Progger: An Efficient, Tamper-Evident Kernel-Space Logger for Cloud Data Provenance Tracking

Ryan K. L. Ko
Cyber Security Lab
The University of Waikato
Hamilton, New Zealand
Email: ryan@waikato.ac.nz

Mark A. Will
Cyber Security Lab
The University of Waikato
Hamilton, New Zealand
Email: maw41@waikato.ac.nz

***Abstract*—Cloud data provenance, or "what has happened to my data in the cloud", is a critical data security component which addresses pressing data accountability and data governance issues in cloud computing systems. In this paper, we present Progger (Provenance Logger), a kernel-space logger which potentially empowers all cloud stakeholders to trace their data. Logging from the kernel space empowers security analysts to collect provenance from the lowest possible atomic data actions, and enables several higher-level tools to be built for effective end-to-end tracking of data provenance. Within the last few years, there has been an increasing number of proposed kernel space provenance tools but they faced several critical data security and integrity problems. Some of these prior tools' limitations include (1) the inability to provide log tamper-evidence and prevention of fake/ manual entries, (2) accurate and granular timestamp synchronisation across several machines, (3) log space requirements and growth, and (4) the efficient logging of root usage of the system. Progger has resolved *all* these critical issues, and as such, provides high assurance of data security and data activity audit. With this in mind, the paper will discuss these elements of high-assurance cloud data provenance, describe the design of Progger and its efficiency, and present compelling results which paves the way for Progger being a foundation tool used for data activity tracking across all cloud systems.**

***Index Terms*—Cloud Computing; Data Provenance; Data Security; Accountability; Tamper-evident logging; Time Synchronisation.**

I. Introduction

Data is arguably the most important asset in cloud computing [1], [2]. This holds especially true because the nature of cloud computing's business model requires us to rely on a third party to process, store or manage our data. Such a reliance also involves a reliance on a trust relationship. This trust relationship can be enhanced by an increased level of confidence. One clear obstacle to a high-level of confidence in a third-party cloud computing service provider is the inability for cloud users to know "what has happened to my data in your cloud?" [3], [4], [5]. This question in itself, embodies the gist of the problem this paper is attempting to solve – cloud data provenance. We define cloud data provenance as the meta-data describing the derivation history of data in a cloud computing environment. We argue that provenance information should not only be collected from the most atomic data activities in the kernel, but also maintain a high level of log data integrity, time accuracy and efficient space requirements. In Section II, we will review related work and paradigms and discuss why their gaps present a strong case for the development of Progger.

II. Related Work

A. Missing the Point – Traditional System-Centric Tools

Typically, there are system-centric logs and there are data-centric logs [3]. Many existing cloud systems are deploying out-of-date, system-centric loggers which log events accessing or utilising systems, and were useful for a 'perimeter defence' security mindset. However, they do not treat data as a first-class citizen and are unable to track each datum's life cycle (which is relevant to today's needs). Clearly a rethink is needed in terms of logging the evolution of data in systems such as clouds. Current system-centric logs are insufficient to address the end-to-end tracking needs of data across several systems within and outside of a cloud [6] due to the following reasons.

Firstly, current monitoring tools [7], [8], [9] (e.g. HyTrust [10]) are only monitoring utilisation and performance, overlooking the flow of data in the cloud. As such, users concerns such as "where is my data in your cloud?" or "who has touched my data?" are not addressed directly [1]. Secondly, while most clouds adopt file-integrity checking systems (FICS) (e.g. TripWire [11]) to detect file intrusions, a serious loophole exists. If a file was accessed n times, only the n^{th} change is reflected in FICSs while the provenance of the 1^{st} to $(n\text{-}1)^{th}$ file intrusions are missed out.

B. Critical Gaps Identified in Related Provenance Techniques

To our best knowledge, Hewlett-Packard TrustCloud's [5] Flogger [12] was the first data-centric logger for cloud data provenance tracking. Flogger was eventually integrated with HP ArcSight's SIEM (Security Information and Event Management) tools to detect data changes. While it was a breakthrough and much more effective way of logging provenance compared to file-system dependent PASS [13] or LASAGNE [14] and earlier data provenance works prior to the notion of cloud computing [15], [16], [17], [18], [19], some limitations exist. Improvements to the provenance concepts Flogger introduced were made. For example, DataPROVE [20] was proposed to cater to the causality issue identified and S2Logger [21] was proposed to solve the scalability and visualisation challenges in end-to-end data activity tracking

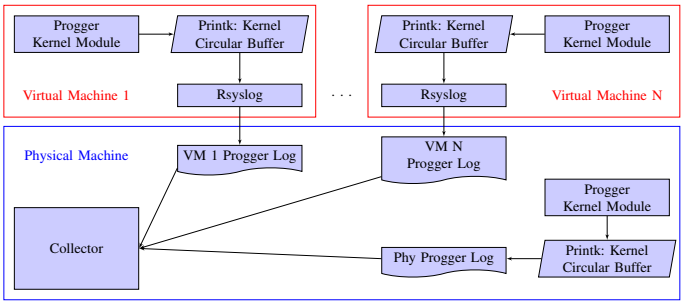

Fig. 1. Progger deployment in the Cloud.

in the cloud. However, all of them still were unable to maintain the authenticity and integrity of provenance logs (e.g. ensuring tamper-evident features, restricting the chance of fake/ malicious entries to a very minute percentage). They were also unable to ensure that timestamps for provenance logs collected from several different machines still observe a causal integrity when the logs are joined together in a database. For example, the above techniques, by themselves, are unable to assure the integrity of causal sequences such as *"event A (on machine 1) → event B (machine 2) → event C (machine 1)"*. It is easy for a microsecond difference in the timestamp to disrupt the integrity of the order of the log entries when the logs from several machines are combined together. Lastly, they did not ensure that the logs are recorded with minimal clutter and wastage of storage sizes.

III. Progger - Design Overview & Key Breakthroughs

These gaps established a strong need to release a brand-new tool, Progger (Provenance Logger), which inherently addressed these gaps – setting the stage for high assurance data accountability in clouds and a proliferation of tamper-evident, time-accurate and space efficient data provenance loggers and analysis tools.

A. Basic Architecture

While Progger is built as a Linux kernel module which is inserted into every machine within the cloud infrastructure, both virtual and physical, its concepts can also be easily adopted into other operating systems such as the Unix or Windows environment.

For a VM, the Progger log file can be stored on the virtual disk, or stored on the physical host using a virtual serial device as shown in Figure 1. Because of the rate of growth of the log file, storing it on the VM is not ideal. Storing on the physical machine allows for easier collection and size trimming.

When loaded, the module modifies the addresses of certain system calls in the system call table allowing Progger to create logs when these calls are accessed. The base address of the system call table varies with each different kernel version, but it can be found by searching */boot/System.map* for the system call table entry. Progger finds this address dynamically on insertion. By default the kernel does not allow the system call table to be modified, so page protection needs to be disabled before Progger saves the original address and adds the custom system calls; after which, page protection is enabled again. Temporally disabling page protection can be seen as a security risk, even though it is for a very short period of time. To reduce this risk, local interrupts can be disabled while the page protection is disabled, which minimises the chance of another program being able to modify the table in the very short time period.

In order for Progger to create logs, a technique was required for the kernel module to write log data to a file. The simplest would be to open the file directory from within the kernel, however this is not recommended practice [22]. There are file systems (e.g. /proc/) which allow for direct kernel access, but they are designed for small configurations or statistics, and not for log files. Also for VMs, this would mean the log would have to be on the virtual disk and not piped through to the physical machine.

A common way of dealing with this problem is to have the kernel pass the information to a user-space process which handles the log file access. There are many ways of achieving this, most of which involves having the kernel copy data to a buffer which can be then be accessed by a user-space process. The technique chosen for Progger was *printk* because it is robust, i.e. the kernel doesn't directly perform file access, and is a default logging mechanism for the kernel [23]. It also performs well with large outputs because it is copying memory to the kernel's built-in circular log buffer and maintains relatively constant performance (this is futher discussed in the Section IV). This is an important attribute for Progger because it records all data being written and read.

When Progger uses *printk*, a low priority is used so they are not shown on screen, and with the log entries by default going into the standard log file for the system. To allow Progger to have its own log file, *rsyslog* was used because it allows kernel log messages to be filtered into files, and many Linux distros use *rsyslog* as their default kernel logger daemon. *Rsyslog* also handles the timestamp insertion for each entry into the Progger log file which keeps the events ordered by time. Other information can also be added to Progger logs by using *rsyslog* to redirect certain events, such as IP address changes.

B. Addressed the Time Synchronisation Issue

Within a cloud environment, tracking an event across multiple machines relies on the log files to contain accurate timestamps. This is so that they can be joined together to replicate the actual sequence of events. However keeping the system time on a virtual machine in sync with the rest of the cloud is a challenge. This is because virtual machines share the resources of the physical machine and therefore do not have knowledge of the period where they were not running. Operating systems typically keep time by counting ticks, which are from a hardware interrupt, or via a tickless approach where some hardware device tracks the time from when the system booted. Tickless timekeeping is easier to support within the virtual infrastructure however can still experience drift.

There are a few solutions to this problem, such as reading the time periodically from the physical machine to correct drift. However it becomes important to make sure this system does not 'travel back in time' during the bid to correct drift. Often, the Virtual Infrastructure has mechanisms in place to try and keep VMs in sync with the physical machine, e.g. VMware's tools provide a synchronization feature [24]. If this is not reliable enough, the other main solution is to use the Network Time Protocol (NTP). For a large virtual infrastructure, a high performance NTP server would be required to be able to serve every machine, especially if the VMs are requesting time frequently to reduce the amount of drift experienced.

This issue will differ with each cloud environment and the severity will vary depending on each day's load. Therefore, for Progger to handle time drift by itself would be incredibly difficult. However it does provide a means for joining logs together and correctly without disrupting time sequences within the joined log file. This is possible because Progger runs on every machine, meaning if an event occurs between two machines in the cloud infrastructure, the machines time may differ by minutes. But because Progger logs other events such as the creation of sockets and the acceptance of connections, analysing these entries in the two machines log files allows the time difference to be known and eventually corrected, during the joining of VM log files to a physical machines. The physical machines' log times should be accurate so that they can be used to correct time as well. It is important to note that it is recommended that the software processing the Progger logs handles this, not Progger.

C. Reducing Clutter and Log Sizes

The currently implemented system calls and their basic definitions can be found in Figure 2 along with their output log formats. The size of each log entry has been kept to a minimum and further optimisations could be made to make them even smaller if desired. For example, once the information about a process has been logged, it would be possible to just log the data that is specific to the type of call. Progger has kept the logs simplistic for easier initial analysis, while maintaining a relatively smaller log size.

D. Preventing Log Tampering

Preventing log tampering is another important feature that needs to be implemented for a security logger such as Progger. This is a very complex and difficult requirement. By default, access to the log file requires root privileges. However, this does not stop a user with sudo access from tampering with the log. Progger stops log tampering by only allowing the *rsyslog* process access to the file within the system calls themselves. Therefore, if a user tries to open the log file, Progger will return a permission error even if they actually do have permissions. For Progger to know if *rsyslog* is trying to access the log file, there are two methods which can be used: (1) pass the process ID of *rsyslog* to Progger on insertion, or (2) check the process name. Passing the process ID is more secure because another process can be also named *'rsyslog'*. Because the log file on a VM is actually a pipe to the physical machine, the log can still be read by some analyser or collector process. For the physical machine's log file, the analyser/collector process ID would also have to be passed to Progger on insertion. It is still possible for a malicious user to kill this collector process and spawn a process with the same process ID as the collector had. However, this is extremely unlikely. If Progger wanted to protect against this attack, it would have to prevent the termination of the collect process ID, for example by intercepting the kill system call. However, we acknowledge that adding fake entries into the Progger log is still possible with this method by a user (with superuser access) creating and inserting another kernel module. This *other* module can call the *printk* function with the Progger tag at the start of the output, eventually leading to it being used by rsyslog to write to the Progger log file instead of a generic one. Currently Progger cannot prevent this due to performance or implementation issues. For example one solution would be to use the write system call to check that the data going into the Progger file came from Progger itself, however this would dramatically impact performance. It is also important to realise that such an ability to insert a kernel module would have also shown that the root of the machine has already been compromised, i.e. leaving the owner of the system with not much further options.

Another solution would be to sign the entries but this would make Progger harder to setup and maintain. This translates to a need to manage all the signatures for all entries on every machine. Alternatively, log entry values can be used to create a hash, thus 'signing' the entry. Unfortunately this would not stop entries being added because Progger is open source, making it easy to find out how the hash is calculated. There is, however, a way around this problem, instead of outputting the current entries signature, output the previous entries signature, or create a signature using a hash chaining approach, where the current signature contains a cryptographic digest of the previous log entry [25], [26].

To make both techniques more secure, instead of using the previous entries' signature, we use the previous signature of the last entry with the same operation, e.g. an *'open'* system call entry would include the signature of the last open entry. This technique makes it very hard to change or add a log entry because to get the information needed to generate the signature of the previous entry, the Progger log file needs to be read, and access to it is blocked. Even if the malicious user knew the correct signature, say from accessing memory directly, when Progger outputs more log entries, a log entry will have the same signature as the malicious entry as shown in Figure 3, allowing the collector/analyser to detect tampering. If only one entry was added, it would be easy to tell which was the malicious entry, therefore the malicious user would have to be either constantly adding entries to the log so that it's not possible to tell which branch is fake, or editing memory directly to correct the signature of the next log entry that has the same operation type. Even then, if hash chaining was used, the malicious user would have to be constantly

System Call	Description	Log Format
__NR_open	Opening and creating files.	Type,User,PID,PPID,SID,PSID,Program,File,WD,Flags,Mode,FD
__NR_close	Closing a file descriptor.	Type,User,PID,PPID,SID,PSID,FD
__NR_rename	Rename a file.	Type,User,PID,PPID,SID,PSID,Program,OldFile,NewFile,Path
__NR_unlink	Remove a file.	Type,User,PID,PPID,SID,PSID,Program,File,Path)
__NR_unlinkat	Remove a file.	Type,User,PID,PPID,SID,PSID,Program,File,Path,DirFD,Flags
__NR_read	Reading data from a file descriptor.	Type,User,PID,PPID,SID,PSID,FD,Offset,HexData
__NR_write	Writing data to a file descriptor.	Type,User,PID,PPID,SID,PSID,FD,Offset,HexData
__NR_pwrite64	Writing data to a file descriptor with an offset.	Type,User,PID,PPID,SID,PSID,FD,Offset,HexData
__NR_dup	Duplicate a file descriptor.	Type,User,PID,PPID,SID,PSID,FD,FD
__NR_dup2	Duplicate a file descriptor.	Type,User,PID,PPID,SID,PSID,FD,FD
__NR_mkdir	Create a directory.	Type,User,PID,PPID,SID,PSID,Name,Path,Mode
__NR_rmdir	Remove a directory.	Type,User,PID,PPID,SID,PSID,Name,Path
__NR_symlink	Create a symbolic link.	Type,User,PID,PPID,SID,PSID,Path1,Path2,WD
__NR_link	Link a file to an existing file.	Type,User,PID,PPID,SID,PSID,Path1,Path2,WD
__NR_linkat	Link a file to an existing file.	Type,User,PID,PPID,SID,PSID,Path1,Path2,Dir1,Dir2,Flags
__NR_chown	Change file owner and group.	Type,User,PID,PPID,SID,PSID,Program,File,Owner,Group
__NR_fchown	Change file owner and group.	Type,User,PID,PPID,SID,PSID,Program,FD,Owner,Group
__NR_lchown	Change file owner and group.	Type,User,PID,PPID,SID,PSID,Program,File,Owner,Group
__NR_fchownat	Change file owner and group.	Type,User,PID,PPID,SID,PSID,Program,File,DirFD,Owner,Group,Flags)
__NR_chmod	Change file permissions.	Type,User,PID,PPID,SID,PSID,Program,File,Mode
__NR_fchmod	Change file permissions.	Type,User,PID,PPID,SID,PSID,Program,File,Mode
__NR_fchmodat	Change file permissions.	Type,User,PID,PPID,SID,PSID,Program,File,DirFD,Mode,Flags
__NR_sendfile	Transfer data between two file descriptors.	Type,User,PID,PPID,SID,PSID,OutFD,InFD,Offset,Count
__NR_socket	Create a socket.	Type,User,PID,PPID,SID,PSID,Program,SockFD,sType,sProtocol,sFamily
__NR_connect	Connect to a remote host.	Type,User,PID,PPID,SID,PSID,SockFD,IP,Port
__NR_accept	New socket from incoming connection.	Type,User,PID,PPID,SID,PSID,SockFD,IP,Port
__NR_sendto	Transmit a message on a socket.	Type,User,PID,PPID,SID,PSID,SockFD,Flags,Len,DestIP,HexData
__NR_recvfrom	Receive a message from a socket.	Type,User,PID,PPID,SID,PSID,SockFD,Flags,Len,DestIP,HexData
__NR_sendmsg	Transmit a message on a socket.	Type,User,PID,PPID,SID,PSID,SockFD,Flags,Len,HexData
__NR_recvmsg	Receive a message from a socket.	Type,User,PID,PPID,SID,PSID,SockFD,Flags,Len,HexData
__NR_pipe	Data channel between two processes.	Type,User,PID,PPID,SID,PSID,FDRead,FDWrite,Flags
__NR_pipe2	Data channel between two processes.	Type,User,PID,PPID,SID,PSID,FDRead,FDWrite,Flags

Fig. 2. System Calls implemented by Progger.

fixing the signatures before *rsyslog* outputted them, or edit the Progger processes memory to change the variable were the previous entries signature is stored to make Progger output the malicious signature branch.

Finally once root is compromised, it is very difficult to 100% guarantee that the log has not been altered. The proposed techniques make it very hard to tamper with logs, but with the right access and skill level, it is still possible.

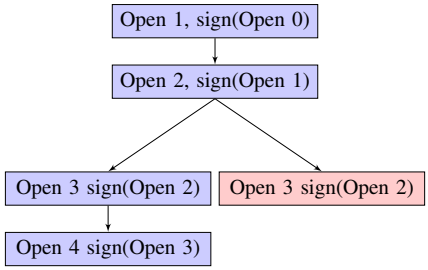

Fig. 3. Example of a malicious log entry with signing of the last entry.

E. Logging Root Usage

Logging the data actions and actual identities of the users running the *sudo* command is very important for keeping track of all users data. When a logged system call is called via a sudo, it is usually recognised as root, not the real user. This is a problem when tracking access to files and data, because when analysing the logs we need to know who was running with root privileges.

There are a few solutions to solving this problem. One would be a custom version of the sudo program which would be able to log the real user and information on the process being executed, and using the default Linux authentication log. For example. Red Hat based systems' */var/log/secure* contains the authentication logs and tracks the user and command that sudo executed.

Progger takes a different approach by logging when the user initially runs the sudo program, because sudo will open supporting files like */etc/passwd*. This allows Progger to log if a user runs the sudo command. However it then needs to know what operations where performed when the real user was elevated to root privileges. Progger accomplishes this by logging the current tasks session, which is unique for each user login and will remain the same for the duration of the session. This means that if multiple users were running sudo commands at the same time, the Progger logs still show which root action belongs to which user. The Session ID also allows Progger to keep track of a user if they are logged into the system multiple times. For example if they are connected to a machine twice using ssh, Progger can differentiate between the two connections.

IV. Performance

(All performance results were recorded on a VM with 2 x 2.6GHz Intel Xeon E312xx (Sandy Bridge) cores, 4GB RAM, 8GB harddisk with 27kB read/s and 4kB write/s.)

If any generic logger logged all the operations of the root user, the system may become very slow and unresponsive. Therefore in order for Progger to log all sudo operations performed by a user, it checks the effective user id of the current tasks session to see if it is a real user or root. This allows all actions of the user to be logged, while not logging the systems root actions to maintain performance.

In Progger's logging of data system calls, extra operations are performed every time the system call is used, which will impact the performance of the system. Thus logging code for each system call needs to be kept at a minimum, but should still be able to dump all the required information. However the code collecting the data is not the main overhead, it is the actual process of dumping the data from the kernel. Because

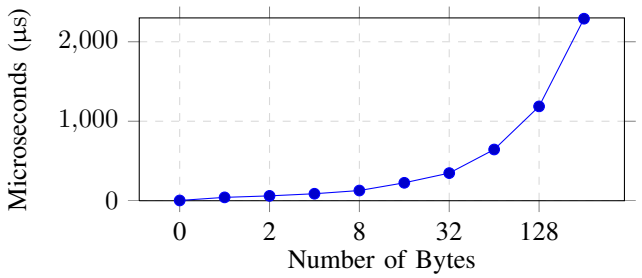

Fig. 4. *printk* performance for varying byte sizes.

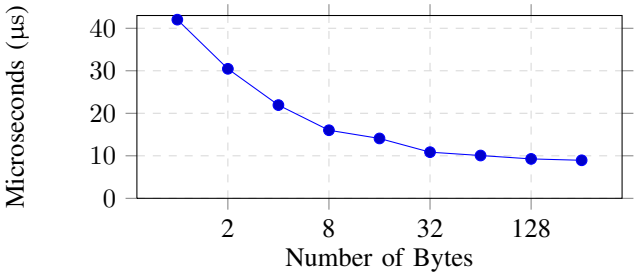

Fig. 5. *printk* cost per byte for varying byte sizes.

Progger uses *printk* to output logs, the data is copied to the kernels circular log buffer. It is this copy which impacts the performance of the system the most. Figure 4 shows how *printk* scales with varing byte sizes in terms of time. The test system call with *printk* printing 0 bytes, takes 3.21 µs, were printing 1 byte completes in 42.025 µs on average, which is over 10 times slower. However 2 bytes takes 60.926 µs, which shows there is a setup cost of using *printk* because adding 1 byte only increased the time by 18.901 µs. Therefore adding more bytes does increase the time, however the cost of each byte decreases slightly as shown in Figure 5.

The performance of the *open, close, read and write* system calls will now be evaluated individually, with the results shown in Figure 6. The 'Default' column shows the time taken running the test program without Progger logging that call, and the 'Progger' column is where the call is logged. The Progger log file for these performance tests was located on the VM's disk.

The same test program was used for both the *open* and *close* system call, where a file was opened then closed again in a large loop to allow for the time for one call to be calculated more accurately. These results show that because the log format for *open* is more verbose than *close*, the *printk* operation takes longer, i.e. the call itself takes longer. For the *read* system call test, a file was opened, 15 bytes were read, then it was closed and repeated in a loop. With Progger only logging the *read* system call, the time was still consistent with the *printk* results. The test for *write* was the same as *read* except it wrote 15 bytes to the file, and the results by default were far larger than the other three calls tested. Subsequently, when Progger was logging the *write* system call, the time was dramatically higher even though it should be similar to *read*. This comes down to the performance of the VM, as the disk I/O has clearly effected the results with the limited write speed, with both rsyslog and the test program writing to disk at once. Also with the *read* test, because of the large loop, it is possible that it getting a lot of disks cache hits, which would have improved the timings. The results described in this section clearly show that the performance of Progger is primarily that of *printk* and can be heavily impacted by the performance of the actual machine, like the disk. With the performance of the *write* system call, it is clear that having the log file not stored on the primary disk, or having it piped through to the physical host is very important.

System Call	Default	Progger
Open	3.54 µs	758.7 µs
Close	3.54 µs	401.5 µs
Read	6.118 µs	673.8 µs
Write	299.06 µs	2072.08 µs

Fig. 6. Four common system calls' performance.

V. Example Scenarios

Note that the actual log entry types are numerical values (refer to Figure 7), but have been replaced with a description for readability. Many unrelated log entries have also been removed so only the key entries for files are shown in this paper to make them easier to follow. Finally, time values have been removed to make the logs concise. The following is a non-exhaustive list of the capabilities of Progger.

A. Detecting a File Creation

Alice first creates an empty file, then edits this file using the Vim text editor to add one line. After which, she reopens this file and adds another line before saving and closing the file again. Only actions related to the file are shown, other actions by Vim for libraries and swap files have been removed.

Command: *alice@host$ touch file1.txt*
Log:

1	`Create,alice,`**`11140`**`,11002,11002,11001,touch,file1.txt,/home/alice/,2369,438,3`
2	`Duplicate,alice,`**`11140`**`,11002,11002,11001,3,0`
3	`Close,alice,`**`11140`**`,11002,11002,11001,3`
4	`Close,alice,`**`11140`**`,11002,11002,11001,0`

Process 11140 creates the file "file1.txt" in the directory "/home/alice/", and returns a file descriptor with the value 3. This file descriptor is then duplicated to 0, which is stdin. Finally both file descriptor 3 and 0 are closed. This is a simple scenario, and Progger gives a lot of information on the created file. The file was created by Alice using the touch program in her home directory. The flags and mode are also logged. This log output is interesting because the duplicate to stdin is not needed, which requires extra system calls.

Command: *alice@host$ vim file1.txt* (Enters "one", saves file, then closes)
Log:

1	Move,alice,**11205**,11002,11002,11001,vim, file1.txt,file1.txt~,/home/alice/
2	Create,alice,**11205**,11002,11002,11001,vim, file1.txt,/home/alice/,577,436,**3**
3	Write,alice,**11205**,11002,11002,11001,**3**,0, 6F6E650A
4	Close,alice,**11205**,11002,11002,11001,**3**
5	Chmod,alice,**11205**,11002,11002,11001,vim, file1.txt,33204
6	Remove,alice,**11205**,11002,11002,11001,vim, file1.txt~,/home/alice/

Vim is running with a process id of 11205, and the first operation to the users file is it moves the original file "file1.txt" to a backup "file1.txt~", before recreating "file1.txt" which returns file descriptor 3. Then once Alice types "one" in Vim and writes the change, 0x6F6E650A(one\n) is written to file descriptor 3. Alice then quits the file resulting in the file descriptor 3 being closed. Finally Vim fixes the permissions of the newly created file and removes the backup. Every time Vim writes to the file, it moves the original file to a backup, creates a new file and rewrites everything, even if only one character is replaced.

Command: *alice@host$ vim file1.txt* (Appends "two", saves then closes)
Log:

1	Open,alice,**11313**,11002,11002,11001,vim, file1.txt,/home/alice/,0,0,**3**
2	Read,alice,**11313**,11002,11002,11001,**3**,0, 6F6E650A
3	Close,alice,**11313**,11002,11002,11001,**3**
	...
4	Move,alice,**11313**,11002,11002,11001,vim, file1.txt,file1.txt~,/home/alice/
5	Create,alice,**11313**,11002,11002,11001,vim, file1.txt,/home/alice/,577,436,**3**
6	Write,alice,**11313**,11002,11002,11001,**3**,0, 6F6E650A74776F0A
7	Close,alice,**11313**,11002,11002,11001,**3**
8	Chmod,alice,**11313**,11002,11002,11001,vim, file1.txt,33204
9	Remove,alice,**11313**,11002,11002,11001,vim, file1.txt~,/home/alice/

This time because "file1.txt" is not empty, Vim as process 11313 reads the data from the file, then closes it. The backup and new file are created like before. When Alice adds the new line and writes the change, Vim writes the whole file again (0x6F6E650A74776F0A = "one\ntwo\n"). This actually happens every time Vim writes to the file, it moves the original file to a backup, creates a new file and rewrites everything, even if only one character is replaced.

B. Detection of a DNS query and Pinging a Server

Command: *bob@host$ ping www.google.co.nz*
DNS Query Log:

1	Socket,bob,11426,11409,11409,11408,ping, **4s**,2050,0,2
2	Connect,bob,11426,11409,11409,11408,**4s**, 55682,13568
3	SendTo,bob,11426,11409,11409,11408,**4s**, 16384,34,0,78C501000001000000000000377777706676F6F676C6502636F026E7A0000010001
4	RecvFrom,bob,11426,11409,11409,11408,**4s**, 0,1024,55682, 78C581800001000300070000C0377777706676F6F ... 00102001067C10100013000000000000053497F
5	Close,bob,11426,11409,11409,11408,**4s**

Ping creates a socket with the descriptor 4. Then it connects to the DNS server 130.217.X.X:53, note the values for the IP address and port need to be byte reversed. The following data is then sent as a DNS query:
78C5 = ID, 0100 = Flags, 0001 = Questions, 0000 = Answer RRs, 0000 = Authority RRs, 0000 = Additional RRs, 0377777706676F6F676C6502636F026E7A00 = Name (www.google.co.nz), 0001 = Type (A), 0001 = Class (In)
The query result is then received from 130.217.X.X, before the socket is closed. Progger can therefore easily track network activity on a machine, which would allow for the detection of certain malware.

Ping Log:

1	Socket,bob,11426,11409,11409,11408,ping,**4s**, 2,0,2
2	Connect,bob,11426,11409,11409,11408,**4s**, -655524534,260
3	Close,bob,11426,11409,11409,11408,**4s**

Ping creates a new socket, connects to 74.125.237.216, then closes the socket.

C. Detecting the Use of 'echo' to Create a File

Another scenario for Progger is where Bob saves his wifi password to a text file in his root directory. Then Alice views this file by using the cat program. By default Alice does not have access to open the file, however she has permissions to run cat as root using the sudo command.

Command: *bob@host$ echo "mywifipassword" >wifi.txt*
Log:

1	Create,bob,10444,10443,10444,10443,bash, wifi.txt,/home/bob/,577,438,**3**
2	Duplicate,bob,10444,10443,10444,10443,**3**,**1**
3	Close,bob,10444,10443,10444,10443,**3**
4	Write,bob,10444,10443,10444,10443,**1**,0, 6D797769666970617373776F72640A

The file "/home/bob/wifi.txt" is created in Bob's home directory with file descriptor 3 returned. The file descriptor is then duplicated in 1 (stdout), before being closed. Therefore instead of echo writing to terminal, the output actually goes into "wifi.txt" (0x6D797769666970617373776F72640A = mywifipassword\n). Stdout was not closed in this test, it was just linked to another file descriptor.

Command: *alice@host$ sudo cat /home/bob/wifi.txt*
Log:

1	Open,alice*,11026,11020,**11002**,11001,cat, /home/bob/wifi.txt,/home/alice/,0, 3396397472,**3**
2	Read,alice*,11026,11020,**11002**,11001,**3**,0, 6D797769666970617373776F72640A
3	Write,alice*,11026,11020,**11002**,11001,**1**,0, 6D797769666970617373776F72640A
4	Close,alice*,11026,11020,**11002**,11001,**3**
	...
5	Open,alice,11101,11002,**11002**,11001,ls,., /home/alice/,67584,1,3

The file "/home/bob/wifi.txt" is opened by root. However because Alice is the effective user of the session, her username is shown with an asterisk to show it was performed as root. The data is then read from the file and written to stdout. Then the file is closed. After a few more log entries, Alice runs the ls command as herself, this shows that her session id (11002) is the same for both the root actions and her own. When the sudo command is run, there are also other entries in the Progger log for the authentication stage however these have been removed for this paper.

D. Detecting a File Received From Outside the Cloud

Bob on a remote machine outside of the cloud copies a file onto a machine in the cloud using SCP.

Command: *scp somefile.txt bob@130.217.X.X:~/file2.txt*
Log:

1	Accept,root,**1490**,1,**1490**,1,**3s**,55682, 2962
2	Open,root,**11684**,**1490**,**1490**,1,sshd, /proc/self/oom_score_adj,/,577,438,10
	...
3	Open,bob,**11688**,**11684**,**11684**,**1490**,sshd, /proc/self/task/2076/attr/exec,/,0,0,6
4	Open,bob,**11688**,**11684**,**11684**,**1490**,sshd, /proc/self/task/2076/attr/current,/,2,0,6
	...
5	Create,bob,**11689**,**11688**,**11689**,**11688**,scp, /home/bob/file2.txt,/home/bob/,65,420,3
6	Read,bob,**11689**,**11688**,**11689**,**11688**,0,0, 6F6E650A74776F0A
7	Write,bob,**11689**,**11688**,**11689**,**11688**,3,0, 6F6E650A74776F0A
	...
8	Close,bob,**11689**,**11688**,**11689**,**11688**,**3s**

The ssh daemon accepts a connection from 130.217.X.X (IP partially masked for security reasons). The name of the daemon is obtained by the process id 1490, and that the second entry logs the program name. Then entries three and four show that a new session has been created (11688) and is running as Bob. Another new session was created (11689) which creates the file "file2.txt" in Bob's home directory. Data is then read from stdin, and written to "file2.txt". After skipping over some entries, the socket is closed.

Linking the session id 11689 to the remote host 130.217.X.X is not straight forward. Many session and process ids need to be known in order to trace it back to the ssh daemon. An important detail to understand is that the ssh daemon will have the same process id for multiple connections, which leads to a problem of which IP address is the session actually linked to, since all sessions trace back to the process 1490 in this case. However when the ssh daemon accepts a connection, it has to create a new process to handle that connection before it can listen for new connections. This means of the ssh daemons child processes which are remote sessions like 11684, the child with the lowest process ID will be the first accept log entry, because it had to be created before any more accepts could occur.

E. Detecting File Transfers to a Machine Outside the Cloud

Similar to the last scenario, Bob is sending the file from within the cloud. If the remote machine is within the cloud also, the machines Progger log would be the same as before. So for this scenario, the remote machine is outside the cloud.

Command: *scp file1.txt bob@130.217.X.X:~/file1.txt*
Log:

1	Pipe,bob,**1635**,1591,1591,1590,5,6,0
2	Pipe,bob,**1635**,1591,1591,1590,7,8,0
3	Close,bob,**1635**,1591,1591,1590,5
4	Close,bob,**1636**,1635,1591,1590,6
5	Close,bob,**1636**,1635,1591,1590,7
6	DUP2,bob,**1636**,1635,1591,1590,5,0
7	DUP2,bob,**1636**,1635,1591,1590,8,1
8	Close,bob,**1636**,1635,1591,1590,5
9	Close,bob,**1636**,1635,1591,1590,8
10	Close,bob,**1635**,1591,1591,1590,8
	...
11	Connect,bob,**1636**,1635,1591,1590,3s,55682, 5632
	...
12	DUP,bob,**1636**,1635,1591,1590,0,4
	...
13	Open,bob,**1635**,1591,1591,1590,scp,file1.txt, /home/bob/,2048,0,3
	...
14	Write,bob,**1635**,1591,1591,1590,6,0, 433036363420382066696C65312E7478740A
15	Read,bob,**1636**,1635,1591,1590,4,0, 433036363420382066696C65312E7478740A
16	Write,bob,**1636**,1635,1591,1590,3s,0, 0F........C8E5
	...
17	Read,bob,**1635**,1591,1591,1590,3,0, 6F6E650A74776F0A
18	Write,bob,**1635**,1591,1591,1590,6,0, 6F6E650A74776F0A
19	Read,bob,**1636**,1635,1591,1590,4,0, 6F6E650A74776F0A
20	Write,bob,**1636**,1635,1591,1590,3s,0, DC........B469

Before SCP creates a new thread for handling the network socket, it creates some pipes for communication between the two processes. The two processes close unneeded descriptors, resulting in one pipe for 1635 to 1636 and the other for 1636 to 1635. The read end of the [5,6] pipe is set to stdin,

and the write end of the [7,8] pipe is set to stdout. Then the unneeded descriptors are closed. Now process connects to 130.217.X.X:22 resulting in the socket descriptor 3 being returned. Next, in the child process stdin gets duplicated to descriptor 4, which is actually the read end of the pipe. The file "file1.txt" is then opened by the parent process before the data "C0664 8 file1.txt\n" gets written to the pipe. This data is then read from the pipe in the child process and written to the socket to be sent to 130.217.X.X. Now the data is read from the file and written to the pipe. Then the child process reads this data from the pipe and sends it to 130.217.X.X. The file and socket are eventually closed.

F. MySQL Monitoring

Alice has a MySQL daemon running and connects to it using the local client. She views all the values in the fruit table then adds an entry.

Command: *mysql>use food_db;*
Log:

1	Write,alice,2217,1682,1682,1681,**3s**,0, 0800000002666F6F645F6462
2	Read,mysql,2218,2038,1682,1681,**29s**,0, 08000000
3	Read,mysql,2218,2038,1682,1681,**29s**,0, 02666F6F645F6462
	...
4	Open,mysql,2218,2038,1682,1681,mysqld, ./food_db/fruit_tb.MYD,/var/lib/mysql/, 2,432,**31**

The MySQL client sends the command to the daemon through the socket descriptor pair 3 and 29. The daemon first reads the command type, then the parameters (0x666F6F645F6462 = food_db). The client is given information about the database selected, which results in the fruit_db.MYD database file being opened, returning 31.

Command: *mysql>select * from fruit_tb;*
Log:

1	Write,alice,2217,1682,1682,1681,**3s**,0,1700 00000373656C656374202A2066726F6D206672756 9745F7462
2	Read,mysql,2218,2038,1682,1681,**29s**,0, 17000000
3	Read,mysql,2218,2038,1682,1681,**29s**,0, 0373....66727569745F7462
4	Read,mysql,2218,2038,1682,1681,**31**,0,03000 709FE054170706C650000000000000000000030008 08FE0642616E616E610000000000000000300080 8FE064F72616E67650000000000000000
5	Write,mysql,2218,2038,1682,1681,**29s**,0, 0100....000007FE00002200
6	Read,alice,2217,1682,1682,1681,**3s**,0, 0100....000007FE00002200
7	Write,alice,2217,1682,1682,1681,1,0, 2B2D2D2D2D2D2D2D2D2B0A
8	Write,alice,2217,1682,1682,1681,1,0, 7C204E616D652020207C0A
	...

The client sends the command to the daemon, then the daemon reads the data from the database file descriptor 31. The entries in the table are Apple, Banana and Orange. The query result is sent back to the client which then writes to standard out in a the text table format, for example 0x4E616D65 = Name, which is the header cell of the table. The actual output is shown below:

Command: *mysql>insert into fruit_tb values('Mango');*
Log:

1	Write,alice,2217,1682,1682,1681,**3s**,0,2500 000003696E7365727420696E746F2066727569745 F74622076616C75657328274D616E676F2729
2	Read,mysql,2218,2038,1682,1681,**29s**,0,2500 0000
3	Read,mysql,2218,2038,1682,1681,**29s**,0,0369 .....6616C75657328274D616E676F2729
4	PWrite,mysql,2218,2038,1682,1681,**31**,60,03 000709FE054D616E676F00000000000000000
5	Write,mysql,2218,2038,1682,1681,**29s**,0,070 0000100010002000000
6	Read,alice,2217,1682,1682,1681,**3s**,0,07000 00100010002000000
7	Write,alice,2217,1682,1682,1681,1,0,51756 57279204F4B2C203120726F772061666665637465 642028302E303120736563290A

As before, the command is sent to the daemon from the client. Then the value Mango (0x4D616E676F) is written to the database along with some other data. The daemon sends back a response to the client which informs the user that the query was okay, and how many rows were affected.

G. Real Log Outputs

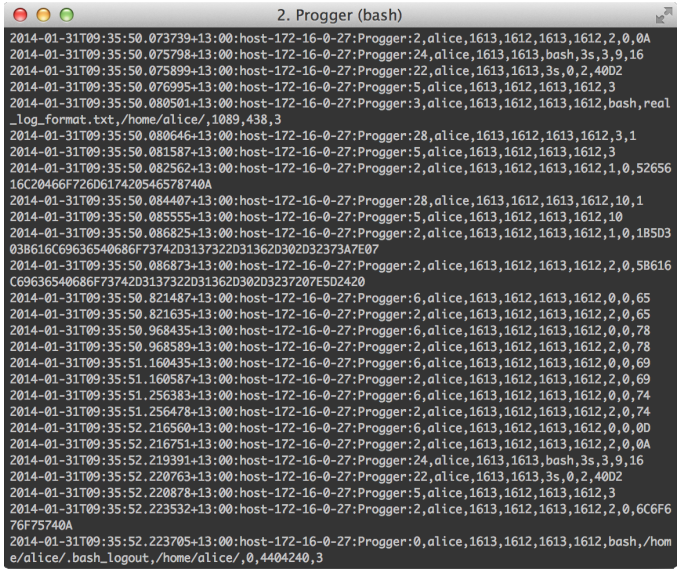

Fig. 7. Screen shot of a Progger log file.

A sample of a real Progger log is shown in Figure 7, where Alice echoed some text into a new file. The format used by *rsyslog* is a precise timestamp with time zone information, host name, then the Progger message.

VI. Tested Systems & Future Work

Progger has been developed the 2.6.32 kernel with CentOS 6.4 as the test operating system. Other kernel versions above 2.6.32 have been tested however because Progger is open source, if it does not compile with a certain kernel version, it will only require minor changes. For example if the method for getting the user ID changes, there should only be one line which needs to be edited for it to be fixed.

In the future, we would like to include support for IPv6 for Progger. With Progger logs, we can provide cloud data accountability and data tracking information on different levels of granularity and levels of detail. For example, we can envision dashboards with data provenance visualisations which empower administrators to understand the status and derivation history of certain data. Auditing interfaces can also be created on top of Progger to empower data governance audits.

VII. Conclusion

We presented Progger, a kernel-space data activity logging tool which address prior provenance tools' limitations by providing log tamper-evidence, prevention of fake or manual entries, an accurate timestamp synchronisation across several machines, efficient log space growth, and the accurate logging of root usage of the system. Resolving all these critical issues enables Progger to provide high assurance of data security and data activity audit. The trust relationships between data owner and cloud service providers will also be enhanced. We described Progger's design, implementation, data-centric system calls tracked, efficiency, and compelling results which paves the way for it to be a foundation for cloud data provenance tracking. Progger can be accessed at https://github.com/CROWLaboratory/Progger

VIII. Acknowledgements

This research was supported by the 2013 University of Waikato Research Trust Contestable Fund. The authors would also like to thank Brad Cowie, Alan Tan and Kang Du for their feedback on Progger.

References

[1] R. K. L. Ko, G. Goh, T. Mather, S. Jaini, and R. Lim, "Cloud Consumer Advocacy Questionnaire and Information Survey," Cloud Security Alliance Cloud Data Governance Working Group, Cloud Security Alliance, Tech. Rep., 2011.

[2] R. K. L. Ko, "Data Accountability in Cloud Systems," in *Security, Privacy and Trust in Cloud Systems*. Springer-Verlag, 2013.

[3] R. K. L. Ko, M. Kirchberg, and B. S. Lee, "From System-centric to Data-centric logging - Accountability, Trust and Security in Cloud Computing," in *Defense Science Research Conference and Expo*, 2011.

[4] R. K. L. Ko, B. S. Lee, and S. Pearson, "Towards Achieving Accountability, Auditability and Trust in Cloud Computing," in *Advances in Computing and Communication*, 2011.

[5] R. K. L. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee, "TrustCloud: A Framework for Accountability and Trust in Cloud Computing," in *Proceedings of IEEE World Congress on Services (SERVICES'11)*, 2011.

[6] Y. S. Tan, R. K. L. Ko, P. Jagadpramana, C. H. Suen, M. Kirchberg, T. H. Lim, B. S. Lee, A. Singla, K. Mermoud, D. Keller, and H. Duc, "Tracking of Data Leaving the Cloud," in *Proceedings of IEEE 11th International Conference on Trust , Security and Privacy in Computing and Communications (TrustCom'12)*, 2012.

[7] CloudKick. (2011) Cloudkick - cloud monitoring and management. Https://www.cloudkick.com/.

[8] S. E. Hansen and E. T. Atkins, "Automated system monitoring and notification with swatch," *Proc. of the 7th Systems Administration Conference (LISA VII) (USENIX Association, CA)*, p. 145, 1993.

[9] VMware. (2011) Performance monitoring for cloud services. Http://www.hyperic.com/products/cloud-status-monitoring.

[10] HyTrust. (2010) Hytrust appliance. Http://www.hytrust.com/product/overview/.

[11] G. H. Kim and E. H. Spafford, "Tripwire: A case study in integrity monitoring," in *Internet Besieged: Countering Cyberspace Scofflaws*, D. E. Denning and P. J. Denning, Eds. New York: ACM Press / Addison-Wesley, 1998, pp. 175–210.

[12] R. K. L. Ko, P. Jagadpramana, and B. S. Lee, "Flogger: A File-centric Logger for Monitoring File Access and Transfers with Cloud Computing Environments," in *3rd IEEE International Workshop on Security in e-Science and e-Research (ISSR'11), in conjunction with IEEE TrustCom'11*, 2011.

[13] K.-K. Muniswamy-Reddy, D. A.Holland, U. Braun, and M. Seltzer, "Provenance-aware Storage Systems," in *Proceedings of the Conference on USENIX'06 Annual Technical Conference (ATEC'06)*, 2006.

[14] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in Provenance Systems," in *Proceedings of the Conference of USENIX Annual Technical Conference (USENIX'09)*, 2009.

[15] P. Buneman, S. Khanna, and W. C. Tan, "Why and Where: A Characterization of Data Provenance," in *Proceedings of 8th International Conference on Database Theory (ICDT'01)*, 2001.

[16] U. Braun and A. Shinnar, "A Security Model for Provenance," Harvard University Computer Science, Tech. Rep. TR-04-06, 2006 (Accessed: 1/07/2013).

[17] C. F. Reilly and J. F. Naughton, "Exploring Provenance in a Distributed Job Execution System," *Provenance and Annotation of Data, Journal*, pp. pp 237–245, 2006.

[18] P. Reynolds, C. Killian, J. L. Wiener, J. C.Mogul, M. A.Shah, and A. Vahdat, "Pip: Detecting the Unexpected in Distributed Systems," in *Proceedings of 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

[19] Y. S. Tan, R. K. L. Ko, and G. Holmes, "Security and data accountability in distributed systems: A provenance survey," in *Proceedings of the 15th IEEE International Conference on High Performance Computing and Communications(IEEE HPCC13)*. ZhangJiaJie, China: IEEE Computer Society, 2013.

[20] O. Q. Zhang, M. Kirchberg, R. K. L. Ko, and B. S. Lee, "How to Track Your Data: The Case for Cloud Computing Provenance," in *Proceedings of IEEE 3rd International Conference on Cloud Computing Technology and Science (CloudCom'11)*, 2011.

[21] C. H. Suen, R. K. L. Ko, Y. S. Tan, P. Jagadpramana, and B. S. Lee, "S2Logger: End-to-End Data Tracking Mechanism for Cloud Data Provenance," in *Proceedings of 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'13)*, 2013.

[22] G. Kroah-Hartman, *Driving Me Nuts - Things You Never Should Do in the Kernel*. Linux Journal (Issue No. 133), May, 2005.

[23] O. P. Peter Jay Salzman, Michael Burian, *The Linux Kernel Module Programming Guide*, 2007 (ver 2.6.4).

[24] VMWare, "Timekeeping in vmware virtual machines," VMware, Tech. Rep., 2011.

[25] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters, "Building an encrypted and searchable audit log." vol. 4, pp. 5–6, 2004.

[26] D. Sandler, K. Derr, S. Crosby, and D. S. Wallach, *Finding the evidence in tamper-evident logs*, 2008.