Misdirection: Not The File You Thought You Wrote

Kyle Martin, Seth Troxler
kdmarti2@ncsu.edu, sltroxle@ncsu.edu

Abstract

Ransomware is a form of malware that seeks to extort its victims by denying file access through encryption and then demanding a ransom. Solutions that target ransomware have grown in usefulness, but no single solution is able to defeat ransomware with 0 data loss and without requiring some form of user intervention. In this paper, we present a ransomware prevention system called Misdirection. Misdirection seeks to address the above issues with existing solutions by utilizing proven IO heuristics and a copy-on-write style file system. To prove Misdirection's merit, a standard user environment was developed to deploy simulated ransomware. Evaluation was based on successful prevention of file damage without user interaction. IO performance evaluations of Misdirection were also performed. The outcome of our evaluation reveals that Misdirection has potential to be a valid protection mechanism against ransomware, although further improvements are necessary. Misdirection's approach of IO redirection & tracing process pedigree is a novel way of gauging the intent of a process in order to determine maliciousness.

1 Introduction

Ransomware is a form of malware that encrypts users files and demands payment for access. In the current threat landscape, this attack is one of the most profitable forms of cyber-crime, with Cryptowall causing over \$200M in losses between 2014 - 2016, with the average ransom demand being \$300 [16, 25, 20].

Currently there are several distinct families of ransomware, some being harder to stop than others, all potentially devastating to a user who becomes infected. Once the ransomware has executed, it is essentially impossible to recover encrypted files [13, 9]. A recent backup can be an acceptable option if available. If the backup is attached to the infected PC, however, it too could be encrypted. Keeping a backup that is not connected to the victimized device is a best practice in preventing oneself from being victimized by ransomware [13]. However, often backups are either not being maintained or the event for the backup to occur automatically has not yet happened leaving the target potentially vulnerable to ransomware.

The difficulty in preventing ransomwares is influenced by multitude of factors. First, ransomware is able to successfully complete its objective at user level [9]. Preventative measures like signature based endpoint protection have failed to detect and prevent evasive or new variants of ransomware [9, 23, 5, 22]. Even when a signature has been developed for detecting a new variant, the damage from malicious data encryption has already occurred or detection of ransomware varients is too slow to prevent zero data loss [22]. If a variant has been detected by AV, the usage of packers allows for the same variant to become undetectable [15]. The next logical steps for preventing ransomware are monitoring systems that detect ransomware-like action or automatic recovery [22, 5].

Despite recent advances in combating ransomware, user data loss can still occur due to a slow detection [22] or an inexperienced user having to do manual file recovery [5]. Newly-developed defenses are still potentially vulnerable to certain types of malware, such as multi-process malware. Multi-process malware is potentially able to bypass heuristic analysis that depends on a process not working in tandem with other processes [17]. Current state of ransomware defense system are focused on single process malware, where the ransomware IO pattern shown by Kharaz et. al in UNVEIL [9] are contained within a single process. It is our belief that as ransomware and malware defense mechanisms improve, attackers will migrate to multi-processes ransomware to bypass single process heuristics.

In this paper, we make the following contributions:

- A ransomware detection scheme based on factors such as the number of file deletes and the entropy of file writes.
- A process inheritance model wherein parents inherit the trustworthiness of their children, thus increasing success at detecting malware across multiple processes.
- A windows kernel driver with the above functionality embedded.

The remainder of this paper proceeds as follows. Section 2 gives a brief history of ransomware. Section 3 presents Misdirection's structure. Section 4 describes the design of our process inheritance structure, as well as our ransomware decision-making structure. Section 5 evaluates our design with regards to protecting a given directory from malicious encryption & deletion attempts, as well as examines the IO overhead incurred by our implementation. Section 6 discusses limitations of and suggested improvements to our work. Section 7 describes related work and makes comparisons between our implementation and other existing solutions. Section 8 concludes.

2 Background

Variants of ransomware can be found as early as 1989, where the AIDS Information Trojan restricted host data access [28]. Since 1989, ransomware has evolved and improved in methods of attack vectors and encryption.

Two common vectors through which malware may obtain access to a victims computer are phishing and drive-by downloads [20, 7, 26, 3]. Once ransomware has gained execution on a target's box, it will obtain or create encryption keys. The RSA key pair is generated on the attackers C&C server, while the ransomware downloads the public key [26, 20, 10, 27]. The RSA public key could then be used in two different ways. The first is using the key directly to encrypt files, while the second is using the key to encrypt an AES symmetric key and then using the AES key to encrypt files [10, 26, 20]. By using AES, as the encryption method, the attacker is able to generate a unique key for a victim and benefit from the faster encryption speed of AES compared to RSA. This unique key is then sent to the attacker's command and control server for safe keeping [20].

Once encryption has been completed, ransomware will employ various methods to alert the user that they have been comprised. These alerts will contain instructions for the user to follow in order to decrypt their files. These instructions will normally instruct the user to go to a tor website and submit a payment in bitcoin. [26]. These Tor websites provide a takedown resistant base of operations [21], which allows attackers to use the same payment website through multiple campaigns.

Bitcoins are cryptographically signed messages that facilitate money transfers. Bitcoin is often used as the medium of sending funds to attackers because it is more anonymous than standard payment methods. The private key used to authorize fund transfers is not explicitly tied to a real person, allowing the attacker to reduce the amount of information publicly available to a bitcoin address [10]. The anonymity provided by bitcoin has been scrutinized to some extent by [18] and has shown potential for information leakage. However, online bitcoin mixers exist that make user de-cloaking more difficult to accomplish [14]. Once the bitcoin payment has been received, the attacker will then proceed to unlock the users files.

2.1 Threat Model

The threat model and assumptions are proposed here. Ransomware will execute at the user level and will not compromise the kernel. Our mini-filter, being at the kernel level, will not be unloaded by malware as malware will run in the context of a non-administrator. Malware will not successfully gain administrative privileges. Also, ransomware will not perform code migration techniques to run as a white listed process, such as explorer. All user defined data is contained within the user's Documents folder. All processes in a process pedigree are guaranteed to exit, if at least one process in this pedigree has made at least one change to protected user data.

3 Overview

This research addresses potential future problems around multi-process ransomware, while removing user interaction in recovery of encrypted data. To achieve this goal, we implemented a Windows kernel mini-filter driver that acts as a process monitor for the system. In addition, Misdirection provides an IO sandbox that allows a process to have a private copy of a file through copy-on-write mechanisms.

Ransomware initially operates at user level. Therefore, in order to ensure that ransomware doesn't corrupt or avoid Misdirection, it is critical that Misdirection operate at kernel level. A filesystem minifilter driver is a driver filter that sits above the actual filesystem driver & storage stack drivers in the kernel. These filters are able to monitor all IO calls as they flow through the IO stack. An example IO stack can be seen in Figure 1. In this diagram, it can be seen that filters process IO calls in order based on position in the stack, or "altitude." Misdirection has a low altitude so that it will be positioned last on the stack. This prevents other existing minifilters on the stack from repeating Misdirection IO requests. IO passes both ways through this stack, however, our filter only watches IO calls going down the stack as all the required manipulations need to happen before the call is executed on the filesystem.

In order to provide an IO sandbox, our system implements a copy-on-write (CoW) filesystem. In a typical CoW filesystem, such as one described by [8], any writes to a file will create a copy of that file. In our implementation, accessing a protected file will create a private copy of that file, also known as a shadow copy. This allows each process to have its own copy of the file it is writing to. When a process exits, its shadow copy will be

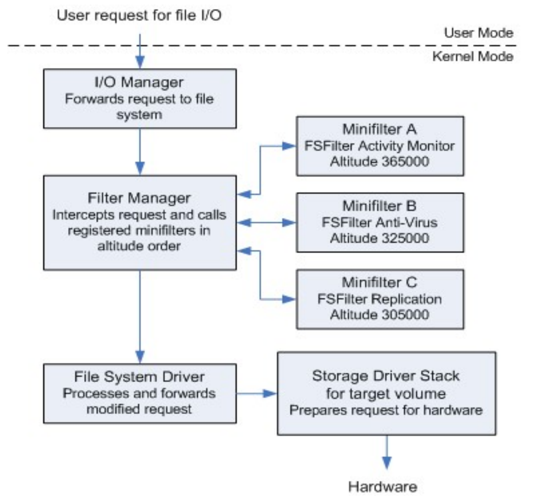

Figure 1: The MSDN example IO stack [2]. Misdirection sits below all other minifilters, right on top of the File System Driver.

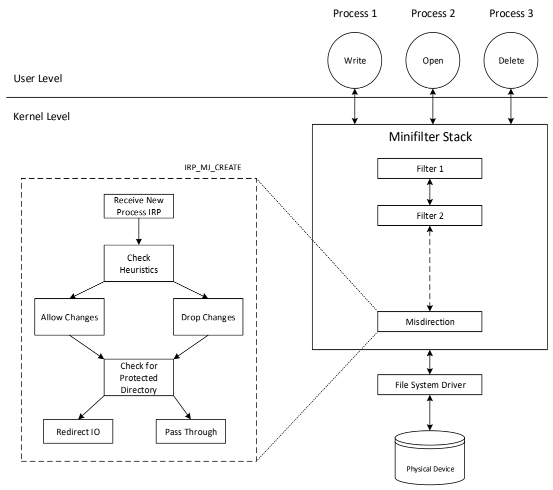

Figure 2: An overview of the design of Misdirection. Misdirection's position in the minifilter stack allows unrestricted access to all IO between the users and file system.

run against IO heuristics to gauge the intent of the process. Non-malicious changes will then be merged with the original or "real" file, and malicious changes will be dropped.

Whether or not a processes changes are allowed is determined by a set of IO heuristics. The two main heuristics are the number of deletes vs. the number of touched files and number of high entropy shadow files. To calculate entropy, the following formula is used:

$$c = \sum_{0}^{255} P_{B_i} log_2 \frac{1}{P_{B_i}}$$

In this formula, $P_{B_i} = \frac{F_i}{totalbytes}$, and F_i is the number of instances of byte value i in the array. This results in an entropy score between 0 and 8, 8 being a perfectly even distribution of bytes. File entropy has been used in various applications, such as malware detection [22, 24, 12].

With ransomware, it is expected that high-entropy files will be created as a result of the strong encryption used [22, 9]. This assumption allows us to flag high entropy inducing operations as potentially malicious in all but the case of a file that inherently has high entropy.

Entropy and file deletion heuristics have been applied to ransomware detection in other works, namely [22, 5, 9], however, in this work we build upon previous implementations. Misdirection utilizes some of these proven heuristics and adds a method to monitor process pedigrees. A graphical description of our system is given in Figure 2.

With Misdirection, we solve the following challenges:

- How to track ransomware that splits itself into multiple processes.
- How to update and properly apply benign changes to a file in a manner requiring no user interaction

The challenge of tracking multi-process ransomware is addressed in Misdirection by implementing a process-node linked list. The nodes in this linked-list contain information about other nodes they are related to (i.e. parent child relationships) as well as information about files that each process has accessed. This system allows tracking of both shadow and real files of a given process all the way up the its pedigree. Once all processes in a given pedigree have exited, heuristic analysis on all shadow files that these processes have generated is translated into a "trust score." If the trust score is above a defined threshold, the shadow file(s) will replace the original file(s) on disk, implementing the change. Conflicts are resolved by last write depending on which process's pedigree ends first.

4 Design

The design of Misdirection is broken down into several areas. First, we discuss how Misdirection monitors the file system. Then, we discuss methods used to protect files, track processes, and evaluate changes applied to files.

4.1 Filesystem Monitoring

Misdirection registers three different call backs to ensure accurate monitoring of the windows file system: IRP_MJ_CREATE, IRP_MJ_SET_INFORMATION, and IRP_MJ_WRITE [2]. When these *IO Request Packets* (IRPs) are sent through the miniflter stack, Misdirection will intercept, analyze, and perform actions to ensure the safety of user data. The minifilter protected files the files that are located in the users documents folder.

As previously mentioned, Misdirection registers function callbacks to trigger when specific IRPs are sent or received. When Misdirection is finished handling an IRP, the minifilter forwards the IRP to the next minifilter on the stack. Since Misdirection is last on the minifilter stack, the request is forwarded to the file system for processing. By intercepting the IRP before the file system receives the request, Misdirection is able to manipulate this request. The manipulation of IRPs allows Misdirection full IO control when a user level process attempts to access a file. Through this manipulation, Misdirection is able to fool user level processes into believing they are working on an original file when in fact they are editing a copy of that file [2].

By hooking a particular callback of IRP_MJ_CREATE, Misdirection monitors all of the IO access calls conducted by user level processes. While monitoring these processes, all IO requests that do not access a file within the protected directory are passed through unchanged. When a file in the protected directory is accessed, an IO redirect will occur regardless of the type of access. The process of IO redirection is as follows: when an IRP referring to a file in the protected directory is received, a copy of the targeted file will be made and the IRP_MJ_CREATE request will be manipulated accordingly. The manipulation will set the target file name to be that of the shadow copy pertaining to that request. Shadow copies are denoted by a file extension of "$PID," where PID is the requesting process's ID. This forces the user processes to interact with their shadow copies and not the actual file. There are two primary reasons for this design choice. First, it prevents direct access of user data by any non-whitelisted process. Secondly, having interaction be done on a copy of a file removes the need for real-time heuristics on file changes, as changes are examined only at the end of a process pedigree's lifespan. This IO redirection process gives our minifilter a copy-on-write style where all operations on a file are done to a copy of the original file.

4.2 Protecting Files From Processes

Since shadow files are unique to a process, it is important that they are protected from other user level processes. In order to achieve this, any file with an extension beginning "$PID" is blocked from being created, moved, or manipulated within the protected directory by a process whose PID does not match the file extension. This protection is accomplished by analyzing the extension of the file being accessed by a given process in an IRP_MJ_CREATE call. In order to prevent overwrite operations, a separate callback is registered utilizing IRP_MJ_SET_INFORMATION.

As mentioned earlier, when a process accesses a file in the protected directory a copy of that file will be created as a shadow file. If multiple processes access the same file, each process will have their own shadow copy. In the event that another process exits and a change is made to the original file, any existing shadow copies will not see this change. Access monitoring of shadow files is not needed as these files can be thought of as a per-process IO sand box.

The only exception to the creation of these shadow files occurs when the requesting process execution path is contained in Misdirection's IO whitelist. This IO whitelist is needed for particular system processes that generate an absurd amount of IO access request for every file in a directory. This whitelist prevents the excessive creation of shadow files in the protected directory. Also, a process whitelist is implemented to prevent certain processes from becoming parent nodes. Within the Windows operating systems, some processes will never end. These never ending process break the assumption that process pedigree's that change a file in the protected directory will end. This is crucial for IO heuristics to trigger as all IOTrace information is inherited to the oldest node.

4.3 Keeping Track of Processes

While IO redirection provides the fundamental components of creating an IO sandbox, keeping track of a process's pedigree is fundamental to Misdirection's IO heuristics. Misdirection keeps track of a processes pedigree and only allows IO changes when the oldest process node in a process pedigree returns.

Misdirection utilizes PsSetCreateProcessNotifyRoutineEx to monitor when a user level process is loaded or exited [2]. When a process opens a file in the protected directory, a process node representation is created in memory and recorded in a linked list structure. If the process is a child, it will be linked to its parent process node, assuming the parental node still exists in memory.

When a processes accesses a file, the path name to the real file and the shadow file will be recorded in a linked list structure called IOTrace. This structure will be appended to the IOTrace linked list contained in the accessing process's nodal representation. Refer to Figure 3 for clarity.

4.4 Handling Process Exits

When a process exits, Misdirection will intercept the event. A process that has no children or parent upon exit, but has a populated IOTrace, will be removed from the active process list and placed into a process analysis queue.

If an exiting process has a parent, all of its IOTrace will be appended to the oldest parent in the exiting process

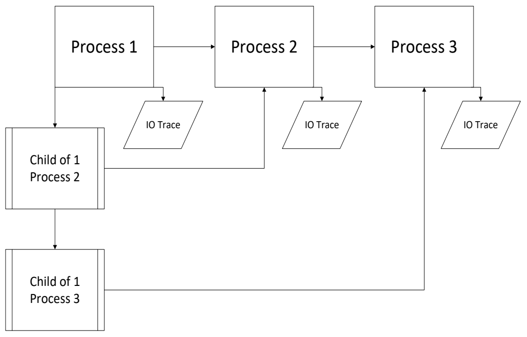

Figure 3: A representation of our process linked list structure. Note that the full process node of a child is at the same level at it's parent, and that the identifying factor is that a given child is also in its parent's child list structure.

pedigree. This is referred to as IO inheritance. This ensures that the IO events observed by Misdirection will all be analyzed in reference to the oldest node in a process pedigree. If an exiting process has a parent but no children the process will be removed completely from the active process list and deleted from the minifilter, as all of its IO has been inherited. If, upon exit, a process has a child process that is still active, the exiting node will be changed into a ghost process node in order to maintain process pedigree. A ghost process exists solely for the process pedigree, and as such will not cause another exit event for IO heuristics. Ghosts structures are cleaned and removed when the last child node in its pedigree has exited. This is important in the case where multiple children survive the parent node and one of the children is proven malicious. Since the ghost node structure has been maintained, all the children of this ghost process can now also be deemed malicious and have their changes dropped.

In order to initiate IO heuristics, an IRP_MJ_CREATE request is needed. This request causes the minifilter to check its process analysis queue for an exited process. Assuming one exists, handling of the IRP_MJ_CREATE is halted and the queued process's IOTrace is checked against Misdirection's IO heuristics. The results of the heuristic analysis provide a means of generating a trust score. If the trust score is greater than the trusted threshold, the process is deemed benign and the IOTrace of this process is moved to the IO change queue. If the trust score is below the threshold, the process will be deemed malicious and all of its IO will be dropped and its shadow file deleted.

Similarly to initiating IO heuristics, the IO change queue is acted upon when IRP_MJ_CREATE requests are received. Upon the event of IRP_MJ_CREATE a function will check to see if the IO queue contains IOTrace nodes. If nodes are present in this queue all shadow files in the list will overwrite their respective real files, allowing benign processes to make changes to the file system. It is in this way that Misdirection approves IO changes.

5 Evaluation

Misdirection was evaluated from two main standpoints: ransomware detection & prevention and IO overhead.

5.1 Ransomeware Detection & Prevention

Misdirection was developed for the Windows 7 64bit Operating System. The evaluation of malware detection was done on a VMWare Virtual Machine instance of the "Home Premium" edition of Windows, emulating a large ransomware target user base. Unfortunately, we were unable to test our system against current malware samples, and instead emulated the behavioral patterns that ransomware is known to follow.

We utilized the open source encryption software Gpg4win to simulate typical ransomware behavior [1]. Gpg4win supports both OpenPGP and S/MIME(X.509) cryptography standards and uses RSA for signing and encrypting files - making it ideal for generating strongly encrypted files. The software also supports a "delete after encrypting" feature, which is exactly the behavior desired for testing. While we lacked a proper malware testing suite, ransomware simulted via Gpg4win served as an initial benchmark for Misdirection's effectiveness.

In order to test the success of Misdirection's heuristics, a Powershell script was developed to enact ransomware behaviors on files in our protected directories. Once Misdirection was verified against a single file, we scripted an overwrite of all files in the protected directory with encrypted data. Misdirection correctly identified that the changes to the files were of high entropy data and rejected those changes.

The next test involved deletes of protected files. In this test, all files in the protected directory were again encrypted, but this time the script created encrypted versions of the files instead of overwriting the original files. Then, the script called for a delete of all the non-encrypted files in the directory. Misdirection correctly identified that a mass deletion attempt had occurred and that the process was associated with some high entropy files, and thus denied the attempted deletes. Note that the encrypted versions of the files still existed, as this does not impede a from accessing their data.

While this methodology works well enough for basic text files, we also had pdf, mp3, and jpg files to test with. However, since these files are all high entropy by default, our entropy heuristic is not useful in this context. Thus, we only ran the test on file deletions on the rest of our file set, and had similar success in this context. Given that, Misdirection performs as expected in terms of halting suspicious behavior based on these heuristics alone. This does not mean that we interpret Misdirection as a successful defense against ransomware. In order to make that statement, we would require a dedicated malware testing environment, as well as requiring more heuristics being implemented for Misdirection.

5.2 False Positives & Negatives

We were unable to evaluate all false positive/false negative scenarios, however, a few scenarios are given here that highlight the strengths & weaknesses of Misdirection in this regard. A false positive scenario for Misdirection would be an un-whitelisted process that invokes mass deletes on files in a protected directory. Another scenario would be a process that naturally outputs high entropy files, such as WinRAR or 7-Zip. In both these cases, the output constitutes creating a separate file from the original, and thus our filter would not impede the process. This is because Misdirection solely blocks actions that affect user data, and creating a new encrypted or zipped file does not fit this criteria. A false negative scenario would involve a process overwriting a file with lower entropy data or padding the encrypted data with a large number of the same byte to throw off the entropy score. In the case of writing low entropy data, that would indicate a weaker form of encryption was used, and that the encryption is potentially breakable by other means. In the case of throwing the entropy score, we would require input from another heuristic in order to make a decision on the files malicious intent.

5.3 IO Overhead

The IO overhead calculations were also done on a VM running Windows 7 Home Premium 64bit. The VM used for bench-marking was assigned 2x4.0GHz processor cores and 4GB of DDR4 RAM. In order to emulate a user directory, a variety of file types were used and placed in the users ”My Documents” folder. These files included 10 each of pdf, mp3, jpg, and txt files. The pdf, mp3, and jpg files were of random varying sizes, while the txt files were deliberately sized for testing. Sizes of these files can be seen in the performance graphs in 5. Due to the nature of Misdirection’s shadow file system, overhead is most likely to occur the first time a process opens a file (creating a shadow file) and when the last parent process in a process tree exits and its benign changes are copied to the real file. The process exit overhead is only incurred when the process is deemed to be benign, as malicious changes are not copied to the real file. Tests were done

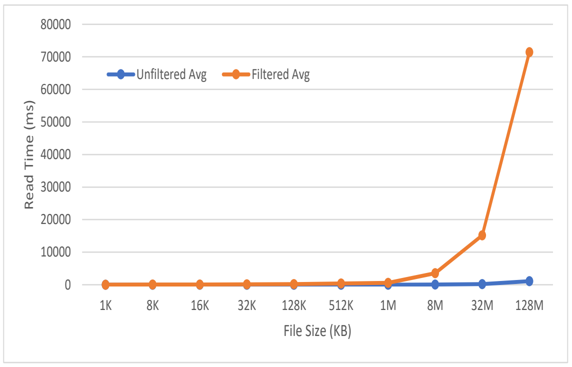

Figure 4: This curve demonstrates the increasingly noticeable overhead caused by large files. When attempting to open a 128MB file, a 50 second wait time was incurred!

to determine the time it took to open files and merge potential changes on a process exit, referred to henceforth as ”closing” a file.

In order to test open times of files, the powershell cmdlets ”measure-command”, ”get-content”, and ”start-process” were used. Measure-command reports the length of time it takes for a command to execute, making it ideal for testing read timing. The get-content cmdlet was used to read in the txt files, while the start-process cmdlet was used for pdf, mp3, and jpg files to open the files with their default Windows applications: Adobe Acrobat DC, Windows Media Player, and Windows Photo Viewer, respectively. Preliminary tests were run without our minifilter loaded to create a baseline for file open times. These tests ran 5 times each and an average open time per file was found. A snapshot of the VM was taken before running any tests, and was used as a restore point between each test for consistency. Once preliminary tests were completed, tests were run with Misdirection loaded in the same manner as stated above.

The results of these tests showed a distinct performance overhead for Misdirection, noticeable to the user when attempting to open files sized 32MB and up. The average open times and overhead for all files are given in 4. It is clear that a large amount of overhead is induced when opening files, ranging from 326% - 50,000% depending on the size of the file. This overhead is due to Misdirection having to create a copy of every file opened. The overhead of closing a file is dominated by the same issue of copying the entire file from the shadow file to the original file. Originally, our design only redirected IO during a write access call, which would have avoided this overhead in certain cases. However, due to our limitation of not being able to handle all forms of deletes, Misdirection redirects

Size (KB)	Unfiltered Avg (ms)	Filtered Avg (ms)	Overhead %
1	0.982	26.2	2668
8	0.511	56	10950
16	0.961	68.6	7136
32	0.630	142.6	22638
128	14.782	171.2	1158
512	2.945	360.8	12250
1024	5.801	550.2	9485
8096	41.890	3528.2	8423
32678	171.137	15013.4	8773
131072	844.646	70389	8334
txt files			

Size (KB)	Unfiltered Avg (ms)	Filtered Avg (ms)	Overhead %
286	16	1175.2	7345
322	13.4	347.2	2591
346	1.4	574	41000
380	9.8	751.8	7671
387	3.8	628.4	16537
464	16.6	590.4	3557
531	10.6	1303.2	12294
643	8.4	610.2	7264
1724	5	1861.8	37236
11039	17.4	8853.2	50880
pdf files			

Size (KB)	Unfiltered Avg (ms)	Filtered Avg (ms)	Overhead %
4757	38.4	4062.2	10579
5646	21.2	4892	23075
6014	20.4	5097.4	24987
6089	24	5336.4	22235
6195	59.4	5297.6	8919
6251	17.8	3007.8	16898
6524	1999	7388.6	370
6930	34.4	6207.8	18046
7661	39.2	6413.8	16362
8037	27.4	7478.2	27293
mp3 files			

Size (KB)	Unfiltered Avg (ms)	Filtered Avg (ms)	Overhead %
28	28.6	310.2	1085
58	19	432.6	2277
61	18.4	883.2	4800
80	22.6	631.6	2795
105	17.4	436.8	2510
136	29.8	932.6	3130
242	183.2	597.6	326
319	21.8	1204.6	5526
680	18.2	1792	9846
695	18.4	451.8	2455
jpg files			

Figure 5: These tables give the overhead of Misdirection based on file type and size.

all IO from one file to another, incurring this overhead in unnecessary cases.

6 Discussion

6.1 Limitations

Misdirection has several limitations that need to be resolved in order to be considered a valid form of defense against ransomware. In order for our IO heuristics to work, we developed a process inheritance model. For this model to function, an IO whitelist was required. This IO whitelist allows full write access to the Protected Directory." This whitelist contains several programs such as explorer, SearchIndexer, and svchost that constantly generate excessive amounts of IO requests for the files in every directory. These programs have full IO control over the protected directory, so simple code injection techniques will bypass our protection mechanism. Future developments will be needed in order to harden these whitelisted processes to code injection techniques or to create a method of changing the IO requests to read-only without breaking the operating system. We feel the second option of modifying the IRPs to read-only access is the preferred route.

Misdirection currently utilizes two IO heuristics to detect ransomware execution - deleted files and entropy. These heuristics, while effective, are not the only IO heuristics that could be used for detecting ransomware. Sdhash, a similarity scoring software, has been proven to be another solid heuristic [22] that would have fit well with Misdirection. This is because Misdirection already tracks both the shadow file and the actual file in the IOTrace, making similarity comparisons a function call away. Due to time constraints this feature was not implemented, but given ideal implementation time sdhash heuristics would provide a nice addition to Misdirection.

6.2 Suggested Improvements

The overhead of Misdirection when updating a file to reflect the changes conducted by a benign processes is very noticeable even through causal observation. This overhead is likely incurred by the amount of loops, comparisons, and floating-point operations needed for our driver to function. The primary factor in this overhead is the number of comparisons in loops, as this number increases exponentially based on the number of processes that are in the process list and with the amount of IO nodes in each of the processes nodes.

A major improvement to Misdirection would be to change how the data is represented in memory. A data structure that is conducive to look-ups and updates to the structure itself would be much more efficient than the current implementation. As previously mentioned, the process nodes are linked together in a sorted linked list based on process PIDs. The time complexity to find a process node is $\mathcal{O}(N)$ where N is the number of nodes in the list. This $\mathcal{O}(N)$ overhead is incurred due to the fact that we do not take advantage of having a sorted list, meaning that overhead could have been negated by simply not having a sorted list. Also, all update operations will have an overhead of $\mathcal{O}(N)$, where N is the number Process nodes in the linked list

To reduce the IO overhead induced by locating a process or updating the process linked list structure, a red-black tree could have been implemented so that locating and updating operations would have a time complexity of $\mathcal{O}(\log n)$ instead of $\mathcal{O}(n)$.

Another source of overhead for Misdirection was keeping track of a process's IO. As discussed previously, the IO is represented in a linked list structure where the head of the list is contained in the process node itself. When updating the IO structure to signify that a write has occurred, an IRP_MJ_WRITE call will be used to initiate string comparisons to locate and signify that a write has occurred within said process's IO. These string comparisons are quite costly to conduct, as the comparison operation is $\mathcal{O}(ny)$ where n is character the length of the string and y is the number of IO nodes contained in the list.

To reduce processing intensive string comparisons, a small, lightweight hashing algorithm that produces a word-sized hash could be utilized to implement a red-black tree structure. It is quite apparent that hash collisions will occur with this small spacial representation, but these will be resolved using string comparisons. While we are still using intensive string comparisons to resolve hash collisions, the overhead will be significantly reduced because we are not using string comparisons for every IO structure and instead only using them when a hash collision occurs. Also, as mentioned previously, reducing the amount of comparisons by using this red-black tree could reduce overhead to $\mathcal{O}(\log y) * n$ instead of $\mathcal{O}(ny)$.

A major source of overhead for Misdirection was conducting Shannon entropy analysis on large files. Shannon entropy is quite intensive on large files, such as movies, which are included in a normal users day to day usage. Also, using Shannon entropy to analyze certain files such as movies, music, jpeg, and pdf files that have built in compression has reduced effectiveness due to these types of files having naturally high entropy. To increase Misdirection's effectiveness and reduce overhead, it may be worth reducing the number of bytes examined for some file types to 512 or less, just enough to test the header and some data of these files.

Evaluation of Misdirection revealed slow file system update times. This was to be expected since file system updates only occur when an entire process pedigree has ended, meaning that if a single process has touched many files then file system operations are halted for quite a while. To reduce this wait for file system updates, a suggested improvement would be passing processing of IO heuristics to a user level process. While this doesn't completely alleviate the issue, it would reduce some processing delays.

Another interesting heuristic that could be implemented is the usage of Decoy Documents. While decoy documents have been discussed as a way of detecting insider threats, [4], the same concept could be applied on detecting malicious intent. By using these decoy documents as canaries for encryption, we can set the initial entropy of these files to a set amount and see if these files are manipluated to contain high entropy reflected as strong encryption. If a process has an IO Trace that cotains a high entropy canary file, we can deem that process pedigree as malicious.

7 Related Work

With ransomware being a lucrative business for malactors, new techniques and defense have been developed out of necessity to combat this threat. These new techniques can be loosely categorized as discovery, real time protection, or recovery.

Discovery entails finding new variants of ransomware that have gone undetected by existing infrastructure or tools. With Unveil, Kharaz et. al provide new insight on how to detect new variants of ransomware by breaking down the different sequences that ransomware may employ to encrypt a file. Using IO fingerprinting methods based on high entropy buffer writes, Kharraz et. al were able to detect a new family of ransomware.

The real time protection category of ransomware defense attempts to utilize real time IO heuristics to detect the intent of a process [22]. Scaife et. al employ such a strategy with CryptoDrop by utilizing several different IO heuristics. The primary IO heuristics consist of file extension changes, file similarity measurements, and the Shannon Entropy of a file. File type tunneling describes a characteristic of ransomware where a multitude of different file types are converted to a single file type post-encryption. Similarity measurement refers to the similarity of two files, which is computed through an external program, sdhash. Shannon entropy has been explained in 3. CryptoDrop was able to prevent data loss by detecting and stopping live ransomware samples.

The ransomware recovery strategy employed by Kolodenker et. al and Continella et. al in Paybreak and ShiledFs respectively demonstrates the ability to recover files that have been encrypted [11, 5]. Paybreak focuses on the ability to record the secret key used for encryption by monitoring Window's Crypto Apis, and then storing these decryption keys in an encrypted database that the sole user of the system has access to. When malicious encryption occurs the user will have the ability to manually recover encrypted files by decrypting them with the keys that are stored in paybreak's data base. Paybreak's success strongly deters the development of ransomware via the windows API for encryption and deters any new variants from using symmetric encryption. ShieldFS on the other hand implements the shadow file system provided by Microsoft as a means to automatically recover from malicious encryption of user data [5]. ShieldFs employs IO heuristics to detect if ransomware has caused damage to a user's data, and implements recovery by accessing the shadow file system.

Our work is mostly related to that of Scaife et. al and Continella et. al in CrytpoDrop and ShieldFS. Our main focus is to provide an IO sandbox for every process that is accessing a protected file. We utilize IO heuristics similar to that of related works in detecting maliciousness and provide automatic file recovery similar to that of ShieldFS, but with the included heuristic of maintaining a complete process child, parent relationship tree, also known as a process complete pedigree. By maintaining the pedigree of a process, we are not vulnerable to multi-process malware. This is because the changes to the file system by the children of a process are tracked until the root process returns. When this happens, Misdirection initiates is IO heuristics and makes decisions based on those changes.

Regarding copy-on-write filesystems, a lot of work has been done to reduce overhead and create a feasible solution for complete version control. What is meant by that is a system that can recover any file from any point in time, such as Devecsery et. al's Arnold, the Eidetic System [6]. This system can not only restore any file, it can restore itself to any state it has ever been in at with performance overhead averaging 8-12%. Of course, ransomware could still encrypt the data used to maintain that state, but with the techniques used in this paper are worth examining in an attempt to reduce our own overhead.

Another example of the viability of copy-on-write file systems is Elephant [19]. In this work, Santry et. al present a file system capable of automatically retaining important versions of users files. This file system operates by manipulating name spaces in order to maximize available disk storage and file version control. The points argued in this work reaffirm our comfort with the decision to make copies of files per process, which could potentially incur significant storage overhead. The authors of this work point out that modern file systems should offer more to the user, a policy Misdirection takes full advantage of.

8 Conclusion

In this paper, we proposed a system designed to stop multi-process ransomware and recover from any ransomware-inflicted damage without alerting the user. While these goals were met in concept, the implementation leaves much to be desired and has ample room for improvement. Additional heuristics, improved IO performance, a more efficient data structure implementation, and FLOPs in the Windows kernel are but a few of the areas we identified as weaknesses of Misdirection. We believe that the issue of multi-process malware is a real one and will need addressing sooner rather than later. While the implementation is flawed, we believe the concept to be solid and that future work could improve this to the point of user-viability.

References

[1] Gpg4win. `https://www.gpg4win.org/`.

[2] Msdn magazine: See the latest issue. `https://msdn.microsoft.com/en-us/dn308572.aspx/`.

[3] C. T. Alliance. Lucrative ransomware attacks: Analysis of the cryptowall version 3 threat. Technical report, Cyber Threat Alliance, October 2015.

[4] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. *Baiting Inside Attackers Using Decoy Documents*, pages 51–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[5] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi. Shieldfs: A self-healing, ransomware-aware filesystem. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 336–347, New York, NY, USA, 2016. ACM.

[6] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 525–540, Broomfield, CO, 2014. USENIX Association.

[7] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker. Manufacturing compromise: The emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 821–832, New York, NY, USA, 2012. ACM.

[8] S. Kasampalis. Copy on write based file systems performance analysis and implementation. Technical Report IMM-MSC-2010-63, Department of Informatics, Technical University of Denmark, 2010.

[9] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda. Unveil: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 757–772, Austin, TX, 2016. USENIX Association.

[10] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, DIMVA 2015, pages 3–24, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[11] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele. Paybreak: Defense against cryptographic ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 599–611, New York, NY, USA, 2017. ACM.

[12] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security Privacy*, 5(2):40–45, March 2007.

[13] S. Mehmood. Enterprise survival guide for ransomware attacks. Technical report, SANS Institute, April 2016.

[14] M. Mser, R. Bhme, and D. Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *2013 APWG eCrime Researchers Summit*, pages 1–14, Sept 2013.

[15] J. Oberheide, M. Bailey, and F. Jahanian. Polypack: An automated online packing service for optimal antivirus evasion. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 9–9, Berkeley, CA, USA, 2009. USENIX Association.

[16] G. O'Gorman and G. McDonald. Ransomware: A growing menace. Technical report, Symantec, November 2012.

[17] M. Ramilli, M. Bishop, and S. Sun. Multiprocess malware. In *2011 6th International Conference on Malicious and Unwanted Software*, pages 8–13, Oct 2011.

[18] F. Reid and M. Harrigan. *An Analysis of Anonymity in the Bitcoin System*, pages 197–223. Springer New York, New York, NY, 2013.

[19] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 110–123, New York, NY, USA, 1999. ACM.

[20] K. Savage, P. Coogan, and H. Lau. The evolution of ransomware. Technical report, Symantec, August 2015.

[21] N. Scaife, H. Carter, and P. Traynor. Oniondns: A seizure-resistant top-level domain. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 379–387, Sept 2015.

[22] N. Scaife, H. Carter, P. Traynor, K. R. B. Butler, undefined, undefined, undefined, and undefined. Cryptolock (and drop it): Stopping ransomware attacks on user data. *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 00:303–312, 2016.

[23] D. Sgandurra, L. Muoz-Gonzlez, R. Mohsen, and E. C. Lupu. Automated dynamic analysis of ransomware: Benefits, limitations and use for detection. *CoRR*, abs/1609.03020, 2016.

[24] I. Sorokin. Comparing files using structural entropy. *Journal in Computer Virology*, 7(4):259, 2011.

[25] Trendmicro. Ransomware bill seeks to curb the extortion malware epidemic, Apr 2016.

[26] J. Wyke and A. Ajjan. The current state of ransomware. Technical report, Sophos, April 2016.

[27] A. Young and M. Yung. Cryptovirology: extortion-based security threats and countermeasures. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 129–140, May 1996.

[28] A. L. Young. Cryptoviral extortion using microsoft's crypto api. *International Journal of Information Security*, 5(2):67–76, 2006.