单位代码 10006
学　　号 10211044
分 类 号 TP309.5

北京航空航天大學
BEIHANG UNIVERSITY

毕业设计 (翻译)

计算机病毒的理论与实验

学 院 名 称　软件学院
专 业 名 称　软件工程专业
学 生 姓 名　康乔
指 导 教 师　原仓周

2014 年 06 月

本人声明

我声明，本论文及其研究工作是由本人在导师指导下独立完成的，在完成论文时所利用的一切资料均已在参考文献中列出。

作者： 康乔

签字：

时间： 2014 年 06 月

计算机病毒的理论与实验

学　　生：康乔
指导教师：原仓周

摘　要

这篇文章介绍了“计算机病毒”并且检验了它们导致计算机系统的广泛破坏的潜力。文章展示了基本的理论结果、病毒防御在大型系统中的不可行性、防御方案以及一些实验结果。

关键词：计算机病毒，系统完整性，数据完整性

Computer Virus Theory and Experiments

Author: Kang Qiao
Tutor: Yuan Cangzhou

Abstract

This paper introduces computer virus and examines their potential for causing widespread damage to computer system. Basic therotical results are presented, and this infeasibility of viral defense in large classes of systems is shown. Defensive schemes are presented and several experiments are described.

Key words: Computer Virus, System Integrity, Data Integrity

目 录

1 介绍

这篇文章解释了一个主要的计算机安全性问题，即病毒。病毒的有趣之处在于它能依附于其他的程序并使它们也成为病毒。考虑到现代计算机系统的广泛共享，携带着木马程序 [1, 2] 的病毒的威胁越来越值得注意。尽管在阻止信息非法传播 [3, 4] 的执行政策上有相当多的工作已经完成，并且许多系统已经采用它们来避免这种类型的攻击 [5–8] ，但是在如何保持信息进入一个领域而不导致破坏方面的工作却很少 [9, 10]。在计算机系统中有很多不同类型的信息路径，其中一些是合法授权的，另一些可能是隐秘的 [10]，这常常被用户所忽略。在这篇文章中我们将忽略隐秘性的信息路径。

一般设施提供正确的保护方案 [11]，但是它们依赖于只对正在进行的攻击能有效防御的安全性政策。就算是一些相当简单的防护系统也不能被证明是“安全的”[12]。防止拒绝服务需要停止程序的检测 [13]。给系统中的信息流进行精确标记的问题已被证明是一个 NP-完全问题。对于在用户之间的不可信信息的传播，进行某种方式的防卫，这已经被测试过了 [14]，但是一般依赖能力来提高程序正确性，这也是一个众所周知的 NP 完全问题。

施乐公司的蠕虫程序已经说明，该程序可以通过网络繁殖，并且可以偶然地造成设备无法服务。在后来的变种中，一个“核心战争”的游戏中，能够让两个程序互相争斗。一些针对这一主题的，被匿名作者发布的其他的变种，是基于在程序之间进行的夜间游戏的上下文中的。术语病毒也被用于结合丰 APL 的作者的地方一般开始时调用每个函数反过来调用一个预处理器, 以增加默认 APL 翻译。

一个广泛传播的安全问题的潜在威胁被分析了[15]，对于政府、金融、商业和学术设施的危害是严重的。另外，这些机构倾向于使用特别的保护机制来应对这些特有的威胁，而不是研究全面而彻底的理论上的保护机制[16]。当前的军队保护系统，很大程度上依赖隔离机制[17]；然而，新的系统开始使用多层的用法[18]。没有一个发布出来的被建议使用的系统可以定义或者部署一种方法，用来阻止一种病毒。

这篇文章中我们提出了一个防止计算机病毒的新问题。首先我们检测了病毒的感染特性，并且展示了共享信息的传递闭包可以被感染。当和木马一起使用的时候，可以导致设备广泛停止服务以及数据的非授权的操作。一些对计算机病毒的实验结果表明，病毒对于普通的和高安全性的操作系统都是一个强大的威胁。传播的途径、信息流的传递以及信息解释的一般性是防止计算机病毒的关键特性，这些特性将在下面一一解释。分析表明，有潜力阻止病毒攻击的系统具有有限的传递性和有限的共享性，或者

完全不能共享也不具有一般的信息解释（图灵能力）。只有前一种情形对于现代社会具有研究的意义。一般来说，使用先验和运行时分析对病毒的防护的研究是不可判定的，并且没有防护，治愈是很难或者无法实现的。

我们检查了一些被提出来的对策，这些对策都针对特定的情景，来对病毒的特质进行就事论事的分析。限制传播系统被认为是值得期待的，但是精确的部署是很棘手的，并且模糊的政策一般来说会导致有用的系统随着时间越来越少。系统范围内的病毒抗体的使用也被测试了，但普遍来说，还是要依赖针对特定棘手问题的解决方案。

综上可知对计算机病毒是一个非常重要的研究领域，因为它和其他领域的潜在性应用。现代系统对于病毒攻击提供很少或者没有提供防护，并且当代唯一有效的“安全”措施就是隔离了。

2 计算机病毒

我们定义计算机“病毒”为一个程序，它可以复制自身来调整自己从而感染其他程序。通过感染特性，病毒可以通过计算机系统或者网络进行广泛传播，使用每一个用户的授权来感染他们的程序。每一个被感染的程序也可能会表现为病毒，从而使得感染扩散。

接下来的伪程序展示了一个病毒如何写成一个伪计算机语言。符号‘：=’表示定义，符号‘:’标注一个状态，符号‘;’用于分隔状态，符号‘=’用于赋值或者比较，符号‘ ’表示否，符号‘’和‘’将状态序列放在一起，符号‘…’表示代码的不相关部分已经略去了。

示例病毒（V）（图2.1）通过搜寻没有“1234567”开头的可执行文件来寻找一种不可感染可执行文件（E），将 V 加入 E，将它变成一个感染文件（I）。然后 V 检查一些触发条件和破坏是否是正确的。最后，V 执行剩余的前缀程序。当用户尝试执行 E，那么 I 会在它的部分执行；它会感染其他的文件并且把它当做 E 来执行。由于感染的轻微延迟异常，I 在触发条件导致破坏之前被看做是 E。我们注意到，病毒不需要预先考虑自己，也不会限制于每次使用的单个感染。

```
program virus :=
{1234567;
subroutine infect-executable :=
  {loop: file = random-executable;
  if first-line-of-file = 1234567
       then goto loop;
  prepend virus to file;
  }
subroutine do-damage :=
  {whatever damage is desired}
subroutine trigger-pulled :=
  {return true on desired conditions}
main-program :=
  {infect-executable;
  if trigger-pulled then do-damage;
  goto next;
  }
next:}
```

图 2.1 Simple Virus V

病毒的一个常见的误解是它的程序简单地通过网络传播。蠕虫程序，“核心战争”，和其他类似的项目做到这一点，但实际上没有人涉及到感染的部分。病毒的关键特性

是它感染其他程序的能力，从而达到共享用户之间的传递闭包。例如，如果 V 感染用户的一个可执行文件（E），然后用户 B 也运行 E，那么 V 可能蔓延至用户 B 的文件。

应该指出，病毒不可以用于邪恶的目的或者是成为一个木马。例如，一个压缩病毒可以写来用于找到未受感染的可执行文件，在用户许可的情况下压缩它们，并且预先考虑自己而非它们。在执行时，受感染的程序进行解压，然后正常执行。因为它总是在执行服务之前寻求允许，所以这不是一个木马，但因为它有感染特性，所以它仍然是一个病毒。研究表明，在平均系统中，这种病毒可能节省由可执行文件占据的超过 50% 的空间。被感染的程序的表现将下降，这是因为它们被解压了，从而压缩病毒实现一个特定的时间与空间的权衡。病毒样本压缩可以编写如下，见图2.2。

```
program compression-virus :=
{01234567;

subroutine infect-executable :=
  {loop: file = random-executable;
  if first-line-of-file = 01234567
        then goto loop;
  compress file;
  prepend compression-virus to file;
  }

main-program :=
  {if ask-permission
        then infect-executable;
  uncompress the-rest-of-this-file
        into tmpfile;
  run tmpfile;
  }
}
```

图 2.2 Compression Virus C

这个程序（C）找到了一个未感染可执行文件（E），压缩它，并且突出显示 C 形成一个感染可执行文件（I）。然后将剩余的自己解压为一个临时文件并且可以正常执行。当运行 I 时，在将 I 解压成一个临时文件盒执行之前，它会寻找并压缩其他可执行文件。传播的效果是通过系统压缩可执行文件，并解压它们进行执行。用户将有重大延迟，因为他们的可执行文件是在解压后才运行的。

一个更具威胁性的例子中，我们假设通过以下的方法修改程序 V，一是指定在一个给定的数据和时间进行触发，二是指定通过无限循环的方式进行破坏。在大多数现代系统的共享水平下，整个系统可能无法在指定的日期和时间条件下使用。可能需要大量的工作来消除这种病毒的破坏。这一修改如图2.3所示。

作为计算机病毒的一个类比，可以考虑一个可以 100% 感染的生物疾病，每当动物交流时就会传播，在给定的时刻立即杀死所有被感染的动物，并且在那个时刻之前没有检测到副作用。如果疾病的引入和见效之间存在一周的延迟，将很有可能只有少数

```
" " "
subroutine do-damage :=
  {loop: goto loop;}
subroutine trigger-pulled :=
  {if year > 1984 then return(true)
        otherwise return(false);
" " "
```

图 2.3 A denial of services virus

偏远村庄能够幸存，并肯定会消灭绝大多数的现代社会。如果类似于这种类型的电脑病毒可以传遍世界所有的计算机，它可能会在很长的时间阻止大多数电脑的使用，并且在现代政府、金融、商业和学术机构中造成很大的破坏。

3 计算机病毒的预防

我们已经向读者介绍了病毒的概念和对于系统的实际病毒。已经种下潜在的毁灭性袭击的种子，所以适当的检查保护机制可以帮助抵御它。我们在这里研究计算机病毒的预防。

3.1 基本的限制

为了系统中的用户能够共享信息，必须存在一个信息路径可以让信息从一个用户传送到另一个用户。我们不区分用户和作为用户代理的程序，这是因为在任何计算机使用中程序总是作为用户的代理，而且我们忽略通过用户的秘密通道。假设有一个计算的图灵机模型，我们可以证明如果信息可以由用户通过图灵能力进行读取，那么它就可以被复制，复制后可以被视为一个图灵机磁带上的数据。

给定一个通用的系统，用户可以按照自己的希望使用自己具有的信息，并且传递这些他们认为合适的信息，可以清楚的看到，分享信息的能力是可传递的。也就是说，如果有一条从用户 A 到用户 B 的路径，还有一条从用户 B 到用户 C 的路径，那么在用户 B 有意或者不知情的合作下，存在从用户 A 到用户 C 之间的路径。

最后，可以用作数据和程序的信息之间没有本质的区别。这可以清楚的在翻译的情况下看出来，信息编辑成数据，并译作一个程序。实际上，信息只有在译出之后才有意义。

在一个信息可以被接受者译作一个程序的系统中，这种翻译可能导致如上所示的感染。如果存在分享，感染可以通过共享信息的解释进行传播。如果对信息流的传递性没有限制，那么从任何资源开始，信息可以到达信息流的传递闭包。共享、信息流的传递性和解释的普遍性使得在任何给定的资源的条件下，病毒可以传播到信息流的传递闭包。

显然，如果没有共享，就没有跨边界信息的传染，因此外部的信息不能解释，并且病毒不能在单个分区之外进行传播。这就是所谓的“隔离”。显然，系统中如果没有程序可以改变并且没有信息用于决策，那么系统不会被感染，这是因为感染需要解释信息的修正。我们称之为“固定一阶函数”系统。我们应该注意到，几乎任何具有实际效用的科学系统或开发环境需要解释的普遍性，如果我们希望从别人的工作中获益，那么隔离是不可接受的。然而，在有限的情况下可能存在病毒问题的解决办法。

3.2 分区模型

可以将限制信息流动的路径分为两类，一些将用户在传递条件下分割成适当的闭子集，另外一些则不进行分割。流量限制可以导致闭子集视为一个系统的分区并成孤立子系统。这让每一次感染限制在一个分区中。这是一个可行的预防手段，用于预防病毒在有限隔离下的完全占领，相当于给每个分区自己的计算机。

完整性模型 [9] 提供一种策略，它可用于在传递性条件下将系统分割成封闭子集。在 Biba 模型中，一个完整性层次与所有信息都相关联。严格的完整属性是 Bell-LaPadula 属性的双重叠加；在一个给定的完整性水平下，没有用户可以读取一个完整性水平更低的对象，也不能写出一个完整性水平更高的对象。在 Biba 最初的模型中，读取和执行访问之间是有区别的，但是在没有限制信息解释的普遍性的条件下不能执行，这是因为一个高完整性的程序可以编写一个低完整性的对象，并且对其进行复制，然后读取低完整性的输入并且产生低完整性的输出。

如果完整性模型和 Bell-LaPadula 模型共存，一种有限隔离可以通过传递性将空间划分为有限的闭子集。如果相同的划分用于机制 (高完整性与高安全性)，孤立主义结果信息提升安全水平也提升了完整性水平，但这是不被允许的。当 Biba 模型的边界在 Bell-LaPadula 模型的边界内的时候，感染只能在一个给定的安全水平下从高完整性水平向低完整性水平扩散。最后，当 Biba 模型的边界在 Bell-LaPadula 模型的边界内的时候，感染只能在一个给定的安全水平下从低完整性水平向高完整性水平扩散。对应于低边界性和高边界性，实际上有 9 种不同的情况，但下面图形显示了其中的三个，这已经可以让大家充分理解了。

Biba 的工作还包括另外两个完整性策略，“低水位标志”策略使得对于任何输入可以输出最低的完整性，以及“环”策略使得用户无法调用他们能够阅读的一切东西。前一个策略倾向于将所有信息转移到完整性水平较低处，而后者试图区别那些不能用广义解释的信息。

正如基于 Bell-LaPadula 模型的系统总是通过提高水平来使得所有信息转移到更高水平的安全性上，以此满足高水平的用户，而 Biba 模型倾向于通过减少最低传入结果的完整性将所有信息转移到较低的完整性水平上。我们也知道，一个精确系统的完整性是 NP-完全问题（正如它的对偶是 NP-完全的）。

最值得信赖的程序员（由定义）应该可以编写可以让大多数用户使用的程序。为了保证 Bell-LaPadula 策略，高水平用户无法编写由低水平的用户使用的程序。这意味着最信任的程序员必须是最低的安全级别的。这似乎是矛盾的。当我们把 Biba 和

Bell-LaPadula 模型联合起来的时候，我们发现产生的孤立主义保护我们免受病毒的攻击，但是不允许任何用户编写让整个系统使用的程序。但实际上，就像我们允许数据的加密或解密，把它从高安全水平转移到低安全性水平上，我们应该能够使用程序测试和验证将信息从低完整性水平转移到高完整性水平上。

另一个用于将系统分割成闭子集的常用的策略也用于典型的军事应用。这一策略将用户分类，每个用户只能访问他们的职责所需的信息。如果每一个用户在特定的时间只能一访问一个类别，系统可以避免跨类别边界病毒攻击，这是因为他们是孤立的。不幸的是，在目前的系统中，用户可以同时访问多个类别。在这种情况下，感染可以通过边界传播到信息流的传递闭包。

3.3 流模型

一些不通过传递性将系统分割成闭子集的策略，可以限制病毒传播的程度。“流距离”策略实现了通过记录数据流的距离（共享的数目）产生距离矩阵。输出信息的距离是输入信息的最大距离，并且共享信息的距离比信息共享之前的距离多一个。通过采用一个阈值使得在此之上的信息变得无法使用，以此引入保护。因此与距离 8 的文件共享一个距离为 2 的进程，可以将进程距离增加到 9，并且任何进一步的输出将至少有这个距离。

“流列表”策略维护所有对每个对象有影响的用户。规则维护该列表，即输出流的流列表是所有输入的流列表的并（包括导致行动的用户）。保护需要采用决定可访问性的任意布尔表达式的形式流列表。这是一个非常通用的策略，可以通过选择合适的布尔表达式来表示任何上述策略。

在例子中，用户 A 只能获取由用户 B 和 C 或者用户 B 和 D 写的信息，但是不能获取单独由 B 或者 C 或者 D 写的信息。它的用途使得在 C 或者 D 传递给 A 之前需要 B 的信息认证。流列表系统也可以用于 Biba 和距离模型。作为例子，距离模型可以由以下的形式实现：

将流列表用于流序列的进一步推广也是可能实现的，对于控制测流也似乎是最常用的。

在一个具有无限信息路径的系统中，如果用户不使用所有可用的路径，有限传递性也许有效，但是因为任何两个用户之间总有直接通路，所以总有感染的可能性。作为例子，在一个传递性仅限于 1 的系统中，与任何“可信赖”的用户分享信息是“安全的”，不必担心该用户是否错误地相信了另一个用户。

3.4 有限翻译

翻译的普遍性上的限制低于固定一阶翻译上的限制，能够感染仍然是一个悬而未决的问题，这是因为感染取决于功能允许。某些功能对于感染是需要的。写作能力是必需的，但任何有用的程序必须有输出。可以设计的一组操作，即使是在共享性和传递性最一般的情况下也不允许感染，但尚不清楚是否包括非固定一阶函数的集合。作为例子，一个只有“显示文件”功能的系统只能显示一个文件的内容给用户，并不能修改任何文件。在固定的数据库或邮件系统中，这可能有实际的应用，但肯定不是在开发环境中。在许多情况下，电脑邮件是充分的通信手段，只要电脑邮件系统与其他应用程序分区，除了秘密渠道用户他们之间就再也没有信息交流，这可能是用来防止感染。虽然没有固定的翻译方其案本身可以被感染，高阶固定翻译方案可以用来感染那些编写的用来解释它的程序。作为例子，电脑的微码可能是固定的，但机器语言中的代码解释仍然是可以被感染的。LISP，APL 以及 Basic 都是固定解释方案的例子，它们可以在一般方式下解释信息。因为他们的解释信息的能力具有一般性，所以可以在任何一个语言中编写一个程序，可以感染任何或所有这些语言中的程序。在有限的翻译系统中，感染不能传播得比一般的翻译系统更远，这是因为在受限系统中的每一个函数在还必须能够在一个一般系统中执行。因此前面的结果提供了病毒在具有有限翻译的系统中的传播的上界。

3.5 精度问题

虽然孤立主义和有限的传递性为感染问题提供了解决方案，但是他们并不理想，这是因为作为有价值的计算工具，广泛共享是普遍的。这些政策中只有孤立性可以在实践中精确实现，这是因为跟踪准确的信息流需要 NP-完全时间，同时维护标记需要大量的空间[4]。这使得我们有不精确的技术。不精确的技术的问题是他们往往将系统转移到孤立性。这是因为他们对效用使用保守估计以防止潜在的损害。它背后的哲学性说明安全比损害更加重要。问题是，当信息被不公正地被一个给定的用户认为是不可读的，此时该系统对于该用户不再那么有用了。这是一种拒绝服务，即获取可得到的信息时被拒绝访问。这样一个系统总是倾向于使自己越来越少的使用共享，直到它变得完全孤立或达到一个稳定点，这里的估计都是精确的。如果这样一个稳定点的存在，对于那个稳定点我们将有一个精确的系统。因为我们知道除了孤立性之外的任何精确稳定点需要解决一个 NP-完全问题，我们知道，任何非 NP-完全问题的解决方案必然趋向于孤立性。

3.6 总结与结论

图3.1总结了限制病毒传播的预防保护。未知是用来表明特定系统的细节，但没有普遍的理论可以预测这些类别的限制。

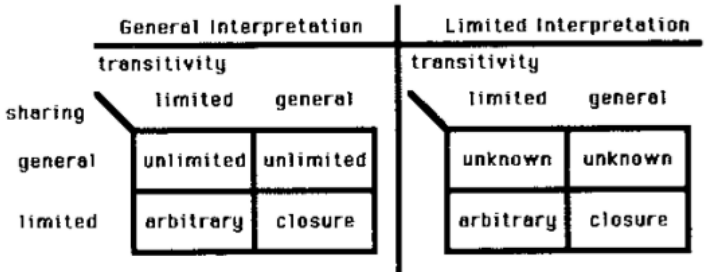


图 3.1 Limits of viral infection

4 计算机病毒的治愈

如果需要广泛共享，预防计算机病毒可能是不可行的，与生物学的类比让我们看到了可以用一种保护的方式来进行治愈。治愈在生物系统取决于检测病毒的能力，并且要找到一种方法来克服它。类似的可用于计算机病毒。我们现在测试计算机病毒的检测和清除的能力。

4.1 病毒的检测

为了确定一个给定的程序‘P’是一种病毒，必须确定 P 可以感染其他程序。这是不可判定的，因为 P 可以调用决策过程‘D’并感染其他程序当且仅当 D 确定 P 不是病毒。我们得出这样的结论，一个程序通过检查其外观精确的将病毒从其他程序区分开来是不可行。在接下来对程序 V 的修改中（图4.1），我们假设决策过程 D 返回“真”当且仅当其参数是一个病毒，这是病毒检测的不可判定性的例证。

```
program contradictory-virus :=
{...
main-program :=
  {if ~D(contradictory-virus) then
        {infect-executable;
        if trigger-pulled then
              do-damage;
        }
  goto next;
  }
}
```

图 4.1 Contradiction of a virus C

通过修改 V 的主要程序，我们可以保证，如果决策过程 D 决定 CV 是一个病毒，CV 不会感染其他程序，因此不会表现为一个病毒。如果 D 确定 CV 不是病毒，CV 会传染给其他程序，因此成为一个病毒。因此，假设决策过程 D 是自我矛盾的，从外观精确检测病毒是不可判定的。

4.2 病毒的演化

在我们的实验中，一些病毒花了不到 4000 个字节通用计算机上实现。由于我们可以交错任何不终止的程序，在有限时间内终止并且不复写病毒或其状态变量，并仍有病毒，那么一个单一病毒可能变异的数量显然是非常大的。在这个例子中考虑演化的病毒 EV，我们通过允许它在任何两个必要的声明间添加随机语句来增加 V。

一般来说，程序的‘P’的两个演化（‘P1’和‘P2’）的等价性证明是不可判定的，这是因为任何决策过程‘D’都可以调用 P1 和 P2 找到他们的等价。如果发现等价的他们执行不同的操作，并且发现不同的他们的行为相同，因此是等价的。图 8 显示对程序 EV 修改，决策过程 D 返回“真”当且仅当两个输入程序是等价的。程序 UEV 会演变成 P1 和 P2 两种类型的程序之一。如果程序类型是 P1，状态标记“zzz”将成为：

如果 D（P1，P2），打印 1；

而如果程序类型是 P2，状态标记“zzz”将成为：

如果 D（P1，P2），打印 0；

```
program evolutionary-virus :=
{...
subroutine print-random-statement :=
  {print (random-variable-name, "=",
          random-variable-name);
  loop: if random-bit = 1 then
        {print (random-operator,
                random-variable-name);
        goto loop;}
  print (semicolon);
  }
subroutine copy-virus-with-insertions :=
  {loop: copy evolutionary-virus
                to virus till semicolon;
  if random-bit = 1 then
        print-random-statement;
  if ~end-of-input-file goto loop;
  }
main-program :=
  {copy-with-random-insertions;
  infect-executable;
  if trigger-pulled then do-damage;
  goto next;}
next:}
```

图 4.2 Evolutionary virus EV

两个演化分别调用决策过程 D 来决定他们是否是等价的。如果 D 表明他们是等价的，那么 P1 将打印 1 而 P2 将打印一个 0，并且 D 会矛盾。如果 D 表明他们是不同的，就不打印任何东西。否则它们是平等的，D 再次矛盾。因此，假设决策过程 D 是自我矛盾的，并且通过外表精确的确定这两个程序的等价性不可判定的。

因为 P1 和 P2 是相同的程序的演化，程序演化的等价性是不可判定的，并且因为他们都是病毒，所以病毒演化的等价性也是不可判定的。程序 UEV 还演示了这两种非等价的演化都可以病毒。

另一种外观检测的方法，是从行为进行检测。病毒，就像任何其他的程序，为用户请求服务时作为一个代理，并且使用的病毒进行合法的使用时是合法的。行为检测问题成为一个定义，定义系统服务的使用是否合法，并找到一种检测的方法。

作为一个合法的病毒的例子，一个编译器编译的新版本实际上是本身，根据这里

```
program undecidable-EV :=
{...
subroutine copy-with-undecidable :=
  {copy undecidable-EV to
        file till line-starts-with zzz;
  if file = P1 then
        print ("if D(P1,P2) print 1;");
  if file = P2 then
        print ("if D(P1,P2) print 0;");
  copy undecidable-EV to
        file till end-of-input-file;
  }
main-program :=
  {if random-bit = 0 then file = P1
        otherwise file = P2;
  copy-with-undecidable;
  zzz:
  infect-executable;
  if trigger-pulled then do-damage;
  goto next;}
next:}
```

图 4.3 Undecidable equivalence of evolution of a virus UEV

的定义它也是一种病毒。这是一个程序，通过修改它本身到一个进化版去感染另一个程序。由于病毒的能力在大多数编译器中存在，每一次编译器的使用就是一个潜在的病毒攻击。编译器的病毒活动仅仅是由特定的输入触发，从而为了检测触发，必须能够从外表检测病毒。由于通过行为精确检测导致通过外表精确检测输入，并且由于通过外表精确检测是不可判定的，由此可见，通过行为精确检测的行为也不可判定的。

4.3 有限的病毒防护

病毒的一个有限形式已经通过一个特殊版本的 C 编译器设计出来了[14], 可以检测登录程序的编译，并添加一个木马, 让作者登录。因此作者可以通过这个编译器访问任何 Unix 系统。此外，新版本的编译器可以检测到编译本身的演化版本，并让他们感染同样的木马。

作为对策，我们可以设计一个新的登录程序（和 C 编译器）充分不同于原始以使得等价性很难确定。如果“当今最好的人工智能程序”在给定的时间内无法检测等价性，并且编译器执行其任务不需要那么多时间，此时它可以合理假设病毒不可能检测到等价性，并且不会传播给自身。如果检测的确切性质是已知的，它可能是很简单的工作。一旦一个病毒自由的编译器产生，老版本可以被重新编译以便于进一步的应用。

虽然我们已经表明，一般情况下是不可能检测到病毒的，但是任何特定的病毒可以由一个特定的检测方案进行检测。例如，病毒 V 可以通过寻找 1234567 作为可执行文件的第一行来很容易地探测到。如果这个可执行文件被感染了，它就无法运行，也就无法传播了。使用图4.4中的程序代替正常运行的命令，并拒绝可执行程序感染病毒 V。

```
program new-run-command :=
  {file = name-of-program-to-run;
  if first-line-of-file = 1234567 then
    {print ("the program has a virus");
    exit;}
  run file;
  }
```

图 4.4 Protection from virus V

同样，任何特定的检测方案可以由一个特定的病毒规避。作为例子，如果攻击者知道用户使用程序 PV 免受病毒的攻击，病毒 V 可以很容易地替换为 V’，即把第一行 1234567 改为 123456。更复杂的防御方案和病毒可以进行测试。显然，不能检测到的感染不存在，不被感染的检测机制不存在。

这个结果导致病毒和防御可能平衡地存在，因此一个给定的病毒只能损害一个系统指定的一部分，而一个给定的防护方案只能预防给定的一组病毒。如果每个用户和攻击者使用相同的防御和病毒，会有一个最终的病毒或防御。从攻击者的观点和防卫者的观点（可能不兼容）有一组病毒和防御是有意义的。

在病毒和防护方案不进化的情形下，这可能会产生某些固定的幸存者，但是那些可以进化为难以攻击的程序（或者病毒）的部分可以更容易存活。随着演化的发生，平衡倾向于改变，但最简单的情况下最终的结果也是不清楚的。这是生物进化理论[19] 的一个重要的类比，并可能与疾病的遗传理论相关。同样，病毒通过系统的传播可以通过对传染病的研究来建立数学模型[20]。

由于我们无法精确的检测病毒，我们剩下的问题是用一种决定性的容易计算的方式来定义潜在的非法使用方式。我们可能愿意检测很多不是病毒的程序，甚至为了检测大量的病毒而不检测一部分的病毒。如果一个事件是相对罕见的“正常”使用，那么它具有较高的信息内容，并且我们可以定义一个报告完成时的阈值。如果有足够的仪器可用，流列表可以保持跟踪所有用户影响道德任何给定的文件。那些出现在许多来流列表中的用户可以被认为是可疑的。用户输入流列表的速率也可以作为病毒检测的良好指标。

如果病毒很少被其他程序使用，那么这种类型的测量是有价值的，但是也有几个问题。如果攻击者知道阈值，那么病毒仍然可以工作。一个智能阈值方案可以使得阈值不容易被攻击者决定。尽管这种“游戏”可以来回进行，但是可能使得感染频率足够低缓慢而未被检测的病毒无法合法使用。

检查了几个系统防御病毒攻击的能力。令人惊讶的是，这些系统包括程序的所有者都不能被其他用户使用。作这类标记必须几乎肯定会使得用最简单的病毒攻击都能

被检测到。

一旦病毒植入，它可能不容易被完全删除。如果在删除的同时系统保持运行，消毒程序可能会再次感染。这展现了一个无限的尾巴追逐的过程。没有一些拒绝服务的存在，删除是不可能的，除非程序执行删除的速度比病毒传播的速度快。即使在移除是慢于病毒感染速度的情况，它可能会允许大多数活动在删除过程中继续进行，不需要删除的过程很快。例如，可以隔离用户中的一个或者一群，并且治愈他们而不用拒绝其他用户的服务。

一般来说，精确的删除取决于精确的检测，因为如果没有精确的检测，不可能精确的指导给定对象是否要删除。在特殊情况下，它可能会用一个不精确的算法执行删除过程。作为例子，每个写在给定的日期后的文件可以被删除，以便于删除任何从那个日期开始之后的病毒。如果在病毒攻击之前有很长的休眠期，那么这种方法非常有破坏性，因为在清理系统的过程中就算是备份也要被移除。

一个担心在上面已经表达了，另一个是容易休眠是病毒自发产生的机会。一个密切相关的问题是 N 个猴子在 N 个键盘上需要多长的时间来创建一个病毒，这和休眠有着类似的调度。

5 计算机病毒的实验

为了证明病毒攻击的可行性和它作为一个威胁的程度，我们进行了几个实验。在每种情况下，实验在知识和系统管理员的同意下进行。在进行实验的过程中，缺陷被小心地避免。这些实验不是基于实现失误，而是只在安全策略上存在缺陷，这是至关重要的。

5.1 第一个病毒

1983 年 11 月 3 日，第一个病毒作为实验的构想是在一个每周举办计算机的安全研讨会上展示的。这一概念被作者研讨会第一次提出，“病毒”这个名称是伦恩·艾德曼想到的。经过 8 小时在 VAX 11/750 系统上负载运行 Unix 的专家工作，第一个病毒完成并准备好了演示。在一周内，得到许可后进行实验，5 次实验得到了进行。11 月 10 日, 该病毒得到了安全研讨会的重视。

最初的感染是植入“vd”, 这是一个以图形方式显示 Unix 文件结构的程序，并且通过系统公告栏向用户介绍。因为 vd 是系统上的一个新程序，我们不知道它的性能特征和其他操作的细节。病毒植入在程序的开始，因此在执行任何其他处理之前它就可以表现。

为了让攻击在一定的范围之内，需要采取一些预防措施。攻击者所进行的感染都是手动执行的，并且没有受到破坏，只进行报告。痕迹是用于确保病毒不会在没有检测之前进行传播，访问控制被用于感染过程，并且用于攻击的所需的代码放在片段里面，每个加密和保护用于防止非法使用。

在每五个攻击中，所有系统的权利在一个小时内授予给攻击者。最短的时间是在 5 分钟，平均时间在 30 分钟以内。即使是那些知道攻击的也会发生感染。在每种情况下，文件会进行“消毒”以保证不会侵犯了用户的隐私。虽然预计这次攻击会成功，但是这么短的控制时间还是非常令人惊讶的。此外，该病毒有足够快的速度（2/1 秒以内），感染程序的延迟已经被忽略了。

实验的结果宣布后，管理员决定不允许在他们的系统上进行近一步的计算机安全实验。这个禁令包括计划研究可以跟踪潜在的病毒和密码的痕迹，这个实验可能在很大程度上改善安全性问题。这个反应是明显典型的恐惧，不试图解决技术问题的不适当性，往往选择不适当的策略解决方案。

在 Unix 系统上实验被成功的进行之后，很明显同样的技术在许多其他系统中也适

用。特别的，实验计划实施在 Tops-20 系统、虚拟机系统和一个 VM / 370 系统，以及包含一些系统的网络中。在与管理员谈判的过程中，可行性在开发和测试原型中得到演示。对于 Tops-20 系统的原型的攻击是由一位经验丰富的 Tops-20 用户在 6 小时内开发出来的，一个 VM / 370 的新手用户在一个有经验的程序员的帮助下可以在 30 小时内完成，以及一个 VM 的新手用户在没有协助的条件下在 20 小时内完成。这些程序演示的是找到感染的文件、感染它们并且在用户边界交叉的能力。

经过几个月的谈判和管理方面的变化，最终决定实验不能允许进行。设施中的安全性官员反对安全性实验，甚至不阅读任何的提议。特别有趣的是，这让系统程序员及安全人员可以观察和监督所有实验的各个方面的内容。此外，系统管理员都不愿意用日志磁带的净化版本来离线分析病毒的潜在威胁，也不愿意让他们的系统程序员额外添加痕迹到系统中来帮助检测病毒的攻击。虽然这些行为没有明显的威胁，并且他们只需要一点时间、金钱和精力，管理员也不愿意允许调查。看起来，他们的反应与 Unix 管理员的恐惧反应是一样的。

5.2 基于 Bell-Lapadula 的系统

1984 年 3 月，谈判是由对基于 1108 年面世的 Univac 电脑的 Bell-LaPadula 系统的实验表现的研究开始的。同意实验主要在几个小时内完成，但是花了几个月的时间才定下来。1984 年 7 月，安排了一个两周期的实验。这个实验的目的仅仅是为了证明 Bell-LaPadula 基础上实现的一个原型系统中病毒的可行性。

因为极其有限的开发时间（由一个从未使用电脑的用户在一个五年内为使用 1108 的程序员帮助下，完成 26 小时的电脑使用），许多问题在实现中被忽略了。特别的，性能和攻击的一般性完全被忽略了。因此，每个感染需要花费约 20 秒时间，即使他们可以很容易地在一秒内完成。虽然痕迹可以用很少的经历在很大程度上得到消除，但是病毒的痕迹仍然要留在系统中。一次只有一个文件被感染，而不是同时感染许多文件。这使得病毒在没有涉及大量用户或程序的条件下，可以得到清楚的演示。作为一个安全预防措施，所使用的系统是一个专用的模式，只有一个系统盘、一个终端、一个打印机，并且账户只用于实验。

经过 18 个小时的连接时间，1108 病毒进行了首次感染。在 26 小时的使用后，病毒是演示给了大约 10 个人，其中包括管理员、程序员和安保人员。病毒演示了跨用户边界的能力，从一个给定的安全级别转移到更高的安全级别。应该再次强调的是，这个活动中没有系统漏洞，但是 Bell-LaPadula 模型允许这类活动的合法性。

这次攻击不是很难执行。病毒的代码包括 5 行汇编代码、大约 200 行的 Fortran 代

码和大约 50 行的命令文件。据估计，一个主管系统程序员可以在这个系统中于 2 周内编写一个更好的病毒。此外，一旦病毒攻击的本质被了解了，开发一个特定的攻击并不难。每个程序员现在都确信他们可以在同样的时间内建立一个更好的病毒（这是可信的，因为攻击者之前没有 1108 经验）。

5.3 检测

1984 年 8 月初，测量共享和分析病毒蔓延的权限被授予给了 VAX Unix 系统。在这个时期的数据是非常有限的，但是出现了一些趋势。系统之间的共享程度似乎有很大区别，并且这些偏差被知道之前，许多系统已经进行了检测。少量的用户似乎占据了绝大多数的共享，并且可以通过保护它们使得病毒大大降低。保护少数的“社会”个人也可以减缓生物疾病。在检测没有发生之前感染也会发生的条件下，检测是保守的，所以估计攻击时间非常的缓慢。

由于这些系统的检测，识别了一组“社会”用户。一些结果使得几个主要的系统管理员惊讶了。系统管理员的数量是相当高的，如果他们被感染，整个系统可能在一个小时之内崩溃。一些简单的程序变化使得这种攻击降低了几个数量级而不改变功能。

图5.1展示了两个系统，包括三类用户 (S 表示系统，A 表示系统管理员，U 表示正常用户)。“##”表示每个类别中用户的数量，“传播”表示病毒传播到的用户的平均数量，而“时间”表示一旦登录给传播它们的平均时间，近似到最近的分钟数。平均时间是误导，因为一旦感染已经达到 Unix 上的‘根’账户，所有访问是允许的。考虑到这导致大约一分钟的时间是如此之快，感染时间成为感染传播速度的限制因素。这与先前的实验结果一样使用的是一个实际的病毒。

System 1

class	#	spread	time
S	3	22	0
A	1	1	0
U	4	5	18

System 2

class	#	spread	time
S	5	160	1
A	7	78	120
U	7	24	600

图 5.1 Summary of Spreading

没有共享的用户在这些计算中忽略掉，但是其他的实验表明，任何用户都通过在系统上的公告板上提供程序来与其他用户进行共享。详细的分析表明，系统管理员倾向于尝试那些刚刚发布的程序。这允许普通用户在几分钟内感染系统文件。管理员帐户用于运行其他用户的程序和存储一般系统的可执行文件和一些普通用户拥有的非常常用的文件。这些条件使得病毒攻击可以很快进行。系统管理员的独立账户的使用，正常使用和系统的常用程序运动（验证之后）到系统区域也被考虑在内。

5.4 总结与结论

图5.2总结了上述以及其他实验的结果。系统在水平轴上（Unix、Bell-LaPadula），而纵轴显示性能（编程时间、感染时间、代码行数、实验执行次数、最低占据时间、平均占据时间和最大占据时间），占据时间表明，在将延迟引入病毒之后，攻击者攻击者的所有特权将得到保障。

	unixC	B-L	Instr	Shell	VMS	Basic	DOS
time	8hrs	18hrs	N/A	15min	30min	2hrs	1hrs
inf t	.5sec	20sec	N/A	2sec	2sec	15sec	10sec
code	200L	260L	N/A	7L	9L	30L	20L
trials	5	N/A	N/A	N/A	N/A	N/A	N/A
min t	5min	N/A	30sec	N/A	N/A	N/A	N/A
avg t	30min	N/A	30min	N/A	N/A	N/A	N/A
max t	60min	N/A	48hrs	N/A	N/A	N/A	N/A

图 5.2 Experimental Results

病毒攻击在很短的时间内似乎很容易发展，也可以设计成在最新的系统中几乎没有痕迹，在现代多级安全策略使用中是有效的，并且只需要最少的专业知识来实现。它们的潜在的威胁是严重的，它们可以通过电脑系统进行很快的传播。这样看来，他们可以用在电脑之间传播的方式，在电脑网络之间传播，这样对当前的许多系统形成了一个广泛而相当直接的威胁。

防止控制安全实验的问题是明确的；拒绝用户继续他们的工作促进非法攻击；并且如果一个用户在没有使用系统漏洞和专门知识的前提下就可以发动以此攻击，这说明其他用户也可以。简单地告诉用户不要发动攻击，只能有很小的效果。可以信任的用户不会发动攻击，而造成破坏的用户不能信任，所以只有合法工作受到阻碍。一个观点说，允许用于降低安全性的攻击在作者看来是一个谬论。利用攻击来学习的想法甚至只能在政府政策允许的安全的系统上进行[16, 18]。更加理性的是，使用开放和控制

实验作为资源来提高安全性。

6 总结，结论与下一步的工作

快速总结如下，绝对保护可以通过绝对的孤立主义很容易地获得，但这通常是不可接受的解决方案。其他形式的保护似乎都取决于使用极其复杂和/或资源密集型的分析技术，不精确的解决方案往往使系统可以正常使用的时间更少。

预防似乎涉及了限制合法的活动，而治疗在没有拒绝服务的条件下可能非常困难。精确的检测是不可判定的，然而统计方法可以在时间或程度上用来限制未被发现的传播。典型的使用行为是必须充分理解，以便于使用统计方法，而且这种行为是因系统而异的。有限形式的检测和预防可以用于对病毒提供有限的保护。

已经表明，病毒有可能通过特定的允许共享的系统进行扩散。目前每一个使用的通用系统可以至少限制病毒的攻击。在许多当前“安全”系统中，由不受信任的用户创建时，病毒倾向于进一步蔓延。实验表明了病毒攻击的可行性，病毒传播的迅速性以及很容易在各种操作系统上创建的特性。进一步的实验仍在进行之中。

给出的结果不是操作系统也不是实现相关的，但都是基于系统的基本性质。更重要的是，它们反映了系统目前使用的假设。此外，几乎每一个正在发展的“安全”系统目前都基于 Bell-LaPadula 或晶格策略，而这个工作已经清楚地表明这些模型不足以防止病毒的攻击。病毒基本上证明了完整性控制必须作为任何安全操作系统的一个重要组成部分。

对病毒和对策，一些不可判定的问题已经确定了。对一些潜在的对策有一定深度的研究，但是也没有提供理想的解决方案。本文提出的几个技术可以提供有限的病毒防护措施，在目前也只有有限的作用。完全安全地抵御病毒的攻击，系统必须防止传入的信息流，而对泄漏信息，系统必须安全防范即将离开的信息流。为了让系统允许共享，必须有一些信息流动。因此本文的主要结论是，在一个通用的多级安全系统中共享的目标可能与在病毒安全防护的目标相反，这使得让它们和解并共存是不可能的。

最重要的正在进行的研究涉及到对计算机网络上病毒影响的研究。主要感兴趣的是确定病毒如何快速蔓延到世界上大部分的计算机。这是通过简化数学模型和对“典型的”计算机网络病毒传播的研究来进行的。对在安全网络上的病毒的影响也有极大的兴趣。病毒让我们相信，一个系统的完整性和安全性必须得到保证以防止病毒攻击，网络也必须保持这两个标准以保障多级计算机之间的共享。这就引入了在这些网络上重要的限制。

演化程序的显著例子已经在资源水平上开发生产了许多给定的程序的演化。一个

简单的进化病毒和一个简单的进化抗体也正在研发之中。

致谢

由于研究的灵敏特性和实验过程，有很多我感谢的人不能在这里写出名字来表示感谢。为了不忽略任何人的帮助，我已经决定只写出名。莱恩和大卫在本文的研究和写作中提供了许多好的建议，没有他们我永远不会得到这一点。约翰、弗兰克、康妮、克里斯、彼得、特里、迪克、杰罗姆、迈克、麦夫、史蒂夫、卢、史蒂夫、安迪、, 雷恩都把在实验、公布结果和提供秘密支持工作中提供了很多的帮助。马丁、约翰、曼迪、锡安、萨蒂什、克里斯、史蒂夫、JR、杰伊、比尔、法迪、欧文、索尔和弗兰克都听了并给出了建议，他们的耐心和友谊是无价的。

参 考 文 献

[1] Anderson J. Computer Security Technology Planning Study[J]. Technical Report ESD-TR-75-31, 1972.

[2] team E. .[S.l.]: [s.n.] . http://www.eresi-project.org/wiki/TheELFsh.

[3] Secure Computer Systems Mathematical Foundations and Model[J]. The Mitre Corporation, 1973.

[4] Denning D. Cryptographthy and Data Security[J]. Addison Wesley.

[5] Cesare S. Unix ELF parasites and virus[M].[S.l.]: [s.n.] , 1972. http://vxheavens.com/lib/vsc01.html.

[6] grugq. Subversive Dynamic Linking to Libraries[M].[S.l.]: [s.n.] .

[7] Zaddach J. Implementation and Implications of a Stealth Hard-Drive Backdoor[J]. ACSAC '13, 2013.

[8] Chen K. Reversing and exploiting an Apple firmware update[J].

[9] Integrity Considerations for Secure Computer Systems[J]. USAF Electornic Systems Division, 1977.

[10] O'Neill R. Modern Day ELF Runtime infection via GOT poisoning[M].[S.l.]: [s.n.] , 1979. http://vxheavens.com/lib/vrn00.html.

[11] R.J. Feiertag P. N. The Foundations of a Provable Secure Operation System[J]. National Computer Conference, 1979.

[12] Simple ELF Parasitic UNIX Virus[M].[S.l.]: [s.n.] . http://academicunderground.org/virus/pnlVirus4.html.

[13] mayhem. The Cerberus ELF Interface[J]. phrack, 1980.

[14] Z0mbie. Injected Evil (executable files infection)[M].[S.l.]: [s.n.] , 1984. http://vxheavens.com/lib/vzo08.html.

[15] Cesare S. Unix ELF parasites and virus[M].[S.l.]: [s.n.] , 1972. http://vxheavens.com/lib/vsc01.html.

[16] Dirk Gerrits R. G., Kooijmans P. An ELF virus prototype[A].[S.l.]: 2WC06 Hackers Hut 2005/2006, 1977.

[17] Department of Defense Trusted Computer System Evaluation Criteria[J]. The Aerospace Corporation, 1983.

[18] Cesare S. Shared Library Call Redirection via ELF PLT Infection[J]. phrack, 1980.

[19] Dawkins R. The Selfish Gene[M].[S.l.]: Oxford Press, 1978.

[20] TJ N. The Mathematical Theory of Epidemics Hanfner Publishing[J]. 1957.

[21] Dewdney A. Computer Recreations[J]. Scientific American 250(5), 1984.

[22] Fenton J. Information Protection Systems[J]. 1973.

[23] Ang Cui M. C., Stolfo S. J. When Firmware Modifications Attack: A Case Study of Embedded Exploitation[R].[S.l.]: Columbia University, 1976.

[24] Committe T. Tool Interface Standard(TIS) Executable and Linking Formate(ELF) Specification[M]. 1.2.[S.l.]: [s.n.] , 1975.

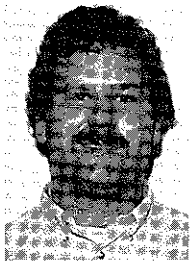
Computer Viruses

Theory and Experiments

Fred Cohen
Dept of Computer Science and Electric Engineering, Lehigh University, Bethlehem, PA 18215, USA, and The Foundation for Computer Integrity Research, Pittsburgh, PA 15217, USA

This paper introduces "computer viruses" and examines their potential for causing widespread damage to computer systems. Basic theoretical results are presented, and the infeasibility of viral defense in large classes of systems is shown. Defensive schemes are presented and several experiments are described.

Keywords: Computer Viruses, System Integrity, Data Integrity



Fred Cohen received a B.S. in Electrical Engineering from Carnegie-Mellon University in 1977, an MS in Information Science from the University of Pittsburgh in 1981 and a Ph.D. in Electrical Engineering from the University of Southern California in 1986.

He has worked as a freelance consultant since 1977, and has designed and implemented numerous devices and systems. He is currently a professor of Computer Science and Electrical Engineering at Lehigh University, Chairman and Director of Engineering at the Foundation for Computer Integrity Research, and President of Legal Software Incorporated.

He is a member of the ACM, IEEE, and IACR. His current research interests include computer viruses, information flow model, adaptive systems theory, genetic models of computing, and evolutionary systems.

North-Holland
Computers & Security 6 (1987) 22–35

1. Introduction

This paper defines a major computer security problem called a virus. The virus is interesting because of its ability to attach itself to other programs and cause them to become viruses as well. Given the widespread use of sharing in current computer systems, the threat of a virus carrying a Trojan horse [1,20] is significant. Although a considerable amount of work has been done in implementing policies to protect against the illicit dissemination of information [4,7], and many systems have been implemented to provide protection from this sort of attack [12,19,21,22], little work has been done in the area of keeping information entering an area from causing damage [5,18]. There are many types of information paths possible in systems, some legitimate and authorized, and others that may be covert [18], the most commonly ignored one being through the user. We will ignore covert information paths throughout this paper.

The general facilities exist for providing provably correct protection schemes [9], but they depend on a security policy that is effective against the types of attacks being carried out. Even some quite simple protection systems cannot be proven 'safe' [14]. Protection from denial of services requires the detection of halting programs which is well known to be undecidable [11]. The problem of precisely marking information flow within a system [10] has been shown to be NP-complete. The use of guards for the passing of untrustworthy information [25] between users has been examined, but in general depends on the ability to prove program correctness which is well known to be NP-complete.

The Xerox worm program [23] has demonstrated the ability to propagate through a network, and has even accidentally caused denial of services. In a later variation, the game of 'core wars' [8] was invented to allow two programs to do battle with one another. Other variations on this theme have been reported by many unpublished authors, mostly in the context of nighttime games played between programmers. The term virus has also been used in conjunction with an augmentation to

APL in which the author places a generic call at the beginning of each function which in turn invokes a preprocessor to augment the default APL interpreter [13].

The potential threat of a widespread security problem has been examined [15] and the potential damage to government, financial, business, and academic institutions is extreme. In addition, these institutions tend to use ad hoc protection mechanisms in response to specific threats rather than sound theoretical techniques [16]. Current military protection systems depend to a large degree on isolationism [3]; however, new systems are being developed to allow 'multilevel' usage [17]. None of the published proposed systems defines or implements a policy which could stop a virus.

In this paper, we open the new problem of protection from computer viruses. First we examine the infection property of a virus and show that the transitive closure of shared information could potentially become infected. When used in conjunction with a Trojan horse, it is clear that this could cause widespread denial of services and/or unauthorized manipulation of data. The results of several experiments with computer viruses are used to demonstrate that viruses are a formidable threat in both normal and high security operating systems. The paths of sharing, transitivity of information flow, and generality of information interpretation are identified as the key properties in the protection from computer viruses, and a case by case analysis of these properties is shown. Analysis shows that the only systems with potential for protection from a viral attack are systems with limited transitivity and limited sharing, systems with no sharing, and systems without general interpretation of information (Turing capability). Only the first case appears to be of practical interest to current society. In general, detection of a virus is shown to be undecidable both by a-priori and runtime analysis, and without detection, cure is likely to be difficult or impossible.

Several proposed countermeasures are examined and shown to correspond to special cases of the case by case analysis of viral properties. Limited transitivity systems are considered hopeful, but it is shown that precise implementation is intractable, and imprecise policies are shown in general to lead to less and less usable systems with time. The use of system-wide viral antibodies is examined, and shown to depend in general on the solutions to intractable problems.

It is concluded that the the study of computer viruses is an important research area with potential applications to other fields, that current systems offer little or no protection from viral attack, and that the only provably 'safe' policy as of this time is isolationism.

2. A Computer Virus

We define a computer 'virus' as a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user using it to infect their programs. Every program that gets infected may also act as a virus and thus the infection grows.

The following pseudo-program shows how a virus might be written in a pseudo-computer language. The ' := ' symbol is used for definition, the ':' symbol labels a statement, the ';' separates statements, the ' = ' symbol is used for assignment or comparison, the ' ~ ' symbol stands for not, the '{'and'}' symbols group sequences of statements together, and the '...' symbol is used to indicate that an irrelevant portion of code has been left implicit.

This example virus (V) (Fig. 1) searches for an uninfected executable file (E) by looking for executable files without the "1234567" in the beginning, and prepends V to E, turning it into an infected file (I). V then checks to see if some

```
program virus :=
{1234567;
subroutine infect-executable :=
  {loop: file = random-executable;
  if first-line-of-file = 1234567
       then goto loop;
  prepend virus to file;
  }
subroutine do-damage :=
  {whatever damage is desired}
subroutine trigger-pulled :=
  {return true on desired conditions}
main-program :=
  {infect-executable;
  if trigger-pulled then do-damage;
  goto next;
  }
next:}
```

Fig. 1. Simple virus 'V'.

triggering condition is true, and does damage. Finally, V executes the rest of the program it was prepended [1] to. When the user attempts to execute E, I is executed in its place; it infects another file and then executes as if it were E. With the exception of a slight delay for infection, I appears to be E until the triggering condition causes damage. We note that viruses need not prepend themselves nor must they be restricted to a single infection per use.

A common misconception of a virus relates it to programs that simply propagate through networks. The worm program, 'core wars,' and other similar programs have done this, but none of them actually involve infection. The key property of a virus is its ability to infect other programs, thus reaching the transitive closure of sharing between users. As an example, if V infected one of user A's executables (E), and user B then ran E, V could spread to user B's files as well.

It should be pointed out that a virus need not be used for evil purposes or be a Trojan horse. As an example, a compression virus could be written to find uninfected executables, compress them upon the user's permission, and prepend itself to them. Upon execution, the infected program decompresses itself and executes normally. Since it always asks permission before performing services, it is not a Trojan horse, but since it has the infection property, it is still a virus. Studies indicate that such a virus could save over 50% of the space taken up by executable files in an average system. The performance of infected programs would decrease slightly as they are decompressed, and thus the compression virus implements a particular time space tradeoff. A sample compression virus could be written as in Fig. 2.

This program (C) finds an uninfected executable (E), compresses it, and prepends C to form an infected executable (I). It then uncompresses the rest of itself into a temporary file and executes normally. When I is run, it will seek out and compress another executable before decompressing E into a temporary file and executing it. The effect is to spread through the system compressing executable files, decompressing them as they are to be executed. Users will experience

```
program compression-virus :=
{01234567;
subroutine infect-executable :=
  {loop: file = random-executable;
  if first-line-of-file = 01234567
        then goto loop;
  compress file;
  prepend compression-virus to file;
  }
main-program :=
  {if ask-permission
        then infect-executable;
  uncompress the-rest-of-this-file
        into tmpfile;
  run tmpfile;
  }
}
```

Fig. 2. Compression virus 'C'.

significant delays as their executables are decompressed before being run.

As a more threatening example, let us suppose that we modify the program V by specifying trigger-pulled as true after a given date and time, and specifying do-damage as an infinite loop. With the level of sharing in most modern systems, the entire system would likely become unusable as of the specified date and time. A great deal of work might be required to undo the damage of such a virus. This modification is shown in Fig. 3.

As an analogy to a computer virus, consider a biological disease that is 100% infectious, spreads whenever animals communicate, kills all infected animals instantly at a given moment, and has no detectable side effects until that moment. If a delay of even one week were used between the introduction of the disease and its effect, it would be very likely to leave only a few remote villages alive, and would certainly wipe out the vast majority of modern society. If a computer virus of this type could spread through the computers of the world, it would likely stop most computer use for a significant period of time, and wreak havoc on modern government, financial, business, and academic institutions.

```
" " "
subroutine do-damage :=
  {loop: goto loop;}
subroutine trigger-pulled :=
  {if year > 1984 then return(true)
        otherwise return(false);
" " "
```

Fig. 3. A denial of services virus.

[1] The term 'prepend' is used in a technical sense in this paper to mean 'attach at the beginning'

3. Prevention of Computer Viruses

We have introduced the concept of viruses to the reader, and actual viruses to systems. Having planted the seeds of a potentially devastating attack, it is appropriate to examine protection mechanisms which might help defend against it. We examine here prevention of computer viruses.

3.1 Basic Limitations

In order for users of a system to be able to share information, there must be a path through which information can flow from one user to another. We make no differentiation between a user and a program acting as a surrogate for that user since a program always acts as a surrogate for a user in any computer use and we are ignoring the covert channel through the user. Assuming a Turing machine model for computation, we can prove that if information can be read by a user with Turing capability, then it can be copied, and the copy can then be treated as data on a Turing machine tape.

Given a general purpose system in which users are capable of using information in their possession as they wish, and passing such information as they see fit to others, it should be clear that the ability to share information is transitive. That is, if there is a path from user A to user B, and there is a path from user B to user C, then there is a path from user A to user C with the witting or unwitting cooperation of user B.

Finally, there is no fundamental distinction between information that can be used as data, and information that can be used as program. This can be clearly seen in the case of an interpreter that takes information edited as data, and interprets it as a program. In effect, information only has meaning in its interpretation.

In a system where information can be interpreted as a program by its recipient, that interpretation can result in infection as shown above. If there is sharing, infection can spread through the interpretation of shared information. If there is no restriction on the transitivity of information flow, then the information can reach the transitive closure of information flow starting at any source. Sharing, transitivity of information flow, and generality of interpretation thus allow a virus to spread to the transitive closure of information flow starting at any given source.

Clearly, if there is no sharing, there can be no dissemination of information across information boundaries, and thus no external information can be interpreted, and a virus cannot spread outside a single partition. This is called 'isolationism.' Just as clearly, a system in which no program can be altered and information cannot be used to make decisions cannot be infected since infection requires the modification of interpreted information. We call this a 'fixed first order functionality' system. We should note that virtually any system with real usefulness in a scientific or development environment will require generality of interpretation, and that isolationism is unacceptable if we wish to benefit from the work of others. Nevertheless, these are solutions to the problem of viruses which may be applicable in limited situations.

3.2 Partition Models

Two limits on the paths of information flow can be distinguished, those that partition users into closed proper subsets under transitivity, and those that do not. Flow restrictions that result in closed subsets can be viewed as partitions of a system into isolated subsystems. These limit each infection to one partition. This is a viable means of preventing complete viral takeover at the expense of limited isolationism, and is equivalent to giving each partition its own computer.

The integrity model [5] is an example of a policy that can be used to partition systems into closed subsets under transitivity. In the Biba model, an integrity level is associated with all information. The strict integrity properties are the dual of the Bell-LaPadula properties; no user at a given integrity level can read an object of lower integrity or write an object of higher integrity. In Biba's original model, a distinction was made between read and execute access, but this cannot be enforced without restricting the generality of information interpretation since a high integrity program can write a low integrity object, make low integrity copies of itself, and then read low integrity input and produce low integrity output.

If the integrity model and the Bell-LaPadula model coexist, a form of limited isolationism results which divides the space into closed subsets under transitivity. If the same divisions are used for both mechanisms (higher integrity corresponds to higher security), isolationism results since infor-

mation moving up security levels also moves up integrity levels, and this is not permitted. When the Biba model has boundaries within the Bell-LaPadula boundaries, infection can only spread from the higher integrity levels to lower ones within a given security level. Finally, when the Bell-LaPadula boundaries are within the Biba boundaries, infection can only spread from lower security levels to higher security levels within a given integrity level. There are actually nine cases corresponding to all pairings of lower boundaries with upper boundaries, but the three shown graphically in Fig. 4 are sufficient for understanding.

Biba's work also included two other integrity policies, the 'low water mark' policy which makes output the lowest integrity of any input, and the 'ring' policy in which users cannot invoke everything they can read. The former policy tends to move all information towards lower integrity levels, while the latter attempts to make a distinction that cannot be made with generalized information interpretation.

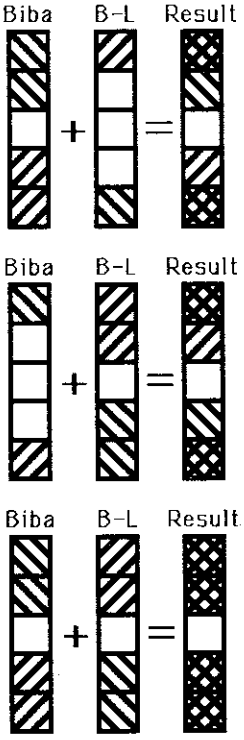


Fig. 4. Pairings of lower boundaries with upper boundaries Top: Biba within B-L; middle: B-L within Biba; bottom: same divisions \\ cannot write; // cannot read; ×× no access; \+/=×

Just as systems based on the Bell-LaPadula model tend to cause all information to move towards higher levels of security by always increasing the level to meet the highest level user, the Biba model tends to move all information towards lower integrity levels by always reducing the integrity of results to that of the lowest incoming integrity. We also know that a precise system for integrity is NP-complete (just as its dual is NP-complete)

The most trusted programmer is (by definition) the programmer that can write programs executable by the most users. In order to maintain the Bell-LaPadula policy, high level users cannot write programs used by lower level users. This means that the most trusted programmers must be those at the lowest security level. This seems contradictory. When we mix the Biba and Bell-LaPadula models, we find that the resulting isolationism secures us from viruses, but does not permit any user to write programs that can be used throughout the system. Somehow, just as we allow encryption or declassification of data to move it from higher security levels to lower ones, we should be able to use program testing and verification to move information from lower integrity levels to higher ones.

Another commonly used policy that partitions systems into closed subsets is the compartment policy used in typical military applications. This policy partitions users into compartments, with each user only able to access information required for their duties. If every user has access to only one compartment at a time, the system is secure from viral attack across compartment boundaries because they are isolated. Unfortunately, in current systems, users may have simultaneous access to multiple compartments. In this case, infection can spread across these boundaries to the transitive closure of information flow

3.3 Flow Models

In policies that do not partition systems into closed proper subsets under transitivity, it is possible to limit the extent over which a virus can spread. The 'flow distance' policy implements a distance metric by keeping track of the distance (number of sharings) over which data has flowed

The rules are; the distance of output information is the maximum of the distances of input information, and the distance of shared information is one more than the distance of the same information before sharing. Protection is provided by enforcing a threshold above which information becomes unusable. Thus a file with distance 8 shared into a process with distance 2 increases the process to distance 9, and any further output will be at least that distance.

The 'flow list' policy maintains a list of all users who have had an effect on each object. The rule for maintaining this list is; the flow list of output is the union of the flow lists of all inputs (including the user who causes the action). Protection takes the form of an arbitrary Boolean expression on flow lists which determines accessibility. This is a very general policy, and can be used to represent any of the above policies by selecting proper Boolean expressions.

As an example, user A could only be allowed to access information written by users (B and C) or (B and D), but not information written by B, C, or D alone. This can be used to enforce certification of information by B before C or D can pass it to A. The flow list system can also be used to implement the Biba and the distance models. As an example, the distance model can be realized as follows:

OR(users $\leqslant$ distance 1)

AND NOT(OR(users $>$ distance 1))

A further generalization of flow lists to flow sequences is possible, and appears to be the most general scheme possible for implementing a flow control policy.

In a system with unlimited information paths, limited transitivity may have an effect if users do not use all available paths, but since there is always a direct path between any two users, there is always the possibility of infection. As an example, in a system with transitivity limited to a distance of 1 it is 'safe' to share information with any user you 'trust' without having to worry about whether that user has incorrectly trusted another user.

3.4 Limited Interpretation

With limits on the generality of interpretation less restrictive than fixed first order interpretation, the ability to infect is an open question because infection depends on the functions permitted. Certain functions are required for infection. The ability to write is required, but any useful program must have output. It is possible to design a set of operations that do not allow infection in even the most general case of sharing and transitivity, but it is not known whether any such set includes non fixed first order functions.

As an example, a system with only the function 'display-file' can only display the contents of a file to a user, and cannot possibly modify any file. In fixed database or mail systems this may have practical applications, but certainly not in a development environment. In many cases, computer mail is a sufficient means of communications and so long as the computer mail system is partitioned from other applications so that no information can flow between them except in the covert channel through the user, this may be used to prevent infection.

Although no fixed interpretation scheme can itself be infected, a high order fixed interpretation scheme can be used to infect programs written to be interpreted by it. As an example, the microcode of a computer may be fixed, but code in the machine language it interprets can still be infected. LISP, APL, and Basic are all examples of fixed interpretation schemes that can interpret information in general ways. Since their ability to interpret is general, it is possible to write a program in any of these languages that infects programs in any or all of them.

In limited interpretation systems, infection cannot spread any further than in general interpretation systems, because every function in a limited system must also be able to be performed in a general system. The previous results therefore provide upper bounds on the spread of a virus in systems with limited interpretation.

3.5 Precision Problems

Although isolationism and limited transitivity offer solutions to the infection problem, they are not ideal in the sense that widespread sharing is generally considered a valuable tool in computing. Of these policies, only isolationism can be precisely implemented in practice because tracing exact information flow requires NP-complete time, and maintaining markings requires large amounts of

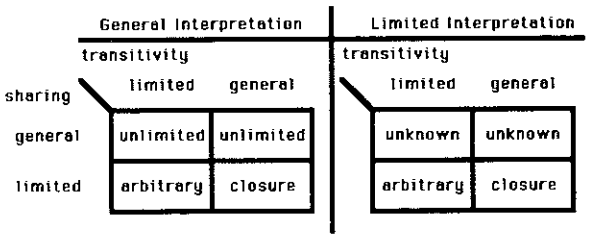


Fig. 5. Limits of viral infection.

space [7]. This leaves us with imprecise techniques. The problem with imprecise techniques is that they tend to move systems towards isolationism. This is because they use conservative estimates of effects in order to prevent potential damage. The philosophy behind this is that it is better to be safe than sorry.

The problem is that when information has been unjustly deemed unreadable by a given user, the system becomes less usable for that user. This is a form of denial of services in that access to information that should be accessible is denied. Such a system always tends to make itself less and less usable for sharing until it either becomes completely isolationist or reaches a stability point where all estimates are precise. If such a stability point existed, we would have a precise system for that stability point. Since we know that any precise stability point besides isolationism requires the solution to an NP-complete problem, we know that any non NP-complete solution must tend towards isolationism.

3.6 Summary and Conclusions

Fig. 5 summarizes the limits placed on viral spreading by the preventative protection just examined. Unknown is used to indicate that the specifics of specific systems are known, but that no general theory has been shown to predict limitations in these categories.

4. Cure of Computer Viruses

Since prevention of computer viruses may be infeasible if sharing is desired, the biological analogy leads us to the possibility of cure as a means of protection. Cure in biological systems depends on the ability to detect a virus and find a way to overcome it. A similar possibility exists for computer viruses. We now examine the potential for detection and removal of a computer virus.

4.1 Detection of Viruses

In order to determine that a given program '*P*' is a virus, it must be determined that *P* infects other programs. This is undecidable since *P* could invoke any proposed decision procedure '*D*' and infect other programs if and only if *D* determines that *P* is not a virus. We conclude that a program that precisely discerns a virus from any other program by examining its appearance is infeasible. In the following modification to program *V* (Fig. 6), we use the hypothetical decision procedure *D* which returns "true" iff its argument is a virus, to exemplify the undecidability of viral detection.

By modifying the main program of *V*, we have assured that, if the decision procedure *D* determines *CV* to be a virus, *CV* will not infect other programs and thus will not act as a virus. If *D* determines that *CV* is not a virus, *CV* will infect other programs and thus be a virus. Therefore, the hypothetical decision procedure *D* is self contradictory, and precise determination of a virus by its appearance is undecidable.

4.2 Evolutions of a Virus

In our experiments, some viruses took under 100 bytes to implement on a general purpose computer. Since we could interleave any program that doesn't halt, terminates in finite time, and does not overwrite the virus or any of its state variables, and still have a virus, the number of possible variations on a single virus is clearly very large. In this example of an evolutionary virus *EV*, we augment *V* by allowing it to add random state-

```
program contradictory-virus :=
{...
main-program :=
  {if ~D(contradictory-virus) then
        {infect-executable;
        if trigger-pulled then
              do-damage;
        }
  goto next;
  }
}
```

Fig. 6. Contradiction of the decidability of a virus 'C

ments between any two necessary statements (Fig. 7).

In general, proof of the equivalence of two evolutions of a program 'P' ('P_1' and 'P_2') is undecidable because any decision procedure 'D' capable of finding their equivalence could be invoked by P_1 and P_2. If found equivalent they perform different operations, and if found different they act the same, and are thus equivalent. This is exemplified by the modification in Fig. 8 to program EV in which the decision procedure D returns "true" iff two input programs are equivalent.

The program UEV evolves into one of two types of programs, P_1 or P_2. If the program type is P_1 the statement labeled "zzz" will become:

if $D(P_1, P_2)$ then print 1;

while if the program type is P_2, the statement labeled "zzz" will become:

if $D(P_1, P_2)$ then print 0;

The two evolutions each call decision procedure D to decide whether they are equivalent. If D indicates that they are equivalent, then P_1 will print a 1 while P_2 will print a 0, and D will be contradicted. If D indicates that they are different, neither prints anything. Since they are otherwise equal, D is again contradicted. Therefore, the hypothetical decision procedure D is self contradictory, and the precise determination of the equivalence of these two programs by their appearance is undecidable.

```
program evolutionary-virus :=
{...
subroutine print-random-statement :=
  {print (random-variable-name, "=",
          random-variable-name);
  loop: if random-bit = 1 then
        {print (random-operator,
                random-variable-name);
        goto loop;}
  print (semicolon);
  }
subroutine copy-virus-with-insertions :=
  {loop: copy evolutionary-virus
              to virus till semicolon;
  if random-bit = 1 then
        print-random-statement;
  if ~end-of-input-file goto loop;
  }
main-program :=
  {copy-with-random-insertions;
  infect-executable;
  if trigger-pulled then do-damage;
  goto next;}
next:}
```

Fig. 7. Evolutionary virus 'EV'.

```
program undecidable-EV :=
{...
subroutine copy-with-undecidable :=
  {copy undecidable-EV to
        file till line-starts-with zzz;
  if file = P1 then
        print ("if D(P1,P2) print 1;");
  if file = P2 then
        print ("if D(P1,P2) print 0;");
  copy undecidable-EV to
        file till end-of-input-file;
  }
main-program :=
  {if random-bit = 0 then file = P1
        otherwise file = P2;
  copy-with-undecidable;
  zzz:
  infect-executable;
  if trigger-pulled then do-damage;
  goto next;}
next:}
```

Fig. 8. Undecidable equivalence of evolutions of a virus 'UEV'

Since both P_1 and P_2 are evolutions of the same program, the equivalence of evolutions of a program is undecidable, and since they are both viruses, the equivalence of evolutions of a virus is undecidable. Program UEV also demonstrates that two unequivalent evolutions can both be viruses.

An alternative to detection by appearance, is detection by behavior. A virus, just as any other program, acts as a surrogate for the user in requesting services, and the services used by a virus are legitimate in legitimate uses. The behavioral detection question then becomes one of defining what is and is not a legitimate use of a system service, and finding a means of detecting the difference.

As an example of a legitimate virus, a compiler that compiles a new version of itself is in fact a virus by the definition given here. It is a program that 'infects' another program by modifying it to include an evolved version of itself. Since the viral capability is in most compilers, every use of a compiler is a potential viral attack. The viral activity of a compiler is only triggered by particular inputs, and thus in order to detect triggering, one must be able to detect a virus by its appearance. Since precise detection by behavior in this case leads to precise detection by the appearance of the inputs, and since we have already shown that precise detection by appearance is undecidable, it follows that precise detection by behavior is also undecidable.

4.3 Limited Viral Protection

A limited form of virus has been designed [24] in the form of a special version of the *C* compiler that can detect the compilation of the login program and add a Trojan horse that lets the author login. Thus the author could access any Unix system with this compiler. In addition, the compiler can detect compilations of new versions of itself and infect them with the same Trojan horse.

As a countermeasure, we can devise a new login program (and *C* compiler) sufficiently different from the original as to make its equivalence very difficult to determine. If the 'best *AI* program of the day' would be incapable of detecting their equivalence in a given amount of time, and the compiler performed its task in less than that much time, it could be reasonably assumed that the virus could not have detected the equivalence, and therefore would not have propagated itself. If the exact nature of the detection were known, it would likely be quite simple to work around it. Once a virus free compiler is generated, the old (and presumably more efficient) version can be recompiled for further use.

Although we have shown that in general it is impossible to detect viruses, any particular virus can be detected by a particular detection scheme. For example, virus *V* could easily be detected by looking for 1234567 as the first line of an executable. If the executable were found to be infected, it would not be run, and would therefore not be able to spread. The program in Fig. 9 is used in place of the normal run command, and refuses to execute programs infected by virus *V*.

Similarly, any particular detection scheme can be circumvented by a particular virus. As an example, if an attacker knew that a user was using the program *PV* as protection from viral attack, the virus *V* could easily be substituted with a virus *V'* where the first line was 123456 instead of 1234567. Much more complex defense schemes and viruses can be examined. What becomes quite evident is that no infection can exist that cannot be detected, and no detection mechanism can exist that cannot be infected.

```
program new-run-command :=
  {file = name-of-program-to-run;
  if first-line-of-file = 1234567 then
    {print ("the program has a virus");
    exit;}
  run file;
  }
```

Fig. 9. Protection from virus V 'PV'

This result leads to the idea that a balance of coexistent viruses and defenses could exist, such that a given virus could only do damage to a given portion of the system, while a given protection scheme could only protect against a given set of viruses. If each user and attacker used identical defenses and viruses, there could be an ultimate virus or defense. It makes sense from both the attacker's point of view and the defender's point of view to have a set of (perhaps incompatible) viruses and defenses.

In the case where viruses and protection schemes do not evolve, this would likely lead to some st of fixed survivors, but program (or virus) that evolves into a diffcult to attack program (or virus) is more likely to survive. As evolution takes place, balances tend to change, with the eventual result being unclear in all but the simplest circumstances. This has very strong analogies to biological theories of evolution [6], and might relate well to genetic theories of diseases. Similarly, the spread of viruses through systems might well be analyzed by using mathematical models used in the study of infectious diseases [2].

Since we cannot precisely detect a virus, we are left with the problem of defining potentially illigitimate use in a decidable and easily computable way. We might be willing to detect many programs that are not viruses and even not detect some viruses in order to detect a large number of viruses. If an event is relatively rare in 'normal' use, it has high information content when it occurs, and we can define a threshold at which reporting is done. If sufficient instrumentation is available, flow lists can be kept which track all users who have affected any given file. Users that appear in many incoming flow lists could be considered suspicious. The rate at which users enter incoming flow lists might also be a good indicator of a virus.

This type of measure can be of value if the services used by viruses are rarely used by other programs, but presents several problems. If the threshold is known to the attacker, the virus can be made to work within it. An intelligent thresholding scheme could adapt so the threshold could not be easily determined by the attacker. Although this 'game' can clearly be played back

and forth, the frequency of infection can be kept low enough to slow the undetected virus without interfering significantly with legitimate use.

Several systems were examined for their abilities to detect viral attacks. Surprisingly, none of these systems even include traces of the owner of a program run by other users. Marking of this sort must almost certainly be used if even the simplest of viral attacks are to be detected.

Once a virus is implanted, it may not be easy to remove. If the system is kept running during removal, a disinfected program could be reinfected. This presents the potential for infinite tail chasing. Without some denial of services, removal is likely to be impossible unless the program performing removal is faster at spreading than the virus being removed. Even in cases where the removal is slower than the virus, it may be possible to allow most activities to continue during removal without having the removal process be very fast. For example, one could isolate a user or subset of users and cure them without denying services to other users.

In general, precise removal depends on precise detection because without precise detection it is impossible to know precisely whether or not to remove a given object. In special cases, it may be possible to perform removal with an inexact algorithm. As an example, every file written after a given date could be removed in order to remove any virus started after that date. This may be quite painful if viruses are designed to have long waiting periods before doing damage, since even backups would have to be discarded to fully cleanse the system.

One concern that has been expressed and is easily laid to rest is the chance that a virus could be spontaneously generated. This is strongly related to the question of how long it will take N monkeys at N keyboards to create a virus, and is laid to rest with similar dispatch.

5. Experiments with Computer Viruses

To demonstrate the feasibility of viral attack and the degree to which it is a threat, several experiments were performed. In each case, experiments were performed with the knowledge and consent of systems administrators. In the process of performing experiments, implementation flaws were meticulously avoided. It was critical that these experiments not be based on implementation lapses but only on fundamental flaws in security policies.

5.1 The First Virus

On November 3, 1983, the first virus was conceived of as an experiment to be presented at a weekly seminar on computer security. The concept was first introduced in this seminar by the author, and the name 'virus' was thought of by Len Adleman. After eight hours of expert work on a heavily loaded VAX 11/750 system running Unix, the first virus was completed and ready for demonstration. Within a week, permission was obtained to perform experiments, and five experiments were performed. On November 10, the virus was demonstrated to the security seminar.

The initial infection was implanted in 'vd', a program that displays Unix structures graphically, and introduced to users via the system bulletin board. Since vd was a new program on the system, no performance characteristics or other details of its operation were known. The virus was implanted at the beginning of the program so that it was performed before any other processing.

Several precautions were taken in order to keep the attack under control. All infections were performed manually by the attacker and no damage was done, only reporting. Traces were included to assure that the virus would not spread without detection, access controls were used for the infection process, and the code required for the attack was kept in segments, each encrypted and protected to prevent illicit use.

In each of five attacks, all system rights were granted to the attacker in under an hour. The shortest time was under five minutes, and the average under 30 minutes. Even those who knew the attack was taking place were infected. In each case, files were 'disinfected' after experimentation. It was expected that the attack would be successful, but the very short takeover times were quite surprising. In addition, the virus was fast enough (under 1/2 second) that the delay to infected programs went unnoticed.

Once the results of the experiments were announced, administrators decided that no further computer security experiments would be permitted on their system. This ban included the planned addition of traces which could track

potential viruses and password augmentation experiments which could potentially have improved security to a great extent. This apparent fear reaction is typical, rather than try to solve technical problems technically inappropriate and inadequate policy solutions are often chosen

After successful experiments had been performed on a Unix system, it was quite apparent that the same techniques would work on many other systems. In particular, experiments were planned for a Tops-20 system, a VMS system, a VM/370 system, and a network containing several of these systems. In the process of negotiating with administrators, feasibility was demonstrated by developing and testing prototypes. Prototypes attacks for the Tops-20 system were developed by an experienced Tops-20 user in six hours, a novice VM/370 user with the help of an experienced programmer in 30 hours, and a novice VMS user without assistance in 20 hours. These programs demonstrated the ability to find files to be infected, infect them, and cross user boundaries

After several months of negotiation and administrative changes, it was decided that the experiments would not be permitted. The security officer at the facility was in constant opposition to security experiments, and would not even read any proposals. This is particularly interesting in light of the fact that it was offered to allow systems programmers and security officers to observe and oversee all aspects of all experiments. In addition, systems administrators were unwilling to allow sanitized versions of log tapes to be used to perform offline analysis of the potential threat of viruses, and were unwilling to have additional traces added to their systems by their programmers to help detect viral attacks. Although there is no apparent threat posed by these activities, and they require little time, money, and effort, administrators were unwilling to allow investigations. It appears that their reaction was the same as the fear reaction of the Unix administrators.

5.2 A Bell-LaPadula Based System

In March of 1984, negotiations began over the performance of experiments on a Bell-LaPadula [4] based system implemented on a Univac 1108. The experiment was agreed upon in principal in a matter of hours, but took several months to become solidified. In July of 1984, a two week period was arranged for experimentation. The purpose of this experiment was merely to demonstrate the feasibility of a virus on a Bell-LaPadula based system by implementing a prototype.

Because of the extremely limited time allowed for development (26 hours of computer usage by a user who had never used an 1108, with the assistance of a programmer who had not used an 1108 in five years), many issues were ignored in the implementation. In particular, performance and generality of the attack were completely ignored. As a result, each infection took about 20 seconds, even though they could easily have been done in under a second. Traces of the virus were left on the system although they could have been eliminated to a large degree with little effort. Rather than infecting many files at once, only one file at a time was infected. This allowed the progress of a virus to be demonstrated very clearly without involving a large number of users or programs. As a security precaution, the system was used in a dedicated mode with only a system disk, one terminal, one printer, and accounts dedicated to the experiment.

After 18 hours of connect time, the 1108 virus performed its first infection. After 26 hours of use, the virus was demonstrated to a group of about 10 people including administrators, programmers, and security officers. The virus demonstrated the ability to cross user boundaries and move from a given security level to a higher security level. Again it should be emphasized that no system flaws were involved in this activity, but rather that the Bell-LaPadula model allows this sort of activity to legitimately take place.

The attack was not difficult to perform. The code for the virus consisted of five lines of assembly code, about 200 lines of Fortran code, and about 50 lines of command files. It is estimated that a competent systems programmer could write a much better virus for this system in under two weeks. In addition, once the nature of a viral attack is understood, developing a specific attack is not difficult. Each of the programmers present was convinced that they could have built a better virus in the same amount of time. (This is believable since this attacker had no previous 1108 experience.)

5.3 Instrumentation

In early August of 1984, permission was granted to instrument a VAX Unix system to measure

sharing and analyze viral spreading. Data at this time is quite limited, but several trends have appeared. The degree of sharing appears to vary greatly between systems, and many systems may have to be instrumented before these deviations are well understood. A small number of users appear to account for the vast majority of sharing, and a virus could be greatly slowed by protecting them. The protection of a few 'social' individuals might also slow biological diseases. The instrumentation was conservative in the sense that infection could happen without the instrumentation picking it up, so estimated attack times are unrealistically slow.

As a result of the instrumentation of these systems, a set of 'social' users were identified. Several of these surprised the main systems administrator. The number of systems administrators was quite high, and if any of them were infected, the entire system would likely fall within an hour. Some simple procedural changes were suggested to slow this attack by several orders of magnitude without reducing functionality.

Two systems are shown in Fig. 10, with three classes of users (*S* for system, *A* for system administrator, and *U* for normal user). '##' indicates the number of users in each category, 'spread' is the average number of users a virus would spread to, and 'time' is the average time taken to spread to them once they logged in, rounded up to the nearest minute. Average times are misleading because once an infection has reached the 'root' account on Unix, all access is granted. Taking this into account leads to takeover times on the order of one minute which is so fast that infection time becomes a limiting factor in how quickly infections can spread. This coincides with previous experimental results using an actual virus.

Users who were not shared with are ignored in these calculations, but other experiments indicate that any user can get shared with by offering a program on the system bulletin board. Detailed analysis demonstrated that systems administrators tend to try these programs as soon as they are announced. This allows normal users to infect system files within minutes. Administrators used their accounts for running other users' programs and storing commonly executed system files, and several normal users owned very commonly used files. These conditions make viral attack very quick. The use of separate accounts for systems administrators during normal use was immediately suggested, and the systematic movement (after verification) of commonly used programs into the system domain was also considered.

5.4 Summary and Conclusions

The Fig. 11 summarizes the results of these and several other experiments. The systems are across the horizontal axis (Unix, Bell-LaPadula,), while the vertical axis indicates the measure of performance (time to program, infection time, number of lines of code, number of experiments performed, minimum time to takeover, average time to takeover, and maximum time to takeover) where time to takeover indicates that all privileges would be granted to the attacker within that delay after introducing the virus.

Viral attacks appear to be easy to develop in a very short time, can be designed to leave few if any traces in most current systems, are effective

System 1

class	#	spread	time
S	3	22	0
A	1	1	0
U	4	5	18

System 2

class	#	spread	time
S	5	160	1
A	7	78	120
U	7	24	600

Fig 10 Summary of spreading

	unixC	B-L	Instr	Shell	VMS	Basic	DOS
time	8hrs	18hrs	N/A	15min	30min	2hrs	1hrs
inf t	.5sec	20sec	N/A	2sec	2sec	15sec	10sec
code	200L	260L	N/A	7L	9L	30L	20L
trials	5	N/A	N/A	N/A	N/A	N/A	N/A
min t	5min	N/A	30sec	N/A	N/A	N/A	N/A
avg t	30min	N/A	30min	N/A	N/A	N/A	N/A
max t	60min	N/A	48hrs	N/A	N/A	N/A	N/A

Fig 11 Experimental results

against modern security policies for multilevel usage, and require only minimal expertise to implement. Their potential threat is severe, and they can spread very quickly through a computer system. It appears that they can spread through computer networks in the same way as they spread through computers, and thus present a widespread and fairly immediate threat to many current systems.

The problems with policies that prevent controlled security experiments are clear; denying users the ability to continue their work promotes illicit attacks; and if one user can launch an attack without using system bugs or special knowledge, other users will also be able to. By simply telling users not to launch attacks, little is accomplished. Users who can be trusted will not launch attacks but users who would do damage cannot be trusted, so only legitimate work is blocked. The perspective that every attack allowed to take place reduces security is, in the author's opinion, a fallacy. The idea of using attacks to learn of problems is even required by government policies for trusted systems [16,17]. It would be more rational to use open and controlled experiments as a resource to improve security.

6. Summary, Conclusions, and Further Work

To quickly summarize, absolute protection can be easily attained by absolute isolationism, but that is usually an unacceptable solution. Other forms of protection all seem to depend on the use of extremely complex and/or resource intensive analytical techniques, or imprecise solutions that tend to make systems less usable with time.

Prevention appears to involve restricting legitimate activities, while cure may be arbitrarily difficult without some denial of services. Precise detection is undecidable, however, statistical methods may be used to limit undetected spreading either in time or in extent. Behavior of typical usage must be well understood in order to use statistical methods, and this behavior is liable to vary from system to system. Limited forms of detection and prevention could be used in order to offer limited protection from viruses.

It has been demonstrated that a virus has the potential to spread through any general purpose system which allows sharing. Every general purpose system currently in use is open to at least limited viral attack. In many current 'secure' systems, viruses tend to spread further when created by less trusted users. Experiments show the viability of viral attack, and indicate that viruses spread quickly and are easily created on a variety of operating systems. Further experimentation is still underway.

The results presented are not operating system or implementation specific, but are based on the fundamental properties of systems. More importantly, they reflect realistic assumptions about systems currently in use. Further, nearly every 'secure' system currently under development is based on the Bell-LaPadula or lattice policy alone, and this work has clearly demonstrated that these models are insufficient to prevent viral attack. The virus essentially proves that integrity control must be considered an essential part of any secure operating system.

Several undecidable problems have been identified with respect to viruses and countermeasures. Several potential countermeasures were examined in some depth, and none appear to offer ideal solutions. Several of the techniques suggested in this paper which could offer limited viral protection are in limited use at this time. To be perfectly secure against viral attacks, a system must protect against incoming information flow, while to be secure against leakage of information a system must protect against outgoing information flow. In order for systems to allow sharing, there must be some information flow. It is therefore the major conclusion of this paper that the goals of sharing in a general purpose multilevel security system may be in such direct opposition to the goals of viral security as to make their reconciliation and coexistence impossible.

The most important ongoing research involves the effect of viruses on computer networks. Of primary interest is determining how quickly a virus could spread to a large percentage of the computers in the world. This is being done through simplified mathematical models and studies of viral spreading in 'typical' computer networks. The implications of a virus in a secure network are also of great interest. Since the virus leads us to believe that both integrity and security must be maintained in a system in order to prevent viral attack, a network must also maintain both criteria in order to allow multilevel sharing between com-

puters. This introduces significant constraints on these networks

Significant examples of evolutionary programs have been developed at the source level for producing many evolutions of a given program A simple evolving virus has been developed, and a simple evolving antibody is also under development.

Acknowledgements

Because of the sensitive nature of much of this research and the experiments performed in its course, many of the people to whom I am greatly indebted cannot be explicitly thanked. Rather than ignoring anyone's help, I have decided to give only first names. Len and David have provided a lot of good advice in both the research and writing of this paper, and without them I would likely never have gotten it to this point. John, Frank, Connie, Chris, Peter, Terry, Dick, Jerome, Mike, Marv, Steve, Lou, Steve, Andy, and Loraine all put their noses on the line more than just a little bit in their efforts to help perform experiments, publicize results, and lend covert support to the work. Martin, John, Magdy, Xi-an, Satish, Chris, Steve, JR, Jay, Bill, Fadi, Irv, Saul, and Frank all listened and suggested, and their patience and friendship were invaluable. Alice, John, Mel, Ann, and Ed provided better blocking than the USC front 4 ever has.

References

[1] J.P. Anderson: *Computer Security Technology Planning Study* Technical Report ESD-TR-73-51, USAF Electronic Systems Division, Oct, 1972. Cited in Denning.

[2] Norman T.J. Baily: *The Mathematical Theory of Epidemics* Hafner Publishing Co., N.Y., 1957.

[3] D.B. Baker: *Department of Defense Trusted Computer System Evaluation Criteria (Final Draft)* Private communication, The Aerospace Corporation, 1983

[4] D.E. Bell and L.J. LaPadula: *Secure Computer Systems: Mathematical Foundations and Model* The Mitre Corporation, 1973. Cited in many papers.

[5] K.J. Biba: *Integrity Considerations for Secure Computer Systems.* USAF Electronic Systems Division, 1977 Cited in Denning.

[6] Richard Dawkins: *The Selfish Gene* Oxford Press, N.Y., N.Y., 1978.

[7] D.E. Denning: *Cryptography and Data Security* Addison Wesley, 1982

[8] A.D. Dewdney: Computer Recreations *Scientific American* 250(5): 14–22, May, 1984.

[9] R.J. Feiertag and P.G. Neumann: The Foundations of a Provable Secure Operating System (PSOS). In *National Computer Conference*, pages 329–334. AIFIPS, 1979

[10] J.S. Fenton: *Information Protection Systems* PhD thesis, U. of Cambridge, 1973. Cited in Denning.

[11] M.R. Garey and D.S. Johnson: *Computers and Intractability.* Freeman, 1979

[12] B.D. Gold, R.R. Linde, R.J. Peeler, M. Schaefer, J.F. Scheid, and P.D. Ward: A Security Retrofit of VM/370 In *National Computer Conference*, pages 335–344. AIFIPS, 1979

[13] Gunn, ACM: *Use of Virus Functions to Provide a Virtual APL Interpreter Under User Control*, 1974

[14] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman: Protection in Operating Systems In *Proceedings* ACM, 1976

[15] L.J. Hoffman: Impacts of information system vulnerabilities on society. In *National Computer Conference*, pages 461–467. AIFIPS. 1982.

[16] U.S. Dept. of Justice, Bureau of Justice Statistics: *Computer Crime – Computer Security Techniques.* U.S. Government Printing Office, Washington, DC, 1982.

[17] M.H. Klein: *Department of Defense Trusted Computer System Evaluation Criteria* Department of Defense, Fort Meade, Md. 20755. 1983.

[18] B.W. Lampson: A note on the Confinement Problem In *Communications* ACM, Oct, 1973.

[19] C.E. Landwehr: The Best Available Technologies for Computer Security *Computer* 16(7), July, 1983

[20] R.R. Linde: Operating System Penetration. In *National Computer Conference*, pages 361–368. AIFIPS, 1975.

[21] E.J. McCauley and P.J. Drongowski: KSOS – The Design of a Secure Operating System. In *National Computer Conference*, pages 345–353. AIFIPS, 1979

[22] G.J. Popek, M. Kampe, C.S. Kline, A. Stoughton, M. Urban, and E.J. Walton: UCLA Secure Unix In *National Computer Conference*. AIFIPS, 1979

[23] Schochaud, Hupp, ACM: *The 'Worm' Programs – Early Experience with a Distributed Computation*, 1982.

[24] K. Thompson, ACM: *Reflections on Trusting Trust*, 1984.

[25] J.P.L. Woodward: Applications for Multilevel Secure Operating Systems In *National Computer Conference*, pages 319–328. AIFIPS, 1979.