布隆过滤器在 NIST RDS 和硬盘诊断方面的实际应用

非官方中文译文·安天技术公益翻译组 译注

文档信息			
原文名称	Practical Applications of Bloom Filters to the NIST RDS and Hard Drive Triage		
原文作者	Paul Farrell, Simson L. Garfinkel, Douglas White	**原文发布日期**	2008 年 ACSAC（年度计算机安全应用大会）
作者简介	Simson L. Garfinke 是美国海军研究生院计算机科学院的副教授，其研究方向是计算机安全、隐私、安全应用和数字取证。 http://simson.net/page/Curriculum_Vitae Douglas White 自 1987 年任职于 NIST（美国国家标准与技术研究所），带领着 NSRL（国家软件参考库）项目。其研究领域包括分布式系统、分布式数据库和电信协议、实时生物监测、实时视频处理、系统管理和网络监控。 http://www.linkedin.com/profile/view?id=24752856&authType=NAME_SEARCH&authToken=kTO7&locale		
原文发布单位	美国海军研究生院，美国国家标准与技术研究所		
原文出处	http://simson.net/clips/academic/2008.ACSAC.Bloom.pdf		
译者	安天技术公益翻译组	**校对者**	安天技术公益翻译组
免责声明	• 本译文译者为安天实验室工程师，本文系出自个人兴趣在业余时间所译，本文原文来自互联网的公共方式，译者力图忠于所获得之电子版本进行翻译，但受翻译水平和技术水平所限，不能完全保证译文完全与原文含义一致，同时对所获得原文是否存在臆造、或者是否与其原始版本一致未进行可靠性验证和评价。 • 本译文对应原文所有观点亦不受本译文中任何打字、排版、印刷或翻译错误的影响。译者与安天实验室不对译文及原文中包含或引用的信息的真实性、准确性、可靠性、或完整性提供任何明示或暗示的保证。译者与安天实验室亦对原文和译文的任何内容不承担任何责任。翻译本文的行为不代表译者和安天实验室对原文立场持有任何立场和态度。		

	• 译者与安天实验室均与原作者与原始发布者没有联系，亦未获得相关的版权授权，鉴于译者及安天实验室出于学习参考之目的翻译本文，而无出版、发售译文等任何商业利益意图，因此亦不对任何可能因此导致的版权问题承担责任。 • 本文为安天内部参考文献，主要用于安天实验室内部进行外语和技术学习使用，亦向中国大陆境内的网络安全领域的研究人士进行有限分享。望尊重译者的劳动和意愿，不得以任何方式修改本译文。译者和安天实验室并未授权任何人士和第三方二次分享本译文，因此第三方对本译文的全部或者部分所做的分享、传播、报道、张贴行为，及所带来的后果与译者和安天实验室无关。本译文亦不得用于任何商业目的，基于上述问题产生的法律责任，译者与安天实验室一律不予承担。

布隆过滤器在 NIST RDS 和硬盘诊断方面的实际应用

Paul Farrell
加利佛尼亚州，蒙特利尔，
美国海军研究生院

Simson L. Garfinkel
加利佛尼亚州，蒙特利尔，
美国海军研究生院

Douglas White
马里兰州，盖瑟斯堡，
美国国家标准与技术研究所

摘要

近年来，研究人员花费了大量精力根据已知文件创建大型哈希码集合。由于这些集合越来越大，所以分布它们也越来越困难。同时，随着硬盘存储容量急剧增加，这些集合排除分析“已知的良好文件”的价值也降低了。

本文评估了利用 BF（布隆过滤器）来分布 NSRL（国家软件参考库）RDS（参考数据集）版本 2.19 的情况，其中涉及 1300 万个 SHA-1 哈希值。我们提出了一个开源参考 BF 实现并用大量磁盘映像进行验证。我们讨论了过滤器的调整，探讨了如何用它们启动新的取证功能，并提出了针对布隆过滤器的新型攻击。

1. 简介

之前的研究已经确定了布隆过滤器是一个很有用的表示哈希集合的工具，而且能够将误差率保持到最低[17]。NIST（美国国家标准与技术研究所）发布了一个用 Perl 编写的样本布隆过滤器，以及两个包含 NIST RDS 2.13 子集的布隆过滤器[20]。然而，迄今为止还未发布任何关于速度优化的大型 BF 实现的研究；BF 的磁盘表示尚未标准化；BF 还未被公开纳入开源取证工具中。

与此同时，NSRL RDS 持续增长[19]，2008 年 7 月的 RDS 2.21 版本将 47,553,722 个已知文件映射到 14,563,184 个独特的哈希值。NIST 还宣布，它打算公布急剧增加的哈希（每个文件的每个 512 字节块的哈希值）集合[20]。

存储这些 1400 万 SHA1 哈希值需要 291 MB 字节的空间（在当今世界这样的存储空间是小菜一碟），但是伴随这些哈希值的辅助资料将 RDS 增加到了 6 GB，这使得这些文件难以分布和处理。搜索数据库的时间也增加了，因为大多数工具使用二进制搜索或某种索引树，其存取速度与数据集大小的对数成正比。我们用性能很好的参考硬件进行测试，使用 SleuthKit hfind 命令每秒只能进行 17,000 到 85,000 次 RDS 哈希查询[1]；当哈希值存储在 MySQL InnoDB 数据库时，每秒只能进行 4,000 次查询。

布隆过滤器是一个很有吸引力的处理大型哈希集合的方法。在本文中，我们提出了一种新的 BF 实现，可以在同样的硬件上每秒执 98,000 到 220 万次查询。[2]

所述 RDS 包括每个文件的大量元数据，包括文件名、大小、其分布的软件包和操作系统，以及发布者。SleuthKit hfind 命令、Guidance Software EnCase [11]，或我们评估的其他工具都不适用这些元数据。这些元数据在很大程度上未被使用，一旦有效地访问，就能够协助新硬盘的快速分析和诊断。

1.1 本文的贡献

本文参考了 BF 的先前研究，并将其扩展到 NSRL（国家软件参考库）RDS（参考数据集），特别是：

- 我们提出了 nsrl_bloom，一种以 C 语言编写的新型、高效、高度可配置的开源布隆过滤器实现。
- 我们修改了基于 SleuthKit 的自动化取证工具 fiwalk，将其用于我们的 BF 实现以便自动排除“已知的良好文件”。
- 我们评估用不同参数创建的 BF 的性能，并将 NSRL RD 实际结果与理论预测结果进行比较。
- 我们评估 BF 的性能，方法是将在平面文件（SleuthKit 使用）中查询和在大型 MySQL 数据库中的哈希查询相比较。
- 在新安装 Windows 2000、XP 和 Vista 的情况下，评估 RDS 的覆盖范围。
- 我们提出了一种针对使用 BF 排除“已知的良好文

1 The range results from the fact that hfind's binary search algorithm terminates early when it finds a hash that is in the database. As a result, looking up a hash that is in the data set is roughly 5 times faster than looking up a hash that is not. Surprisingly, MySQL exhibits similar per- formance for both kinds of hashes.

2 Once again, the range is the result of the difference in time between looking up a hash that is not in the database and one that is. Unlike binary searches on sorted data, BFs terminate faster when searching for data that is not present.

件”的新型攻击。

- 我们对 BF 用于跨磁盘分析和提取特征的分布进行了评估。

1.2 相关研究

Bloom 在 1970 年引入了“容许误差的哈希码”理论[2]。Dillinger 和 Manolios 展示了如何实现快速、准确、节省内存、可扩展的和灵活的布隆过滤器[8]。Fan 等人引入了计数布隆过滤器，其中小的整数被存储在向量中而非各个位中[9]。Bloomier 过滤器[5]采用分层的布隆过滤器，将各条目映射到多个集合之一。Broder 提出了计算哈希函数最佳数量的方程，以便实现最低的误报率[3]。Manolios 提供了一个简单的在线计算器来计算这些值[14]。

至今，BF（布隆过滤器）已被应用于各种取证程序[3]。Roussev 等人提出利用 BF 来存储文件哈希值[17]。研究人员还指出通过对文件各部分进行哈希计算，来检测对象版本管理，并指出“使用相对高误报率（>15%）和低每单元比特数（3-5）的过滤器也有可能识别对象版本管理”。不幸的是，他们的项目 md5bloom 的源代码从来没有公布。2007 年，Roussev 等人用多分辨率相似散列扩展了这一研究，用来检测相似但不同的文件[18]。

White 用 Perl 编写了一个样本 BF 实现并通过 NIST 网站予以公布[19]。此代码是有缺陷的，MD5 或 SHA1 代码的每一个位都被多次用来计算多个布隆哈希函数。此外，因为它是用 Perl 编写的，过度的内存耗用使其无法诊断整个 RDS。

1.3 本文大纲

第 2 章讨论 BF（布隆过滤器）和我们的 BF 实现。第 3 章讨论将我们的 BF 实现应用于 NSRL RDS 的经验。第 5 章讨论本文对主流取证研究和实践的影响。第 6 章总结全文。

2. 高性能布隆过滤器

2.1 布隆过滤器介绍

根本上讲，BF（布隆过滤器）是一种数据结构，允许多个值 $V_0 \ldots V_n$ 存储于单个有限位的向量 F 中。因此，BF 支持两个原始操作：将新的值 V 存储于过滤器 F；查询值 V‘是否存在于 F。

BF 计算 V 的哈希值，将该哈希值置于 0 和 m 之间的序列中，从而生成位 i。之后，该位被设置于大小为 m 的位向量 F 中。要查询 V’是否存在，需要计算 V’的哈希值，并生成位 i‘。如果过滤器中没有设置这样的 i’，则 V’不可能存储于过滤器中。

如果位数 i’被设置了，可能是因为 V’之前被存储在过滤器中了。同样地，如果另一个值 V’’的哈希值位数 i’’与 i’相同，则 V‘和 V’‘是彼此的别名；过滤器用户无法确定这两个值中的哪个是先存储的。由于这一性质，据说 BF 能够提供概率数据检索：如果 BF 认为一个值没有被存储在过滤器中，那么该值一定没有被存储。但如果 BF 认为一个值被存储在过滤器中，那么该值有可能被存储了；同样地，别名可能也被存储了。存储于 BF 的信息越多，则别名和误报的概率也会随之增加。

在实践中，多个哈希函数 $f_1 \ldots f_k$ 将单个值 V 存储于过滤器 F 中。这样的位数排列被称为一个元素。存储数据要求在过滤器向量 F 中设置 k 个不同的位（$i_1 \ldots i_k$），而查询则需要检查这些位是否被设置了。

因此，BF 可以用 4 个参数来描述：

m：过滤器中的位数（在本文中，我们用 M 来表示 $\log_2(m)$）；

k：每个元素的哈希函数的数量；

b：过滤器中每个元素设置的位数；

f：哈希函数。

一旦数据被存储在过滤器中，其他参数也可用来描述过滤器的状态：

n：过滤器中存储的元素的数量；

p: 被报告存储于过滤器的值 V 真得存储于过滤器的概率。

正如其他研究所述 [16，15，17]，过滤器中的位没有被设置的概率是：

$$P_0 = (1 - 1/m)^{kn} \quad (1)$$

这近似于：

$$P_0 = e^{-kn/m} \quad (2)$$

理论的误报率是：

$$P_{fp} = (1 - e^{-kn/m})^k \quad (3)$$

2.2 性能特征

BF（布隆过滤器）与 HT（哈希表）类似，两者都是能够紧密存储大量稀疏值的单个数据结构。一旦存储，无论存储的数据量如何，该结构都允许查询某个值是否存在。与哈希表相比，布隆过滤器的优势是能够在很小的空间中存储数据，其劣势在于检索的概率性：给定的布隆过滤器可能认为数据存在但实际上数据却并不存

在。

BF 和 HT 的另一个共同点是引用局部性[7]，这是因为数据在整个结构中进行散列。因为内存缓存和内存层级的性能特征，现代计算机可以实现显著的性能优势，能够将数据结构的内存占用降到最低。对于像 BF 和 HT 这样不以任何预测顺序访问内存的结构来说尤其如此；除了将数据结构缩减到缓存中，没有其他任何方法能够实现引用局部性。

我们以基于英特尔酷睿 2 双核微处理器的现代 Macintosh iMac 台式机为例。处理器运行时内部时钟速度是 2.4 GHz。因为没有引用局部性，忽略 BF 散列和记录的负载，则访问每个 BF 位的速度将大致等于存储元素的内存系统的速度（表 1）。完全适合计算机 32K L1 数据缓存的 16K BF 能够每秒查询约 4000 万次和 20 个函数（缓存的其余部分用于之前所提到的 BF 散列和记录）。另一方面，500 MB 的完全适合主内存的过滤器只能支持每秒 71 万次哈希查询，这是因为计算机的高性能内存子系统仍然需要 70 纳秒来读取每个散列位。

大多数内存系统是流水线式的，允许任何时间同时进行多个读取。此外，酷睿 2 双核可以每周期执行 4 条指令。这一功能可用来执行记录活动，例如递增循环计数器和移位；最终该线程将停止，直到从内存子系统中读取了所请求的位。如果我们能够减少内存读取的延迟和每次查询所需的读取次数，就可以显著加快哈希查询的速度。通过改变 BF 的哈希函数的大小和数量，我们可以优化数据集合的表示。

2.3 nsrl_bloom

White 的原始代码[19]有个缺陷，导致 MD5 或 SHA1 代码的每一个位都被多次用来计算多个布隆哈希函数。其结果是，这些位相互关联，用 perl 编写的 BF 的误报比理论预计的多 10 个。为了使 BF 更加有效，哈希函数必须是真正随机和独立的。为了避免这种相关性，我们根据 BF 的大小简单地将哈希分为几个片段：k = 4 和 M = 28 的 BF 使用 SHA1 的前 28 个位，将其用于布隆哈希函数 f_1；其后的 28 个位用于 f_2，等等。由于这些哈希函数都很强大，所以这些位彼此并不相关。结果是，通过 MD5 创建的 BF 的 k×M 必须小于 128，通过 SHA1 哈希创建的 BF 的 k×M 必须小于 160。

从这个代码库开始，我们实现了快速的、可配置的、用 C 和 C++编写的布隆过滤器。过滤器向量被存储于二进制文件中，每个文件的前 4096 字节包含过滤器参数和注释。这一实现有简单可用的 API（应用程序编程接口），其中包括 6 个 C 函数：

- next_bloom_create（）：创建指定哈希大小、M、k 和可选注释的布隆过滤器。该布隆过滤器可驻留在内存中或备份到文件中。
- nsrl_bloom_open（）：打开一个先前创建的布隆过滤器，读取文件的参数。
- nsrl_bloom_add（）： 向布隆过滤器添加一个哈希值。
- nsrl_bloom_query（）：查询布隆过滤器的哈希组成。
- nsrl_bloom_set_encryption_passphrase-：添加字符串作为布隆过滤器的加密密码（计算字符串的哈希值；在未来的添加和查询中，这被用作 HMAC（哈希运算消息认证码的密））。
- nsrl_bloom_free（）： 释放与布隆过滤器有关的内存。

我们的 C 实现使用 memmap 将 BF 向量映射到内存中，以便进行快速查询；在实践中，这意味着，计算机的虚拟内存子系统根据需要将位向量映射到内存并且不会执行不必要的复制。如果整个 BF 都是必要的，则可以一次性地将过滤器映射到 RAM（随机存取存储器），以尽量减少硬盘驱动器的延迟。

除了正确的哈希函数，这个新的实现比原始 perl 版本更快速和节省内存，使得我们可在完整的 RDS 上进行测试。

该实现还包含 C++类代码，允许对 C API 的零负载存取。

3. 布隆过滤器用于 RDS

在完成 BF 实现后，我们创建了大量包含 NSRL RDS 的 SHA 1 哈希值的布隆过滤器。

3.1 构建过滤器

RDS 作为 ISO 9660 格式的四个 ISO 图像分布。每个图像都包含多个文本文件。RDS 的详细信息可在网上查询。[19]

我们从 NIST 网站上下载了 RDS 2.19 的 ISO 图像（本文编写时 RDS 2.20 尚未发布）。每个 ISO 图像包含若干文本文件和一个 ZIP 文件（ZIP 文件又包含多个文本文件）。我们用两个程序处理这些图像：第一个是 nsrlutil.py，这是一个 Python 程序，它将磁盘映像作为文件安装在 Linux 服务器上，打开 ZIP 压缩文件，将哈

希发送至标准出口；另一个是 bloom，该程序利用命令行中的参数创建新的布隆过滤器，然后使用从标准入口读取的哈希码加载过滤器。通过把这两个出现结合起来，我们可以快速创建大量的 BF 文件，每一个文件都有其具体的参数集。

Memory System	Size	Cycle time	Latency	Time to access 10,000 random bits	# hash lookups per second
L1 Data Cache	32K	3 cycles	1.25 ns	12.5 μs	40,000,000
L2 Cache	4MB	14 cycles	5.83 ns	58.3 μs	8,500,000
667 Mhz DDR2 SDRAM	4GB	5-5-5-15	70 ns	700 μs	710,000
Disk	1000GB	n/a	8.5 ms	85 s	6

表 1：从现代 iMac 计算机（2.4GHz 的英特尔酷睿 2 双核处理器 E6600）的不同内存子系统中存储的布隆表或哈希表中访问位的相对速度。内存延迟的信息请参见[6,12]。磁盘访问时间大约是 8.5ms。一次“哈希查询”需要访问 20 个随机的位。

3.2 精度和验证

我们的目标是创建足够小的 RDS（参考数据集）BF（布隆过滤器），使其能够用于 CD 和旧机器/更小的 PDA 式设备的主内存中。我们随机决定评估 32 MB、64 MB、128 MB、256 MB 和 512 MB 的 BF。这样的 BF 可以通过 2^{28} 到 2^{32} 个一位元素创建（M =28…32）。但每个元素究竟能使用多少个哈希函数？也就是说，k 的最优值是什么？是否有必要选择最优值？

我们知道所希望的过滤器尺寸，也知道 RDS 2.19 具有 13,147,812 个独特的哈希值，我们将这些数字代入最优过滤器方程，发现 512 MB 的过滤器需要 226 个哈希函数，误报率为 6.89×10^{-69}。显然，这种误报率远低于所需的水平，例如，它明显小于存储过滤器的硬盘或数字媒体的故障率。另外，160 位的哈希值中没有足够的熵为 226 个哈希函数提供数据（即向每个函数提供 32 位不相关的输出）。

鉴于只能为哈希函数提供 160 位的数据，所以在 RDS 的现实要求中，需要为参数 k 和 m 选择正确的值。因为 BF 中的每个位对应于唯一的哈希值，当 k = 1 且 m = 2^{160} 时，则不会产生任何误报；但这样的 BF 也不可能很大。好消息是，当 k = 5，m = 2^{32} 时，我们的样本集中并没有出现任何误报；这些设置允许 SHA-1 值的 160 位用于 BF 计算（5×32= 160）。理论误报率是 6.2×10^{-16}。这种大小的布隆过滤器可以方便地下载，存储在 USB 记忆棒和 32 位工作站的内存中。当 k = 4 时，误报率也类似（并且实现速度稍快），但是 k = 5 使得我们的 B 能够容纳更多的信息，而不会降低精度。

为了验证我们的代码，我们编写了回归分析程序，用 1 百万个 RDS 哈希值和 1 百万个非 RDS 哈希值测试每个 BF。根据 BF 的理论，在任何情况下是 RDS 中的值都不会出现在 BF 中。但是当参数 k 和 m 的值小于我们推荐的数值（表 2）时，也会出现偶尔的误报。

3.3 性能

在本节中，我们比较在分类文本文件和 MySQL 数据库（这两种系统被当今的取证工具广泛用于存储 RDS 哈希码）中查询 BF 中哈希值的性能。我们也探讨了调整参数 k 和 m 对 BF 性能的影响。

3.3.1 BF vs hfind 和 MySQL

在 BF 中，每个匹配查询执行 f 操作，而每个非匹配查询则执行 1…f 操作。存储在文本文件中分类的相同数据和执行二进制搜索大致需要 $\log_2(n)$ 操作。MySQL 被配置为使用 InnoDB 表，这样可以实现高性能的操作处理，而且具有行级锁和多版本并发控制。数据被存储在 B 树中。[13]

我们使用红帽 FC6 服务器（两个四核 Xeon 处理器，2MB L2 缓存，3.2GHz 和 8GB RAM）进行测试。用 gcc 4.1.2 编译代码并将代码运行于 2.6.22.9-61 内核。

RDS 作为四个 ISO 映像分布。我们将这些映像的所有哈希值整合到一个文件中。这个 RDS 2.19 文件被导入到干净的 BF（m = 2^{32}，k = 5）、SleuthKit[4]（hfind 使用命令 hfind -i nsrl-sha1 flatfile.txt），以及位于测试机上的单个 MySQL 数据库中，以便减少网络延迟负载对测试结果的影响。

	m (BF size, in bits (MB))										
	2^{22} (512KB)	2^{23} (1MB)	2^{24} (2MB)	2^{25} (4MB)	2^{26} (8MB)	2^{27} (16MB)	2^{28} (32MB)	2^{29} (64MB)	2^{30} (128MB)	2^{31} (256MB)	2^{32} (512MB)
k	**Predicted number of false positives for 1 million random values:**										
1	956,486	791,401	543,274	324,184	177,920	93,314	47,799	24,192	12,170	6,081	3,045
2	996,217	914,866	626,315	295,146	105,096	31,656	8,707	2,285	585	148	37
3	999,753	973,016	740,548	330,423	87,780	16,510	2,552	355	47	6	1
4	999,986	992,448	836,980	392,271	87,111	11,045	1,002	76	5	0	0
5	999,999	998,027	904,503	467,768	95,013	8,708	484	20	1	0	0
6	1,000,000	999,506	946,760	548,411	109,179	n/a	n/a	n/a	n/a	n/a	n/a
7	1,000,000	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	
k	**Actual rate of false positives for 1 million random values:**										
1	956,483	791,511	543,165	323,971	178,416	93,558	48,042	24,448	12,373	6,213	3,156
2	996,126	914,938	625,765	295,401	105,465	31,921	8,735	2,282	582	144	48
3	999,758	972,802	740,271	330,808	88,079	16,518	2,556	378	46	3	0
4	999,989	992,175	836,379	392,427	87,606	10,956	1,049	60	4	0	0
5	999,999	997,916	904,260	467,288	95,662	8,755	479	15	0	0	0
6	1,000,000	999,463	946,587	548,083	109,982	n/a	n/a	n/a	n/a	n/a	n/a
7	1,000,000	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	

表 2 在加载了 1310 万 RDS 2.19 哈希值的 BF 中查询 1 百万个伪随机哈希值 ,出现的预测和实际误报数量。表左侧的 1,000,000 表示每个哈希值都是误报；表右侧的 0 表明没有误报。n/a 表示无法计算（因为 160 位的 SHA-1 没有足够的位数）。此分析表明 m = 2^{32}，k = 5 似乎是存储 SHA-1 值的布隆过滤器的最佳值。

我们测量了两组查询所花费的时间。第一组是从 RDS 2.19 中获取的 1 百万个 SHA1 哈希值 ,这些哈希值一定在数据库中。下一步，我们测量执行 1 百万个哈希值（这些哈希值不在数据库中）的伪随机查询所花费的时间。通过测量的时间，我们可以得知每秒所执行的查询数量（图 1）。

正如预期的，BF 的查询速度快于 SleuthKit 分类文本文件二进制查询和 MySQL InnoDB 表查询。也正如所料，查询不在 BF 中的哈希值显著快于查询在 BF 中的哈希值，这是因为当搜索例程从向量 F 中检索到第一个未设置位 i’时，就会停止搜索。

3.3.2 m 对速度的影响

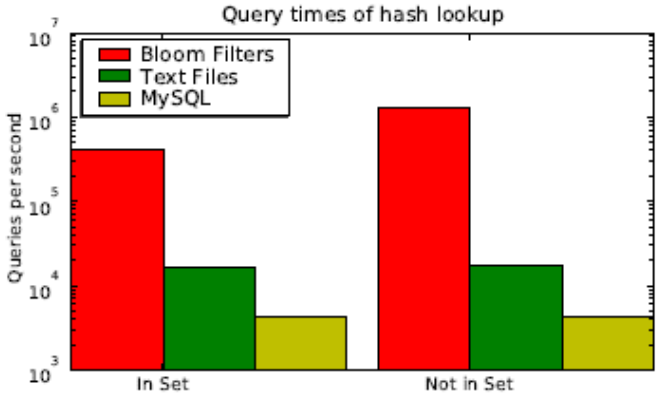
在 2.2 节中 ,我们提出 ,由于 L1 和 L2 缓存的性能，小型 BF 在现代硬件上具备更好的性能。为了验证这一点，我们构建了多个过滤器，其中 k = 5，m 的范围是 2^8 到 2^{32}。然后，我们在哈希值中插入 1 百万个伪随机值并搜索每一个伪随机值。[3]

这种测量 L1 和 L2 缓存影响的方法并不怎么高明，因为这些测试是在运行 Linux 的连网多用户机器上执行的，而且还有很多其他进程也在争夺缓存。另一方面，这种配置类似于大多数从业者使用的机器：即同时运行多个任务的复杂操作系统。尽管如此，我们确实发现当过滤器的大小增加时 ,BF 查询性能会显著降低(图 2)。图 2 还展示了图形大小达到基准系统的 L2 缓存时的拐点，如图中虚线所示。

3.3.3 k 对速度的影响

在 3.1 节中，我们指出，哈希函数数量少的 BF（即 k 值较小）具备较高的性能，这是因为每个哈希函数和从内存中检索相应位都会产生计算负载。为了确定这种影响，我们构建了多个过滤器，其中 m = 2^{20}，k 的范围是 1 到 5。然后，将 1 百万个伪随机值插入哈希值中并查询每个伪随机值。正如预期的那样，当 k 增加时 BF 的性能会降低。的确，每秒查询次数大致与$\frac{1}{k}$成正比（图 3）。

3 Searching for known values is the slowest operation for our BF implementation; searching values not to be present with a BF of k = 5 has the same performance as searching values not to be present in a filter of k = 1,but with the same constant overhead. Since we were interested in measuring the performance of the BF and not the overhead of our implementation, we used k = 5.



Method	In Set	Not in Set
Bloom Filters	424,808	1,314,060
SleuthKit	16,942	17,595
MySQL InnoDB	4,415	4,369

图 1：M = 32，k = 5 的情况下，布隆过滤器、SleuthKit hfind 和 MySQL SELECT 语句（有 InnoDB 表）对 160 位哈希值的每秒查询。“In Set”指的是哈希值在 RDS 2.19 中，而“Not in Set”指的是哈希值不在 RDS 2.19 中。

3.4 用 fiwalk 进行批量取证

我们已经将我们的 BF 实现纳入了开源批量磁盘取证分析程序 fiwalk。fiwalk 使用 Carrier 的 SleuthKit 来执行批量分析，并从要分析的磁盘映像的所有分区中提取被分配和删除的文件。fiwalk 输出一个 walk 文件，该文件包括所有文件的列表、文件的元数据，还有可能包括文件的 MD5 或 SHA1 加密哈希值。

Fiwalk 的当前版本允许通过文件名或扩展名来指定文件。我们修改了 fiwalk，使其可以使用 BF 也可以不使用 BF。我们进一步修改 fiwalk，以便它能够基于磁盘映像上发现的文件来生成 BF。

3.5 RDS 覆盖 Windows 系统

我们创建了虚拟机 Windows 2000 Service Pack 4、Windows XP Service Pack 2 和 Windows Vista Business。通过 qemu-img[1]将.vmdk 文件转换为原始文件，然后用 fiwalk 处理原始文件来生成“walk”文件（包含磁盘映像中的所有文件和文件的 SHA1 哈希值）。之后，将 walk 文件与基准 RDS v2.19 布隆过滤器进行对比。接下来，我们用 Microsoft Update 的最新补丁修复所有虚拟机并重新处理虚拟机。虚拟机中的文件出现于 RDS 2.19 的百分比参见表 3。

这产生了一些有趣的数据点。首先，即使是安装了最新补丁的基本操作系统，只有 60-70%的文件出现于 RDS 中。为什么只有这么多呢？这是因为有些文件是系统独有的，例如硬件签名、收费注册密钥以及用户名。各机器之间的 Swap 文件也总是有所不同。这也表明了自 RDS 2.19 于 2007 年 12 月发布以来微软发布的更新的数量。

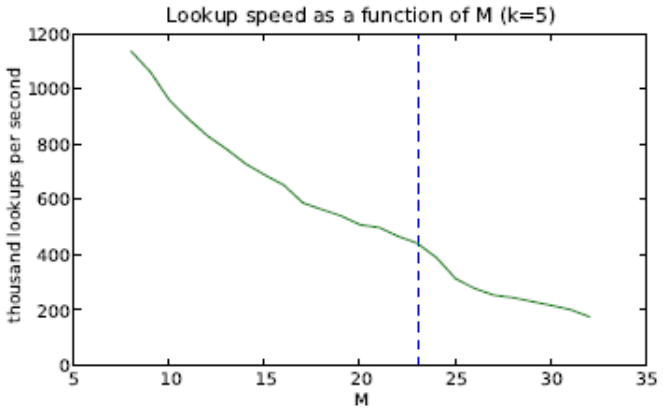


图 2 :由于缓存问题，布隆过滤器的大小增加会造成速度的降低。虚线展示了 2 MB（8 Mbit）的基准系统 L2 缓存（注：这些速度是成功查询的速度；而查询不存在于过滤器的哈希值的速度大约是其 12 倍）。

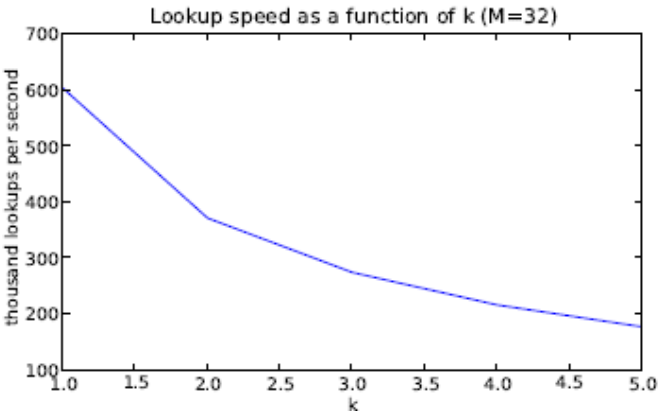


图 3：布隆过滤器每个元素的位数增加都会导致速度的降低，因为查询每个元素会需要更多的工作。

表 3 分析了存在于基本 Microsoft Windows XP 但不存在于 RDS 中的文件。其中大多数是软件安装导致的文件，或实际运行系统生成的日志文件。

PNF files in the WINDOWS/inf directory	707
Windows PC Health Offline Cache files	321
VMWare Tools installation files[a]	130
Start Menu links	95
Other Windows System files	77
Miscellaneous system log files	69
Windows wbem autorecover files	53
Other PC Health files	40
Windows System Restore Files	38
Other Documents and Settings files	41
Windows Prefetch files	31
Miscellaneous system text files	15
File system metadata files	8
Other system shortcuts	7

[a]Artifact of VMWare; not part of the Windows XP base release.

表 3：Windows XP 基本安装中的 1635 个文件不存在于 NSRL RDS 中

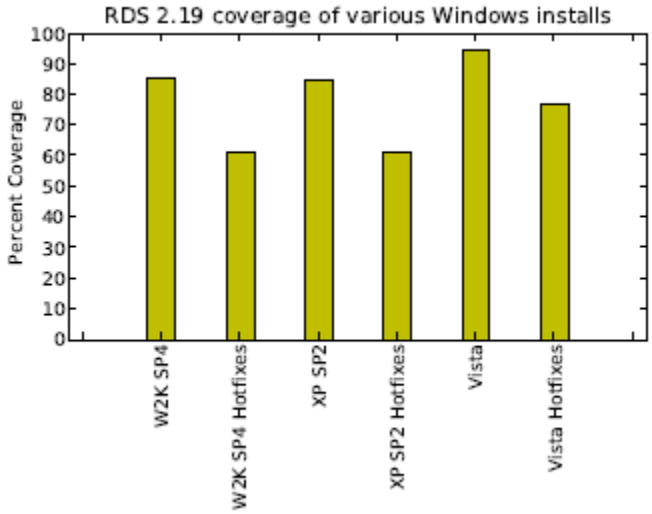


图 4：RDS 2.19 覆盖 Windows 系统

总体而言，不存在于 RSD 中的文件的数量令人担忧：虽然很多不需检查的文件被删除了，但是大量的文件仍然存在。如果 RDS 的主要目的是消除已知的良好文件使其无需进行检查，那么这一目的并没有实现。

3.6 实际数据的覆盖

为了评估 RDS 对现实世界数据的覆盖情况，我们在 891 个硬盘（1998 年至 2006 年之间在二手市场购买）上执行了文件系统 walk。总体来说，45 个硬盘的数据覆盖率达到 80%以上，其中 33 个包含非常多的文件。对于 280 个多于 100 个文件的硬盘，RDS 的平均覆盖率是 36.64%。对于 186 个多于 5000 个文件的硬盘，RDS 的平均覆盖率是 36.62%。再次说明，RDS 有助于减少需要检查的文件，但是效果并不是很好。

3.7 剖析硬盘

虽然如今 RDS 主要用于消除“已知的良好文件”使其免于分析，但我们认为 RDS 的覆盖率限制了这种应用。另一种利用这一资源的方法是使用 RDS 元数据来分析硬盘的可能使用。

我们创建了一个应用程序。通过在 RDS 数据库中查询每个文件的 SHA1、匹配哈希值来检索所有 RDS 对象的列表，以及检索 NIST 确定的产品名的列表，该程序试图分析硬盘。每个哈希码可能多次出现于 RDS，每次出现对应于不同的产品。如果只有一个产品名匹配，则该产品名就被添加到列表中。

以这种方式使用 RDS 使我们能够迅速了解硬盘上安装了什么软件，即使文件删除导致了程序本身无法恢复。

4. 针对布隆过滤器的攻击

将哈希集合作为 BF 发布的显著问题是：攻击者能够很容易地构建与 BF 碰撞的哈希值，这比构建与防碰撞函数（如 SHA-1）碰撞的哈希值容易得多。既然很容易发现碰撞，攻击者就可以利用这个方法来隐藏取证工具（使用 BF 来消除“已知的良好文件”）的错误数据。

假设 SHA-1 是一个强大的哈希函数，发现哈希碰撞的唯一方法是暴力破解攻击，其成功几率是 1400 万除以 2^{160}（假设 RDS 中存在 1400 万个独特的哈希值）。那么，使用哈希碰撞来隐藏错误数据就需要向其添加一个数据块，然后对数据块做微小的改动，直到发现哈希碰撞。在实践中，诸如此类的暴力破解方法永远无法发现碰撞。

然而，如果 160 位被分成 5 组，每组 32 位，并存储在 $m = 2^{32}$ 和 $k = 5$ 的布隆过滤器中，则查找碰撞就会容易得多。1400 万个哈希码代表最多 $14 \times 5 = 7000$ 万个独特的 32 位代码，这些代码全都存储于同一个过滤器中。在 $k = 5$ 的情况下，发现误报要求 5 个 32 位代码组的每一个都要有一个哈希值被设置在过滤器中。每个 32 位代码组的误报概率是 7000 万除以 2^{32} 或 $P = 0.016$。所有 5 个组都出现误报的概率是 $p = (0.016)^5$，或约为 2^{-30}，这使得找到碰撞的难度大致等于破解一个 30 位的加密密钥的难度。

图 5：创建布隆过滤器误报的测试图像

我们使用一个猫咪 JPEG 图像来验证该假设（图 5），附加一个二进制计数器，计算 SHA-1 并检查误报。如果未发现误报，我们就增加计数器的字节并重复该操作。我们发现，当 110,223,107 次迭代增加十六进制字节 03 df 91 06 后，哈希值与 M = 32，k = 4 的布隆过滤器发生了碰撞。找到别名的总计算时间大约是 5.5 CPU 小时。详细信息请参见表 4。

针对这种攻击的唯一防御办法是使用加密的布隆过滤器，例如，用一个 160 位的随机密钥对每个 160 位 SHA-1 进行加密。由于对手不知道密钥，她就无法构建别名。不幸的是，密钥必须严格保密，且 BF 不能在无密钥的情况下使用。

在实践中，这意味着，虽然 BF 是在组织内部分布哈希集合的有用工具，但是将其公布于公开论坛会使其不再适用（因为对手可能会通过创建误报来隐藏数据）。

置换攻击也能有效地防止使用 BF 寻找“已知的坏文件”或跨磁盘分析，这种攻击也能够应对传统的哈希分析。也就是说，黑客工具或错误内容的微小改动都会改变文件的哈希值。因此，与传统的哈希工具相比，使用 BF 查找“已知的坏文件”并不会引入更多的漏洞，但确实会使搜索更加快速。

5. 影响

本章探讨了我们基于本文介绍的 BF 开发的各种取证应用程序。

5.1 关注列表

BF（布隆过滤器）可以用来存储任何类型的哈希值。特别是，它们可以存储从文件或批量数据中提取的特征的哈希值[10]。

我们正在开发一个批量提取应用程序，它能够提取特征、使用可选密钥计算 HMAC（哈希运算消息认证码的密钥），并将结果存储在 BF 中。这个程序可以应用于电子邮件地址列表、信用卡号码，或者其他种类的伪独特信息，以创建关注列表过滤器。这些过滤器可以现场使用来诊断可疑的硬盘，同时将过滤器中的特征遭到破坏的风险降到最低。

5.2 跨磁盘分析

布尔运算可以直接应用到 BF 中。这使得 BF 可以用于跨磁盘分段。该过程很简单。首先，为数据库中的每个磁盘创建包含提取特征哈希值的 BF。设置过滤器需要考虑的最大磁盘数量的阈值（阈值为 n/3，其中 n 是数据库中的硬盘的数量）。接着，设置整数的阈值向量$\bar{F}$，向量的大小等于在 BF 的位数。每个 BF 都被扫描；每遇到一个位 i，向量$\bar{F}$中的相应指数都会递增。在第一遍结束时，向量$\bar{F}$中所有大于阈值的整数都被设置为零。剩余的整数作为相关伪独特特征的过滤器。通过计算$\bar{F} + F_i + F_j$，可以为每个（i，j）磁盘对创建向量$\bar{F}_{i,j}$。

5.3 分割 RDS

文件哈希为取证调查人员提供了一种消除文件集合中不可能修改的已知文件的有用方法。然而，文件集合可以提供更多的有用信息，不过却很难找到。

我们不是将单一的 BF 用于所有的 RDS，而是将数据集合分割成更小的集合：一个集合包含 Windows 安装文件，一个包含普通桌面应用程序的文件，一个包含视频应用程序的文件，等等。

将 RDS 分为两半并将每一半存储在各自的 BF 中，这样可以减少混叠的机会，从而显著降低误报率。虽然用于搜索 BF 的时间增加（因为每个 BF 需要按顺序搜索），但是该方法也有优点：BF 可用于表征文件，而非简单地分为已知和未知。这与使用 Bloomier 过滤器的效果相同[5]。

Original SHA1:	df7ce34d	f723ae1a	675cd06f	e202d060	bd82bd9b
SHA1 of modified file:	cb6b989b	97ad04fb	eaa0ef99	4b8a4059	f2d51dc6
SHA1 of “Index.htm” from “WIN”	C076275E	694CC871	C9624246	CB6B989B	BFFEA55F
SHA1 of “MEMLABEL.PCT” from “Mac”	B961495D	3CDCC1C6	97AD04FB	4CF2CFFE	1BDA3025
SHA1 of “1TXT047.gif” from “WIN2000”	EAA0EF99	B62CD185	3A9DD81A	1FF458C3	734767DF
SHA1 of “H8499.GIF” from “Gen”	4B8A4059	19B1E394	85B7B439	E3A0B940	AD65865F

表 4：对布隆过滤器（M = 32，k = 4）的暴力破解攻击的结果。第一行显示了原始 JPEG（图 5）的 SHA1；第二行显示了被附加十六进制字节 03 df 91 06 的文件的 SHA1；其余各行显示了 RDS 2.19 内部导致误报的特定文件和应用的 SHA1，其别名用方框标出。

5.4 用布隆过滤器进行预过滤

将 RDS 分割为多个 BF 的另一种方法是将 BF 与返回其他信息的较慢的查询服务结合起来。也就是说，不把 RDS BF 视为将 RDS 存储于 MySQL 数据库的替代方法，而是将 BF 用做数据库的加速器：可以首先检查要查询的哈希值是否存在于 BF，如果哈希值存在于 RDS，则可以从 MySQL 数据库获取额外的元数据。

为此，我们的 MySQL 架构为每个哈希存储了更多信息：我们存储了 RDS 分布的所有信息，包括文件名、大小、操作系统 ID、应用 ID、语言，以及 RDS 发布。这些信息被存储在多对 1 和多对多 SQL 表中。可以通过 MySQL 连接器或基于 Web 的 XMLRPC 服务器直接访问这些信息。

6. 结论

通过验证以前研究，我们发现，BF（布隆过滤器）是执行高速哈希匹配的有用工具。但是，我们也发现，BF 并不太适合分布“已知的良好文件”的哈希集合，这是因为对手能够访问过滤器或其参数，而且能够相对容易地修改敌对内容以导致误报。防御这种攻击的方法是在现场用随机选择的加密密钥创建布隆过滤器。

通过用各种不同的 BF 参数测试 RDS，我们发现，有 2^{32} 个 1 位元素（大小为 512 MB）和 5 个哈希函数的过滤器具备很好的性能。我们在公共域中提供了免费的 BF 实现下载；任何人都可以出于任何目的地免费使用或修改。最后，我们指出 BF 如何构建安全关注列表并进行跨磁盘分析。

6.1 可用性

本文中讨论的所有程序都以源代码的方式分布而且用 GNU 编译工具创建。我们已经用 MacOS10.5、Linux 和 FreeBSD 进行了测试。该代码可以从我们的网站服务器下载：http://www.afflib.org/。

6.2 未来研究

我们正在评估将 BF 用于各个文件块或磁盘扇区的哈希集合。

我们正在修改基于 Web 的哈希查询服务，以便社区成员能够自动地提交新的哈希；我们希望实现一个信誉系统和投票算法，以防止数据库中毒。我们可能会进一步修改系统，使用户能够“实时”下载 BF 以便基于 RDS 匹配特定的 SQL 查询（例如，BF 匹配是特定应用程序的一部分）。

我们正在研究 BF 代码的新版本，这一版本将能够直接打开经 ZIP 或 GZIP 算法压缩的 BF。由于 Java 有内存映射文件，而且不断有报告称编写完善的 Java 代码比 C 代码的性能好，我们也在编写 BF 代码兼容的 Java 实现。

6.3 致谢

我们希望向 Brian Carrier、Jesse D. Kornblum、Beth Rosenberg 以及 Vassil Rouss 表达诚挚的感谢，他们对本文的研究提供了非常有帮助的反馈意见。我们还要感谢指出并改正本文之前版本中错误，并建议我们探索误报攻击可能性的匿名审稿人。

本研究获得了美国海军研究生院的研究启动计划的支持。本文中所表达的观点属于作者本人，不反映美国国防部、美国国家标准与技术研究所，或美国政府的官方政策或立场。

参考文献

[1] Fabrice Bellard. Qemu: Open source processor emulator, 2008. http://bellard.org/qemu.

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422–426, 1970. ISSN 0001-0782.

[3] Andrei Broder and Michael Mitzenmacher. Network appli- cations of bloom filters: A survey. Internet Mathematics, 1 (4):485–509, May 2004.

[4] Brian Carrier. The Sleuth Kit & Autopsy: Forensics tools for Linux and other Unixes, 2005. http://www.sleuthkit.org/.

[5] Bernard Chazelle, Joe Kilian, and Ronitt Rubinfeld. The blomier filter: an efficient data structure for static sup- port lookup tables. In Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 30– 39, 2004.

[6] Franck Delattre and Marc Prieur. Intel core 2 duo – test. July 4 2006. http://www.behardware.com/ articles/623-6/intel-core-2-duo-test. html.

[7] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. Commun. ACM, 15(3):191–198, 1972.ISSN 0001-0782.

[8] Peter C. Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In Formal Methods in Computer- Aided Design. Springer-Verlag, 2004. http://www.cc. gatech.edu/fac/Pete.Manolios/research/ bloom-filters-verification.html.

[9] Li Fan, Pei Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking, 8:281–293, June 2000.

[10] Simson Garfinkel. Forensic feature extraction and cross- drive analysis. In Proceedings of the 6th Annual Digi- tal Forensic Research Workshop (DFRWS). Lafayette, Indi- ana, August 2006. http://www.dfrws.org/2006/ proceedings/10-Garfinkel.pdf.

[11] Guidance Software, Inc. EnCase Forensic, 2007. http://www.guidancesoftware.com/ products/ef_index.asp.

[12] William Henning. Intel core 2 duo e6600 review. Neoseeker, September 19 2006. http://www.neoseeker.com/Articles/Hardware/ Reviews/core2duo_e6600/6.html.

[13] Ken Jacobs and Keikki Tuuri. Innodb: Architecture, features, and latest enhancements. In MySQL Users Conference 2006, 2006.http://www.innodb. com/wp/wp-content/uploads/2007/04/ innodb-overview-mysql-uc-2006-pdf.pdf.

[14] Panagiotis Manolios. Bloom filter calculator, 2004. http://www.cc.gatech.edu/~manolios/ bloom-filters/calculator.html.

[15] Michael Mitzenmacher. Compressed bloom filters. pages 144–150, 2001.

[16] James K. Mullin. A second look at bloom filters. Commun. ACM, 26(8):570–571, 1983. ISSN 0001-0782.

[17] Vassil Roussev, Yixin Chen, Timothy Bourg, and Golden G. Richard III. md5bloom: Forensic filesystem hashing re- visited. Digital Investigation, 3(Supplement-1):82–90, 2006.

[18] Vassil Roussev, Golden G. Richard III, and Lodovico Marziale. Multi-resolution similarity hashing. Digital In- vestigation, 4(Supplement-1):105–113, 2007.

[19] Douglas White. NIST national software reference library (NSRL), September 2005. http://www.nsrl.nist. gov/documents/htcia050928.pdf.

[20] Douglas White, August 17 2006. http://www.nsrl. nist.gov/RDS/rds_2.13/bloom/.