference on Advanced Computing and Endogenous Safety & Security, Nanjing, China, 2020: 716–725.

[47] KIM D, KWON Y, SUMNER W N, *et al*. Dual execution for on the fly fine grained execution comparison[C]. The Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, Istanbul, Turkey, 2015: 325–338. doi: 10.1145/2694344.2694394.

[48] XIN Bin, SUMNER W N, and ZHANG Xiangyu. Efficient program execution indexing[C]. The 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, USA, 2008: 238–248. doi: 10.1145/1375581.1375611.

[49] KIM D, SUMNER W N, ZHANG Xiangyu, *et al*. Reuse-oriented reverse engineering of functional components from x86 binaries[C]. The 36th International Conference on Software Engineering, Hyderabad, India, 2014: 1128–1139. doi: 10.1145/2568225.2568296.

[50] GAWLIK R, KOPPE P, KOLLENDA B, *et al*. Detile: Fine-grained information leak detection in script engines[C]. The 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, San Sebastián, Spain, 2016: 322–342. doi: 10.1007/978-3-319-40667-1_16.

[51] HOSEK P and CADAR C. Safe software updates via multi-version execution[C]. Proceedings of 2013 35th International Conference on Software Engineering, San Francisco, USA, 2013: 612–621. doi: 10.1109/ICSE.2013.6606607.

[52] HOSEK P and CADAR C. VARAN the unbelievable: An efficient N-version execution framework[C]. The 20th International Conference on Architectural Support for Programming Languages and Operating Systems, Istanbul, Turkey, 2015. doi: 10.1145/2694344.2694390.

[53] KWON Y, KIM D, SUMNER W N, *et al*. LDX: Causality inference by lightweight dual execution[C]. The 21st International Conference on Architectural Support for Programming Languages and Operating Systems, Atlanta, USA, 2016. doi: 10.1145/2872362.2872395.

[54] LEE D, WESTER B, VEERARAGHAVAN K, *et al*. Respec: Efficient online multiprocessor replayvia speculation and external determinism[J]. *ACM SIGARCH Computer Architecture News*, 2010, 38(1): 77–90. doi: 10.1145/1735970.1736031.

[55] XU Jun, GUO Pinyao, CHEN Bo, *et al*. Demo: A symbolic n-variant system[C]. The 2016 ACM Workshop on Moving Target Defense, Vienna, Austria, 2016. doi: 10.1145/2995272.2995284.

[56] BRUSCHI D, CAVALLARO L, and LANZI A. Diversified process replicæ for defeating memory error exploits[C]. 2007 IEEE International Performance, Computing, and Communications Conference, New Orleans, USA, 2007. doi: 10.1109/PCCC.2007.358924.

[57] CAVALLARO L. Comprehensive memory error protection via diversity and taint-tracking[D]. [Ph. D. dissertation], Università Degli Studi di Milano, 2007.

[58] VOLCKAERT S, COPPENS B, DE SUTTER B, *et al*. Taming parallelism in a multi-variant execution environment[C]. The Twelfth European Conference on Computer Systems, Belgrade, Serbia, 2017. doi: 10.1145/3064176.3064178.

[59] 李秉政, 张铮, 马博林, 等. 编译支持的多变体融合执行设计与实现[J]. 信息安全学报, 2022, 7(4): 114–123. doi: 10.19363/J.cnki.cn10-1380/tn.2022.07.09.
LI Bingzheng, ZHANG Zheng, MA Bolin, *et al*. Design and implementation of integrated multi-variant execution supported by compiler[J]. *Journal of Cyber Security*, 2022, 7(4): 114–123. doi: 10.19363/J.cnki.cn10-1380/tn.2022.07.09.

[60] EL-ZOGHBY A M, ELSAYED M S, JURCUT A D, *et al*.

NG-MVEE: A new proposed hybrid technique for enhanced mitigation of code re-use attack[J]. *IEEE Access*, 2023, 11: 48169–48191. doi: 10.1109/ACCESS.2023.3269881.

[61] 张明权, 于洪, 魏帅, 等. 基于拟态的MCU设计及应用验证[C]. 第三届先进计算与内生安全学术会议论文集, 南京, 中国, 2020: 299–305. **(查阅网上资料, 未找到本条文献信息, 请确认)**.

ZHANG Mingquan, YU Hong, WEI Shuai, *et al*. MCU design and application verification based on mimic defense[C]. Proceedings of the 3rd Conference on Advanced Computing and Endogenous Safety & Security, Nanjing, China, 2020: 299–305.

[62] 潘传幸, 张铮, 马博林, 等. 面向进程控制流劫持攻击的拟态防御方法[J]. 通信学报, 2021, 42(1): 37–47. doi: 10.11959/j.issn.1000-436x.2021013.

PAN Chuanxing, ZHANG Zheng, MA Bolin, *et al*. Method against process control-flow hijacking based on mimic defense[J]. *Journal on Communications*, 2021, 42(1): 37–47. doi: 10.11959/j.issn.1000-436x.2021013.

[63] SO H, DIDEHBAN M, KO Y, *et al*. EXPERTISE: An effective software-level redundant multithreading scheme against hardware faults[J]. *ACM Transactions on Architecture and Code Optimization*, 2022, 19(4): 53. doi: 10.1145/3546073.

[64] DE GROEF W, DEVRIESE D, NIKIFORAKIS N, *et al*. Secure multi-execution of web scripts: Theory and practice[J]. *Journal of Computer Security*, 2014, 22(4): 469–509. doi: 10.3233/JCS-130495.

[65] 姚远, 潘传幸, 张铮, 等. 多样化软件系统量化评估方法[J]. 通信学报, 2020, 41(3): 120–125. doi: 10.11959/j.issn.1000-436x.2020051.

YAO Yuan, PAN Chuanxing, ZHANG Zheng, *et al*. Method of quantitative assessment for diversified software system[J]. *Journal on Communications*, 2020, 41(3): 120–125. doi: 10.11959/j.issn.1000-436x.2020051.

[66] DUAN Jun, HAMLEN K W, and FERRELL B. Better late than never: An n-variant framework of verification for Java source code on CPU x GPU hybrid platform[C]. Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, Phoenix, USA, 2019. doi: 10.1145/3307681.3326604.

[67] JONES J, HISER J D, DAVIDSON J W, *et al*. Defeating denial-of-service attacks in a self-managing N-variant system[J]. *ACM Transactions on Software Engineering & Methodology*, 2019, 28(3): 126–138. **(查阅网上资料, 未找到本条文献信息, 请确认)**.

[68] BAS F, ALCAIDE S, LORENZO R, *et al*. SafeDE: A flexible diversity enforcement hardware module for light-lockstepping[C]. Proceedings of 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design, Torino, Italy, 2021: 1–7. doi: 10.1109/IOLTS52814.2021.9486715.

[69] BAS F, ALCAIDE S, CABO G, *et al*. SafeDE: A low-cost hardware solution to enforce diverse redundancy in multicores[J]. *IEEE Transactions on Device and Materials Reliability*, 2022, 22(2): 111–119. doi: 10.1109/TDMR.2022.3156799.

[70] ZHOU Dacheng, CHEN Hongchang, CHENG Guozhen, *et al*. SecIngress: An API gateway framework to secure cloud applications based on n-variant system[J]. *China Communications*, 2021, 18(8): 17–34. doi: 10.23919/JCC.2021.08.002.

[71] GOLDMAN H G. Building secure, resilient architectures for cyber mission assurance[R]. McLean: MITRE Corporation, 2010.

[72] ROSS R, PILLITTERI V, GRAUBART R, *et al*. Developing cyber-resilient systems: A systems security engineering approach[R]. NIST Special Publication, 2021. doi: 10.6028/NIST.SP.800-160v2. **(查阅网上资料,未找到对应的出版地信息,请确认)**.

[73] LINKOV I, LIGO A, STODDARD K, *et al*. Cyber efficiency and cyber resilience[J]. *Communications of the ACM*, 2023, 66(4): 33–37. doi: 10.1145/3549073.

[74] YADAV S B. A resilient hierarchical distributed model of a cyber physical system[J]. *Cyber-Physical Systems*, 2023, 9(2): 97–121. doi: 10.1080/23335777.2021. 1964101.

[75] 王明楠, 刘勤让, 刘冬培, 等. 一种晶上系统互连网络的容错感知结构[J]. 计算机应用研究, 2023, 40(2): 533–538 doi: 10.19734/j.issn.1001-3695.2022.06.0355.

WANG Mingnan, LIU Qinrang, LIU Dongpei, *et al*. Fault-tolerant awareness structure for network on wafer[J]. *Application Research of Computers*, 2023, 40(2): 533–538 doi: 10.19734/j.issn.1001-3695.2022.06.0355.

于　洪：女，副研究员，主要研究方向为网络空间安全、计算机网络体系结构.

刘勤让：男，研究员，博士生导师，主要研究方向为网络空间安全、宽带信息网络级芯片设计.

魏　帅：男，副教授，主要研究方向为计算机软件、网络体系结构.

兰巨龙：男，教授，博士生导师，研究方向为新型网络体系、网络动力学.

责任编辑：马秀强