Ransomware Prevention using Application Authentication-Based File Access Control

Or Ami
Department of Computer Science
Ben Gurion University of the Negev
oa@post.bgu.ac.il

Yuval Elovici
Department of Software and Information Systems Engineering
Ben Gurion University of the Negev
elovici@inter.net.il

Danny Hendler
Department of Computer Science
Ben Gurion University of the Negev
hendlerd@cs.bgu.ac.il

ABSTRACT

Ransomware emerged in recent years as one of the most significant cyber threats facing both individuals and organizations, inflicting global damage costs that are estimated upwards of $1 billion in 2016 alone [23]. The increase in the scale and impact of recent ransomware attacks highlights the need of finding effective countermeasures. We present *AntiBotics* – a novel system for application authentication-based file access control. AntiBotics enforces a file access-control policy by presenting periodic identification/authorization challenges.

We implemented AntiBotics for Windows. Our experimental evaluation shows that contemporary ransomware programs are unable to encrypt any of the files protected by AntiBotics and that the daily rate of challenges it presents to users is very low. We discuss possible ways in which future ransomware may attempt to attack AntiBotics and explain how these attacks can be thwarted.

KEYWORDS

Ransomware, Access Control, AntiBotics, Biometrics, Authorization

ACM Reference Format:
Or Ami, Yuval Elovici, and Danny Hendler. 2018. Ransomware Prevention using Application Authentication-Based File Access Control. In *Proceedings of ACM SAC Conference (SAC'18).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Ransomware is a type of malware which, upon infection, blocks access to the computer or its resources (such as files) until a ransom is paid. Advanced ransomware (a.k.a. crypto-ransomware, simply referred to in the following as ransomware) prevent data access by encrypting computer files. Users are then instructed to pay the ransom in order to decrypt them. Ransomware emerged in recent years as one of the most significant cyber threats facing both individuals and organizations [27]. The increasing popularity of digital currencies such as BitCoin [20] helped make ransomware attacks more common, since cryptocurrencies facilitate anonymous monetary transactions. Indeed, the number of known ransomware families rose from 30 in 2015 to 101 in 2016, with an increasing average ransom payment rising from $294 to $1077 [27].

SAC'18, April 9-13, 2018, Pau,France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The damage caused by ransomware is not just financial, as they can paralyze public institutions, such as hospitals, that need up-to-the-minute data to function properly. A recent key example is the large-scale WannaCry ransomware [28] attack conducted on May, 2017 that infected more than 230,000 computers in over 150 countries and caused major disruption to over 40 hospitals in the UK only [9, 21].

The increase in the scale and impact of recent ransomware attacks emphasizes the need of finding countermeasures. While several techniques for ransomware detection were proposed in recent years (e.g. [3, 4, 10, 22], see Section 5), effective and difficult-to-evade solutions for preventing ransowmare-afflicted data loss are still urgently needed.

Servers, PCs, smartphones, and other types of computing devices, all provide processes for logging into (a.k.a. signing into) their system, using user credentials (username/password) and/or biometric identification and authentication devices based on fingerprints, facial images, etc. Upon successful login, the user's authenticated identity determines which system resources in general, and computer files in particular, the user is authorized to access and in what manner.

Historically, login mechanisms were developed in order to prevent access by unauthorized humans who may have physical access to the device or may access it via a communication network. In recent years, however, the key threats to the security and integrity of computing devices and their files are posed by automated malicious software. Unfortunately, login mechanisms and biometric authentication/authorization devices in contemporary computing devices provide no defense against automated malware attacks.

We present *AntiBotics* - a novel system for application authentication-based file access control. AntiBotics harnesses biometric authentication mechanisms and human identification schemes such as CAPTCHA, in order to prevent malicious software in general, and ransomware in particular, from modifying or deleting files. AntiBotics enforces a file access-control policy by periodically presenting identification/authorization challenges, for ensuring that applications attempting to modify or delete files are invoked by an authorized user rather than by bots. Via flexible configuration options, AntiBotics allows an administrator to strike a good balance between the level of disruption incurred by users (resulting from the need to respond to challenges) and the level of security that is gained.

We implemented AntiBotics for Windows by developing a file system filter driver [18]. We conducted experimental evaluation of two aspects pertaining to our system's operation. First, using the driver, we have logged the file access patterns of a set of users over

an extended period of time. Based on our analysis of the logs we show that a very low daily rate of challenges presented to users suffices to authorize the file-modifying applications they use and protect the respective application files against unauthorized use. Second, we performed a large-scale evaluation on thousands of ransomware samples and verified that none of them was able to encrypt any of the files protected by AntiBotics.

Although it is encouraging that AntiBotics is able to completely block the operation of contemporary ransomware, the key question is the extent to which new ransomware will be able to evade it. We discuss possible attacks against application authentication-based file access control. Our analysis shows that it significantly raises the bar also for future ransomware.

The rest of this paper is organized as follows. In Section 2, we describe the high-level architecture of AntiBotics and provide simplified pseudo-code of the algorithm used by our Windows implementation to handle file I/O requests. We report on experimental evaluation in Section 3. We then discuss, in Section 4, possible countermeasures against
authentication-based file access control systems such as AntiBotics that may be used by future rasomware. Related work is surveyed in Section 5. We describe our conclusions and planned future work in Section 6.

2 ANTIBOTICS ARCHITECTURE AND IMPLEMENTATION

The high-level structure of the AntiBotics system is presented in Figure 1. AntiBotics implements an authentication-based access control mechanism that assigns *access permits* to *execution objects* based on a policy specified by an administrator and based on the responses to challenges that are presented upon attempts to modify or delete files. The key components of the system, presented by Figure 1, are: 1) the Policy Enforcement Driver (PED), 2) the Policy Specification Interface (PSI), and 3) the Challenge-Response Generator (CRG). A description of these modules, their functionality and their interactions is now provided. This is followed by a more technical discussion of the implementation.

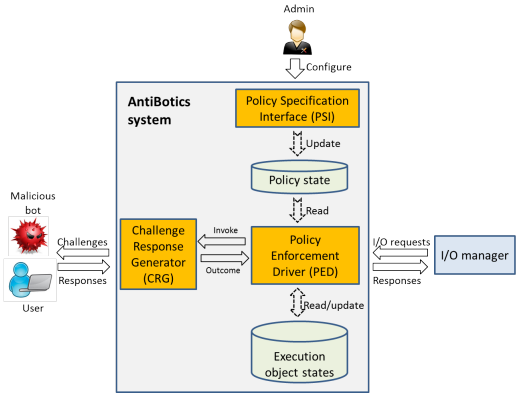


Figure 1: High-level structure of the AntiBotics system.

2.1 The Policy Enforcement Driver (PED)

The PED is the AntiBotics module that interacts with the system software in order to enforce file access-control according to the policy specified by a system administrator using the PSI. In operating systems supporting 3'rd party file system drivers such as Windows, the PED may be installed as a file system driver.

Figure 2 depicts a simplified view of the I/O system software stack of a typical PC. Applications operate in user mode wherein their access to system resources is limited. In order to perform I/O in general, or access files in particular, applications must invoke system calls which trigger the execution of operating system (OS) services running in kernel mode. Unlike code running in user mode, OS code has access to hardware devices. I/O requests are directed to an OS module called the I/O manager, which provides interfaces that allow I/O devices in general (and storage devices in particular) to be discovered, organized and operated.

When an application issues a system call in order to invoke some operation on a file (e.g., to read the file, write to it or delete it), a corresponding I/O request is generated and delivered to the I/O manager which, in turn, directs the request to a set of device drivers that have registered to handle this type of I/O requests. In modern operating systems, these device drivers are able to perform pre-processing and post-processing prior to dispatching the request to the block device driver which communicates directly with the appropriate storage device.

The AntiBotics PED module is implemented as a Windows file-system filter driver. We proceed to describe its functionality and interactions with other modules and defer a more technical discussion of its implementation to Section 2.4.

The PED receives file I/O requests from the I/O manager. Each request specifies the type of the requested file operation and the pathname of the file to which the operation should be applied. The PED is also able to obtain data that identifies the system entity requesting the operation, such as the process ID (PID) and the pathname of the program that invokes the I/O operation. AntiBotics maps this latter data to an *execution object* and maintains the state of execution objects.

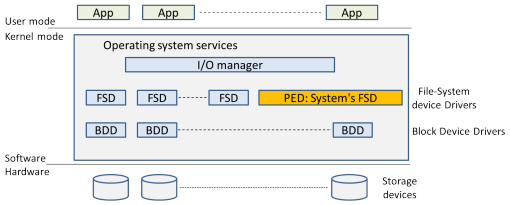


Figure 2: I/O system and PED file-system driver.

The PED allows read requests to proceed but considers file I/O requests that may allow a program to rename, modify or delete a file based on the policy state (edited by the policy specification interface described in Section 2.2) and the state of the requesting execution object. After consulting the policy state and the state of the execution object, the PED considers the I/O request and proceeds in one of the following manners: 1) The execution object is allowed to make the requested I/O operation and the request is

dispatched for execution; 2) The execution object is not allowed to make the requested I/O operation. The request is not dispatched for execution; 3) A challenge is issued. The type of the challenge (in our current implementation either fingerprint scan or CAPTCHA) is determined according to the policy state. The goal of the challenge is to ensure that a user (rather than a bot) initiated the file operation and, optionally, to verify that the user is authorized to perform the operation. If the challenge is responded to successfully within a (configurable) timeout, the execution object is allowed to make the access and is granted a time-limited permit, otherwise the request is refused and is not dispatched for execution.

2.2 The Policy Specification Interface (PSI)

The policy specification interface is a GUI program that allows the administrator to configure the AntiBotics policy. It can only be invoked in admin mode. The administrator's selections made using the PSI are recorded in the policy state and then used by the PED for deciding on whether or not to allow file access and for determining when and how often to invoke the CRG for issuing challenges.

Table 1 lists the configuration parameters that can be set by the PSI. The *Folder policy type* and *Folders list* parameters specify which folders are protected by AntiBotics and which are not. When the *Folder policy type* parameter assumes value 'Inclusive', the default is not to protect and the list specifies the pathnames of protected folders. Otherwise the parameter assumes value 'Exclusive' indicating that the default is to protect, so the list specifies the pathnames of folders that are excluded from protection. For example, the *AppData* folder where Windows programs often create temporary files may be excluded from protection by including it in the folders list and setting *Folder policy type='Exclusive'*.

The *Protected extensions* parameter is a list of file extensions that are to be protected by AntiBotics, such as, e.g., those of office documents (doc*, xls*), image files (.jpg, .png), or database files (.mdb), to name a few. The *Execution object type* parameter determines the granularity of authentication. When it assumes value 'PID', challenges are issued and permits are granted to specific processes. Otherwise it assumes value 'PROG', signifying that authentication is to be done in coarser granularity, per executable pathname. We discuss the implications of different *Execution object type* options further in Section 2.4.

The *Challenge type* parameter specifies the type of challenges issued by AntiBotics. The current AntiBotics implementation supports authentication using either fingerprint scanning (this is the default value) or CAPTCHA challenges. The *Challenge timeout* parameter determines the period of time (in seconds) within which a user must respond to a challenge. We describe the challenge-response mechanism in more details in Section 2.3.

The *Permit duration* parameter specifies the duration (in minutes) of the permit granted to an execution object. The *Permit scope* parameter determines which files an execution object that is granted a permit may access as long as the permit is active (that is, not expired). The permit can be either limited to the file that triggered the challenge (FILE), limited to files of the same type (TYPE), or allow access to *any* file type (ALL). Figure 3 presents a screenshot of the PSI.

2.3 The Challenge Response Generator (CRG)

The CRG is invoked by the PED in admin mode whenever the latter concludes (after consulting the policy state and the execution object states data structures) that an attempt to modify or delete a file made by an execution object cannot be allowed to proceed before a new challenge is successfully responded to. The authentication method used by the challenge is determined according to the policy state.

The current AntiBotics implementation supports fingerprint scan (this is the default value) or CAPTCHA challenges. Figure 4 presents a screenshot of such a challenge. The challenge provides the user with the following information: Identification of the execution object – the program pathname and process ID, name of the file about to be modified/delete, and the time of access attempt.

If the challenge is responded to successfully within the timeout configured in the policy state, a positive response is sent to the PED. Otherwise, when the timeout expires, a negative response is sent to the PED but the challenge window remains on screen. If and when the user will attempt to respond to it, she will receive a message that it was timed-out and a new challenge will be presented. A rate-limiting mechanism prevents execution objects from generating challenges at a too-high rate.

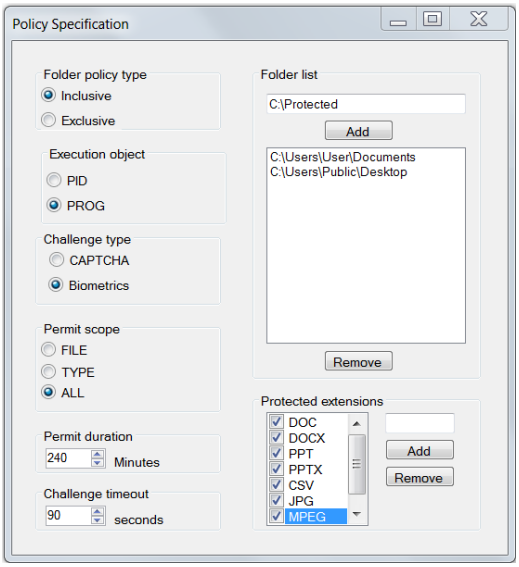


Figure 3: Policy specification interface screenshot.

2.4 Windows Implementation

We implemented AntiBotics for Windows systems. We chose to implement the PED module as a file system filter driver (FSFD) [1, 2]. Filter drivers allow modifying the behavior of a file system by logging, modifying or preventing I/O operations. The ability to inspect file system requests and to optionally modify or prevent them is useful for many applications, such as antivirus, file encryption and remote file replication, to name a few.

Figure 5 presents a simplistic view of the structure of a Windows file system drivers stack, consisting of several file system filter

Table 1: AntiBotics Configuration Parameters.

Name	Description	Default
Folder policy type	Folder policy specification type: Inclusive or Exclusive.	Inclusive
Folders list	List of protected or unprotected folders.	Desktop, Documents
Protected extensions	List of file extensions to be protected.	-
Execution object type	Execution object type: PID or PROG.	PROG
Challenge type	Type of challenges presented for authorizing file access.	BIOMETRIC
Challenge timeout	Time (in seconds) within which challenge must be responded to.	90 seconds
Permit duration	Permit duration (in minutes).	240 minutes
Permit scope	Files permit applies to: FILE, TYPE or ALL.	ALL

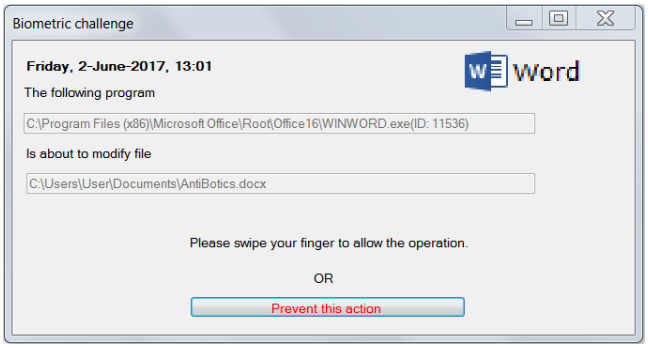


Figure 4: Screenshot of Fingerprint scan challenge.

drivers and a single storage device driver. Typically, I/O requests (termed I/O request packets – IRPs) are transferred from one driver to the other down the chain of drivers in the stack and responses are transferred back upwards in the reverse order. The AntiBotics PED is attached as the bottommost filter driver. Since a file system drivers stack such as that of Figure 5 exists per each file system volume, the PED is added in general to (the bottom of) several such stacks.

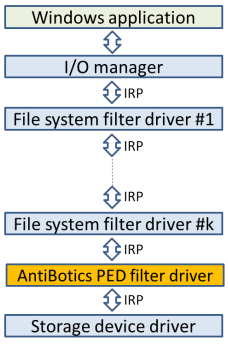


Figure 5: Windows file system driver stack (simplistic).

As we explain in Section 4, filter drivers provide robustness against attempts to unload them after system boot. In order to reduce development time, our implementation uses the Eldos CallbackFilter platform which implements a legacy file system filter driver and exposes an API for the filter driver's management and processing [7].

Algorithm 1 presents a simplified pseudo-code of the manner in which the PED handles file I/O requests.[1] The input consists of the pathname of the file to which the operation is about to be applied (*targetFile*) and the IRP structure.

Handling a file I/O request starts by obtaining the execution object (line 3). Execution then switches according to the type of the requested operation. If file creation is attempted and the policy specifies that the file is to be protected, a permit for the new file is automatically generated and associated with the execution object (lines 5–9). This allows programs to modify the files they created without presenting challenges to the user, as long as the permit generated upon creation remains active. AntiBotics maintains a hash map named *permits* which maps execution objects to their sets of permits. This enables quick lookups of execution object permits.

When an execution object attempts to open a protected file (line 10), AntiBotics checks if the file is about to be opened with write permission (line 11). If this is the case, then a permit for opening the file may be required and the *mod* flag is set in line 12 to ensure that the target file is handled further in lines 27–30. We note that a possible alternative for preventing a file open in write mode that may be considered is to implement *lazy protection* by allowing the file open to succeed and enforce protection only when the execution object invokes a write operation. As we explain in Section 3, however, this alternative strategy fails against some existing ransomware.

Moving a file from one position to another in the directories hierarchy is technically a rename operation applied to the file. Rename operations must be protected against malicious use. As an example, consider a file named *File.txt* in a folder *C*: *FolderA*. In order to move this file to folder *C*:*FolderB*, one can rename the file's pathname from *C*:*FolderA* *File.txt* to *C*: *FolderB* *File.txt*. If a file with the destination pathname already exists, an override option is available that will overwrite this file. Indeed, some ransomware use this option in order to delete user files without explicitly invoking a delete or write operation. Consequently, if a rename request requires to modify or override a protected file, the execution object must hold an active permit for the destination file (as well as for the target).

Rename operations are the more complicated case and are dealt with as follows. First, the *mod* flag is set in line 15 to ensure that the file to which the rename operation is applied (the target file) is handled further in lines 27–30. Then, if the destination file is an execution object, all its permits are deleted in lines 16–17 (if it is

[1] In the actual implementation, the pseudo-code of Algorithm 1 is partitioned between several functions, each dealing with specific I/O requests.

not an execution object, then these two lines have no effect). As we explain in Section 4, this is in order to block possible future *path hijacking* attacks that attempt "stealing" AntiBotics permits. If the operation is about to rename a folder, then the *folderRen* function is called (in line 18. Note that we assume that the expressions in this line are evaluated from left to right) in order to block possible future *folder exclusion attacks* (see Section 4). The *folderRen* function is described shortly. If it returns *true*, signifying that the rename is about to remove protection for some files in the renamed folder, the *authorize* function (explained soon) is called to determine whether the folder rename should be allowed (line 19). Otherwise, if the destination file exists and should be protected (line 20), the *authorize* function is called to determine whether overwriting it should be allowed (line 21).

When an execution object attempts to perform either one of the WRITE, DELETE, SETEOF or SETALLOCATIONSIZE operation types (the latter two change the file's size), AntiBotics marks that the execution object requests to modify the target file (lines 23–24).

After exiting the switch of lines 4–25, AntiBotics checks whether the target file is about to be modified (line 25). If this is the case and it should be protected (line 27), the *authorize* function (described next) is invoked to determine whether modification should be allowed and its response is recorded in the *processRequest* flag (line 28).

If the target is an execution object, all its permits are deleted in line 30 in order to block *path hijacking* attacks against AntiBotics (see Section 4). Finally, if access to both target and destination files (if this is a rename operation) is to be allowed, the IRP is passed on to the next stack driver, otherwise the requested access is denied (lines 32–36).

The *authorize* function is called (in lines **21** and **28** of *handleRequest*) for determining whether or not a file I/O request should be allowed. It receives as operands a reference to the execution object *EO* that made the request and the name of the *targetFile*.

First, the permits of the execution object are retrieved (line 37). If at least one of these permits allows modifying/deleting *targetFile*, *authorize* returns *true*, notifying *handleRequest* that access is allowed (lines 38–42). If no such permit exists, the CRG is invoked for issuing a new challenge in line 43 (see Section 2.3).

If the challenge was responded to successfully within the challenge timeout, the CRG returns a SUCCESS indication. In this case, a new permit for the target file is created and added to the execution object's permits set and *authorize* returns *true* (lines 44–46), notifying *handleRequest* that the request is allowed, otherwise *false* is returned (line 48), notifying *handleRequest* that the request should be rejected.

The *folderRen* function is called in line 18 of *handleRequest* when a folder is about to be renamed. Its tasks are to update the PSI's *folders list* if required, and to determine if the rename might be a *folder exclusion attack* for removing file protection (see Section 4). To perform its first task, the function iterates over the PSI folders list (lines 52–63) in order to find paths that are descendants of targetFile and to update their pathname.

To perform its second task, *folderRen* checks in line 64 whether the target folder is protected but will lose its protection after being renamed and returns *true* or *false* accordingly.

Algorithm 1: *handleRequest*($targetFile, irp$)

```
1  mod ← false, processRequest ← true
2  opType ← irp.requestType
3  EO ← getExecObject()
4  switch opType do
5      case CREATE : do
6          if isProtected(targetFile) then
7              newPermit ← createPermit(targetFile)
8              permits[EO] ← permits[EO] ⋃ {newPermit}
9          break ;
10     case OPEN : do
11         if isRW(irp.AccessMode) then
12             mod ← true
13         break;
14     case RENAME do
15         mod ← true
16         destFile ← irp.destFile
17         permits[destFile] ← ∅
18         if isFolder(targetFile) & folderRen(targetFile, destFile)
             then
19             processRequest ← authorize(EO, targetFile)
20         else if exists(destFile) & isProtected(destFile) then
21         processRequest ← authorize(EO, destFile)
22         break;
23     case WRITE,DELETE,SETEOF,SETALLOCSIZE: do
24         mod ← true
25 end
26 if mod then
27     if isProtected(targetFile) then
28         processRequest & = authorize(EO, targetFile)
29     end
30     permits[targetFile] ← ∅ ;
31 end
32 if processRequest then
33     Pass the IRP to the next driver
34 else
35     return accessDenied
36 end
```

3 EXPERIMENTAL EVALUATION

In the following section we report on two experiments we conducted. In the User File-Access Patterns experiment, we logged the file-related activity of a population of users over an extended period of time. We then analyzed the resulting log files in order to find a good tradeoff point (expressed by specific values of PSI parameters) between the level of security provided by AntiBotics and the level of disruption incurred by users.

In the Ransomware Prevention experiment, we tested a large number of contemporary ransomware samples downloaded from VirusTotal [25]. For each malware, we recorded the types of files it attacks and the manner in which it attempts to modify them. We verified that AntiBotics succeeds in blocking all these attacks.

Algorithm 2: *authorize*(*EO*, *targetFile*)

```
EOpermits ← permits[EO]
for permit in EOpermits do
    if permit.allows(targetFile) then
        return true
    end
end
if CRG.issueChallenge()==SUCCESS then
    newPermit ← createPermit(targetFile)
    permits[EO] ← permits[EO] ⋃ {newPermit}
    return true
else
    return false
end
```

Algorithm 3: *folderRen*(*targetName*, *destName*)

```
targetProtected ← PSI.FolderPolicy == Exclusive
destProtected ← tagetProtected
for folder in Folders list do
    if targetName starts with folder then
        targetProtected ← PSI.FolderPolicy == Inclusive
    end
    if destName starts with folder then
        destProtected ← PSI.FolderPolicy == Inclusive
    end
    if folder starts with targetName then
        newPath ← folder.replace(targetName, destName)
        PSI.FolderList ← PSI.FolderList \ {folder} ⋃ {newPath}
    end
end
return targetProtected and not destProtected
```

Surprisingly, as we describe in Section 3.2, this necessitated a change of our initial design.

3.1 User File-Access Patterns Experiment

To conduct this experiment, we installed a modified version of the AntiBotics Windows PED driver in the personal computers of 36 employees of an R&D organization, 33 of which are software developers and the rest belong to administrative staff. The modified driver generates a log line per every file I/O operation that modifies, deletes, renames or opens a file in write mode (see Algorithm 1) but does not provide any protection and does not trigger any challenges. File types to which accesses are logged include all file types attacked by contemporary ransomware, such as office files, images and movies, and source-code extensions.

Logging was conducted for one month during March, 2017. We then analyzed the logs in order to compute the rate in which challenges will be presented to users as a function of the *Permit duration*, *Permit scope* and *Execution object type* parameter values, assuming all folders are protected. Figure 6 presents the average number of challenges per day that would be required on average according to the logs as a function of the permit duration, assuming that *Permit scope* is set to 'All'. Blue bars (respectively red bars) present the number of challenges if *Execution object type* is set to 'PID' (respectively, set to 'PROG').

As can be expected, the number of challenges is a monotonically non-increasing function of the permit duration. Setting the object type to 'PROG' always results in a lower number of challenges as compared with setting it to 'PID'. This is because if a permit is granted to a program then it applies to all processes executing that program but when the type equals 'PID' a challenge has to be issued for each such process separately. For most of the range values, authentication at the level of programs produces below 60% the number of challenges compared with authenticating at the process level.

As we discuss in section 4, program-level authentication may allow more future ransomware attacks against AntiBotics as compared with process-level authentication, but we estimate these may be blocked. We therefore use program-level authentication in the rest of our experimental evaluation. Based on the user access pattern experiment we choose to conduct the rest of the experimental evaluation with the parameter *Permit scope*=ALL and *Permit duration*=4 hours as these values strike a very good balance between the daily number of challenges required (only slightly more than 3) and the size of the window of vulnerability to some attack types against AntiBotics (see section 4). Figure 7 presents a partition of the challenges according to the types of files that triggered them for the permit duration we chose. The largest gap between challenge rates of program-level and process-level authentication is when .csv files are accessed since some Python development environments generate multiple processes running the Python interpreter.

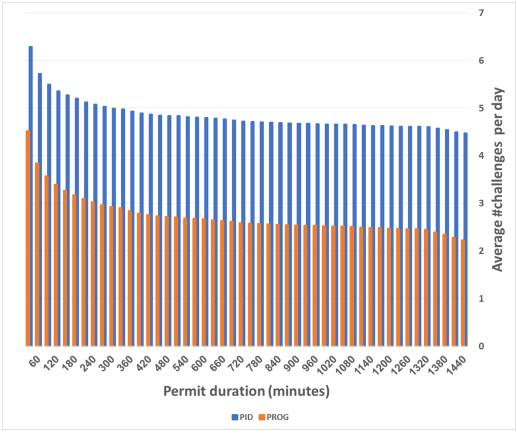
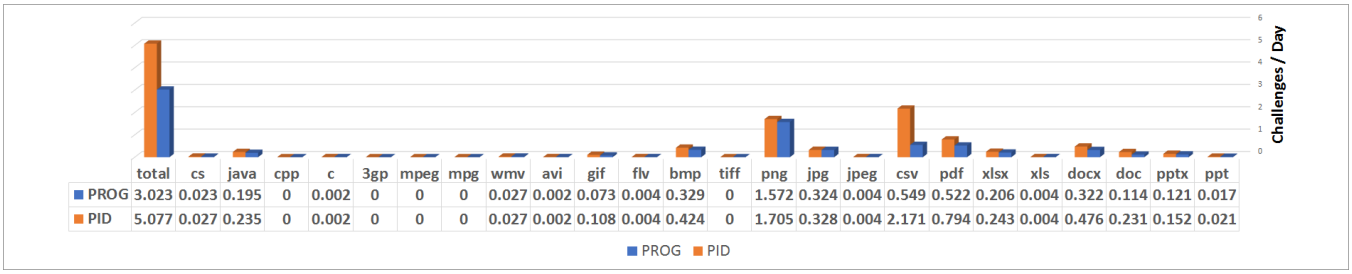


Figure 6: Average number of challenges per day as a function of permit duration for *PermitScope*='ALL'.

3.2 Ransomware Prevention Experiment

The ransomware prevention experiment was conducted using Cuckoo Sandbox. Cuckoo is an open source automated malware analysis system [29]. We used Cuckoo in order to automate the process of testing 13,301 ransomware samples that we downloaded from



	total	cs	java	cpp	c	3gp	mpeg	mpg	wmv	avi	gif	flv	bmp	tiff	png	jpg	jpeg	csv	pdf	xlsx	xls	docx	doc	pptx	ppt
PROG	3.023	0.023	0.195	0	0.002	0	0	0	0.027	0.002	0.073	0.004	0.329	0	1.572	0.324	0.004	0.549	0.522	0.206	0.004	0.322	0.114	0.121	0.017
PID	5.077	0.027	0.235	0	0.002	0	0	0	0.027	0.002	0.108	0.004	0.424	0	1.705	0.328	0.004	2.171	0.794	0.243	0.004	0.476	0.231	0.152	0.021

Figure 7: Required challenges distribution per file type, *Permit duration*=240mins and *PermitScope*='ALL'.

VirusTotal in the course of April, 2017. Each sample was ran by Cuckoo in a separate Windows 7 virtual machine (VM). We installed in each VM the AntiBotics system. We populated the following 4 locations in the file system of each VM with 46 files, each with its own distinct file extension: *C:\Protected*, *Desktop*, *Documents*, and *C:\NotProtected*. AntiBotics was configured such that the *C:\NotProtected* folder is the only folder excluded from protection. This allowed us to determine if the tested ransomware was *active*, by checking whether or not one or more of the files in C:*NotProtected* changed or was deleted in the course of the experiment.[2] In terms of protected extensions, AntiBotics was configured to protect all 46 extensions, which include file types attacked by contemporary ransomware (see extension names in Figure 7).

In each test, a single ransomware sample was ran until either all its processes terminated or a 20-minute timeout expired. Using Cuckoo, a report was generated after each test, listing all files that existed when the test started. For each such file, the report contains 2 hash values, computed based on its contents before and after the test (or a special value indicating that the file did not exist after the test). Using these tests, we were able to compute the rates of encrypted files for protected and unprotected folders.

Table 2 presents the results of the experiment per each ransomware family. The number presented next to each family is the number of samples that were active, that is, the number of family samples that modified at least a single file. Columns 3 and 4 present the rates (in percentages) of attack success on protected and unprotected folders. These rates were calculated by computing for each test of an active sample the rates of modified/deleted files in protected and unprotected folders and then computing the average rates per family. As can be seen, the rates of encrypted files in the unprotected folder are positive for all malware families. In contrast, none of the files in the rest of the file system was changed by any ransomware sample.

The fact that none of the protected files was encrypted by any of the ransomware samples may seem obvious, since, as of now, ransomware do not need to circumvent authentication-based file access control mechanisms (we hope this will change in the near future). We were therefore surprised to find, upon performing the ransomware prevention experiment on a previous version of AntiBotics, that the Spora ransomware family was able to encrypt files also in protected folders! We now explain how this came into pass.

[2]A ransomware may become inactive if it is unable to communicate with its C&C server.

Table 2: Ransomware family encryption rates

Family	#	Encryption Rate Protected	Encryption Rate Unprotected
Locky	164	0	0.6521
Cryptowall	13	0	0.4347
CTBLocker	54	0	0.6521
Sage	570	0	0.7369
Criakl	5	0	0.7173
Cerber	2310	0	0.6521
Revenage	49	0	0.7391
Spora	29	0	0.5072
TeslyaCrypt	12	0	0.5326
JigSaw	7	0	0.6739
WannaCryptor	22	0	0.8260

The previous version of AntiBotics employed *lazy protection*. That is, file open operations were always allowed to succeed and protection was enforced only when the execution object invoked a write operation. It turns out that Spora ransomware encrypt files by using the following mechanism: after opening a victim file in write mode, file contents are mapped into the Spora process address space and encrypted by direct writes to RAM. Thus, the Spora process never issues file write operations. Instead, writes to disk are performed by the Windows system process (PID 4) when writing dirty memory pages to disk.

We do not know whether Spora behaves this way in order to evade detection or for improving performance. Regardless of its motivation, since AntiBotics does not restrict file accesses made by the system process, this allowed Spora to successfully encrypt protected files. We fixed this problem by employing a more eager protection scheme in which we block file open requests in write mode, as explained in Section 2.4.

4 DISCUSSION OF POSSIBLE RANSOMWARE COUNTERMEASURES

Our experimental evaluation shows that AntiBotics provides 100% protection against contemporary ransomware. In the following, we analyze a few possible ways in which future ransomware may attempt to attack the authentication-based file access control scheme employed by it.

Cross-Process Code injection: In this type of attack, a malicious program places its code within the address space of another process and executes it. A ransomware may try to identify a legitimate

running process that has an active permit and inject into it malicious code in order to gain file write access without triggering AntiBotics challenges. Another alternative is for the ransomware to inject itself to the legitimate process and wait until it is granted a permit.

While this type of attack is possible, it is more difficult than regular cross-process code injection, since the attacker must find a window of vulnerability when the legitimate process possesses permits for the files it wants to encrypt. Moreover, if the AntiBotics *Permit scope* parameter is set conservatively to 'FILE' (hence a permit is only granted for accessing the file that triggered a challenge), then the extent of damage inflicted by the ransomware is greatly reduced. More generally, existing security mechanisms such as Microsoft's Windows Defender ATP are already able to detect cross-process code injection attempts [19].

Service shutdown: An obvious way for a ransomware to prevent AntiBotics from interfering with its malicious activity is to simply prevent it from running by unloading the PED driver. However, since the PED driver is implemented as a file system filter driver, an attacker cannot unload the driver while the system is running even if it gains admin mode privileges [16].

Path hijacking: as the reader may recall, when the *Execution object type* is set to 'PROG', permits are granted to executable pathnames. In this case, a ransomware may attempt to gain file access by locating an executable that holds an active permit to the file and by overwriting it. The current version of AntiBotics defends against such attacks by deleting all the permits of an executable once it is about to be modified (see lines **16–17** and line **30** of Algorithm 1), so that once the overwriting ransomware attempts to modify/write a file, a new challenge will be presented to it.

Folder exclusion: in order to remove the AntiBotics protection, a ransomware may try to rename an ancestor of a protected folder in order to exclude it from protection. AntiBotics thwarts such attacks by presenting a challenge whenever it identifies a move (rename) attempt whose result is such an exclusion (see Algorithm 3).

Are there additional ways in which future ransomware will be able to attempt circumventing authentication-based file access control if they have to? The answer to this question is most probably positive and so we do not claim that the above list of attacks is exhaustive. Nevertheless, we do estimate that authentication-based file access control holds the potential for significantly raising the bar for future ransomware.

5 RELATED WORK

More than 20 years ago, Young and Yung [30] identified the potential hazards that may result from combining the power of cryptography with malicious code in order to mount extortion based attacks. They analyzed and demonstrated a few such attacks. Unfortunately, their predictions came true. Gröbert et al. [8] presented several automated methods for identifying cryptographic primitives within binary programs. Lestringant et al. [12] proposed an algorithm with the same goal based on data flow graph isomorphism.

Several detection techniques that specifically target ransomware were proposed in recent years. Ahmadian at el. [3] proposed a mechanism for detecting the process of key exchange between the ransomware and the C&C server. However, as observed in [5], not all ransomware require an external key since a ransomware can contain a hardcoded public key which allows it to run offline and encrypt files without communication to a C&C .

Kharraz et al.[10] conducted a long-term study of ransomware attacks and found that a majority of ransomware simply locked the victim's computer without encrypting user files. The prevalence of Cryptolockers within the general population of ransomware rose significantly in recent years, however. They also provide an analysis of ransomware file system activity based on which they propose a general methodology for detection. In later work by Kharraz et at. [10] they presented UNVEIL – a dynamic analysis system for ransomware detection. Their approach is to construct an artificial user environment and detect when ransomware attempts to interact with user data. UNVEIL computes the entropy level of data written to files in that environment, since high entropy may serve as indication of encrypted data.

Scaife at el.[22] propose an early warning system that monitors file changes and attempts to detect several behavioral indicators for ransomware activity. At a high level, these indicators are hints that user files are being transformed from being usable to being unusable. Their primary indicators are file-type change, similarity of file contents before and after change, and entropy levels. Like us, they use the Eldos platform for monitoring application file accesses. Lee et al. [11] propose CloudRPS, a system for monitoring network, file and server activity for ransomware detection. Mbol et al. [14] focus on the detection of ransomware transformations applied to image files using an entropy-based approach.

Similarly to this work, Continella et al. [6] maintain that pure-detection approaches are insufficient for countering ransomware attacks. They propose an approach that combines automatic ransomware detection and transparent file-recovery capability at the file system level. They implemented ShieldFS, an add-on Windows driver that acts as a copy-on-write mechanism, allowing to roll-back all file modifications done by a process if and when it becomes a suspect by deviating from a detection model. Detection models can be generated either per process or per a "meta process", the latter option more appropriate for multi-process ransomware. Both the long- and short-term behavior of processes are monitored. In addition to checking file I/O system calls, ShieldFS scans the memory of running processes in order to detect evidence to the application of cryptographic primitives, which is a strong ransomware detection signal. Their experimental evaluation shows high detection accuracy and low false positive rates.

Shukla et al. [24] discuss some previous proposals for ransomware detection and show that static approaches are insufficient. They propose a dynamic system for mobile ransomware detection that can adapt to changing ransomware attack patterns.

A few works address ransomware attacks against mobile devices. Andronio et al. [4] analyze the characteristics of mobile ransomware families and present the HELDROID detector. It employs static analysis tools for detecting code paths that may indicate attempts to encrypt files on externa media. Taking a very different approach, Mercaldo et al. [15] and Mercaldo et al. [15] employ formal methodologies for detecting ransomware based on binary program static analysis. Song et al. [26] propose an Android ransomware detection algorithm based on monitoring process behavior in terms of CPU, memory, and I/O usage. Maiorca et al. [13] proposed R-PackDroid,

a detector for Android ransomware based on information extracted from system API packages.

Viewed collectively, it seems that all prior proposals for ransomware detection are statistical in nature in the sense that they may incur false positives/negatives. Often they also need to accumulate sufficient data before the ransomware can be identified, which may allow it to encrypt a number of files before it is detected. An exception to the latter limitation is ShieldFS [6], since it allows roll-back of modifications upon detection. To the best of our knowledge, our work is the first to propose using authentication-based file access control for ransomware prevention. Unlike previous suggestions, our proposal prevents ransomware from accessing all protected files in the first place.

6 CONCLUSIONS AND FUTURE WORK

We present AntiBotics, a system that harnesses biometric authentication mechanisms such as fingerprint scanning and human identification schemes such as CAPTCHA in order to prevent ransomware from modifying or deleting files. AntiBotics enforces a file access-control policy by periodically presenting identification/authorization challenges, for ensuring that applications attempting to modify or delete files are invoked by an authorized user rather than by bots.

We implemented AntiBotics for Windows by developing a file system filter driver and conducted a large-scale evaluation on thousands of ransomware samples, verifying that none of them was able to encrypt any of the files protected by it. We then analyzed a few possible ways in which future ransomware may attempt to attack the authentication-based file access control scheme employed by it and discussed how these attacks can be thwarted.

We logged the file-related activity of a population of 36 R&D organization employees over a period of one month. Analysis of the resulting log files shows that an average daily rate of only about 3 challenges per user suffices to authorize the file-modifying applications they use and protect the respective application files against unauthorized use.

While prior proposals for ransomware detection may incur false positives/negatives and often need to accumulate sufficient data before the ransomware can be identified (possibly allowing it to encrypt a number of files before it is detected), authentication-based file access control prevents ransomware from accessing all protected files in the first place. This is not to say that authentication-based file access is a silver bullet for ransomware prevention, as there are most probably additional ways in which future ransomware will attempt circumventing it (if they have to) in addition to the list of attacks we discuss in Section 4. Nevertheless, we estimate that authentication-based file access control holds the potential for significantly raising the bar for future ransomware.

Shortly after implementing AntiBotics and registering a provisional patent[3], Microsoft announced Windows 10 Insider Preview Build 16232, introducing *controlled folders* [17]. Windows Defender allows defining which applications are white-listed for accessing files in a controlled folder. If a non white-listed application attempts accessing a controlled folder, a notification appears and the user must confirm the access. Controlled folders do not support biometric-based authentication and grant white-listed applications unlimited access to controlled folder files. AntiBotics, on the other hand, supports biometric-based authentication and also provides configurable limited-time permits that provide better protection against code injection attacks.

In our current implementation, all newly created permits are set according to the same global *Permit duration* and *Permit scope* configuration parameters, independently of the PID or program to which they are granted. Their granularity is also determined globally according to the system-wide *Execution object type* parameter. Judging from the evaluation results presented by Figure 7, it may be beneficial to support different configuration options for different protected file types, since for some types (such as image files) the rates of challenges for the 'PID' and 'PROG' settings of the execution object type are almost identical, while for others (most notably the .csv type) there is a very large difference. Security-wise, it is better to provide permits to a *specific* process rather than to *all* processes (instantiating some program) if both options yield more-or-less the same level of user disruption. We leave these extensions of AntiBotics for future work.

Acknowledgments: This research was supported by the Cyber Security Research Center at Ben-Gurion University.

[3]Provisional patent number 62/507,245 protecting AntiBotics, was registered on May 17, 2017.

REFERENCES

[1] [n. d.]. *What is a driver?* https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-.
[2] [n. d.]. *What Is a File System Filter Driver?* https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/what-is-a-file-system-filter-driver-.
[3] Mohammad Mehdi Ahmadian, Hamid Reza Shahriari, and Seyed Mohammad Ghaffarian. 2016. Connection Monitor & Connection Breaker: A Novel Approach for Prevention and Detection of High Survivavle Ransomwares. (2016).
[4] NicolÂťo Andronio, Stefano Zanero, and Federico Maggi. 2015. HELDROID: Dissecting and Detecting Mobile Ransomware. *18th International Symposium, RAID* (2015).
[5] Lucian Constantin. 2016. *New Locky ransomware version can operate in offline mode.* Technical Report. http://www.pcworld.com/article/3095865/security/new-locky-ransomware-version-can-operate-in-offline-mode.html.
[6] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. 2016. ShieldFS: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 336–347.
[7] Eldos. [n. d.]. *CallbackFilter.* https://www.eldos.com/cbflt/.
[8] Felix Gröbert, Carsten Willems, and Thorsten Holz. 2011. Automated identification of cryptographic primitives in binary programs. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 41–60.
[9] The Guardian. 2017. NHS seeks to recover from global cyber-attack as security concerns resurface. https://www.theguardian.com/society/2017/may/12/hospitals-across-england-hit-by-large-scale-cyber-attack. (2017).
[10] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. 2015. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 3–24.
[11] Jeong Kyu Lee, Seo Yeon Moon, and Jong Hyuk Park. 2016. CloudRPS: a cloud analysis based enhanced ransomware prevention system. *The Journal of Supercomputing* (2016), 1–20.
[12] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. 2015. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 203–214.
[13] Davide Maiorca, Francesco Mercaldo, Giorgio Giacinto, Corrado Aaron Visaggio, and Fabio Martinelli. 2017. R-PackDroid: API package-based characterization and detection of mobile ransomware. In *Proceedings of the Symposium on Applied Computing*. ACM, 1718–1723.
[14] Faustin Mbol, Jean-Marc Robert, and Alireza Sadighian. 2016. An efficient approach to detect torrentlocker ransomware in computer systems. In *International Conference on Cryptology and Network Security*. Springer, 532–541.

[15] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. 2016. Ransomware steals your phone. Formal methods rescue it. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 212–221.
[16] Microsoft. 2017. Advantages of the Filter Manager Model. https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/advantages-of-the-filter-manager-model. (2017).
[17] Microsoft. 2017. Enable Controlled Folder Access. https://docs.microsoft.com/en-us/windows/threat-protection/windows-defender-exploit-guard/enable-controlled-folders-exploit-guard. (2017).
[18] Microsoft. 2017. File system minifilter driver. https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/file-system-minifilter-drivers. (2017).
[19] Microsoft. 2017. *Uncovering cross-process injection with Windows Defender ATP*. Technical Report. Microsoft, https://blogs.technet.microsoft.com/mmpc/2017/03/08/uncovering-cross-process-injection-with-windows-defender-atp/.
[20] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
[21] Sky News. 2017. NHS cyberattack: List of hospitals hit by ransomware strike. https://www.symantec.com/outbreak/?id=wannacry. (2017).
[22] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin R.B Butler. 2016. CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data. *IEEE 36th International Conference on Distributed Computing Systems*. (2016).
[23] Info security. 2017. Ransomware Cost Businesses $1bn in 2016. https://www.infosecurity-magazine.com/news/ransomware-cost-businesses-1bn-in/. (2017).
[24] Manish Shukla, Sutapa Mondal, and Sachin Lodha. 2016. POSTER: Locally Virtualized Environment for Mitigating Ransomware Threat. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1784–1786.
[25] Hispasec Sistemas. 2004. Virus total. http://www.virustotal.com. (2004).
[26] Sanggeun Song, Bongjoon Kim, and Sangjun Lee. 2016. The Effective Ransomware Prevention Technique Using Process Monitoring on Android Platform. *Mobile Information Systems* 2016 (2016).
[27] Symantec. 2016. *Internet Security Threat Report*. Technical Report. Symantec.
[28] Symantec. 2017. WannaCry ransomware. http://news.sky.com/story/nhs-cyberattack-full-list-of-organisations-affected-so-far-10874493. (2017).
[29] The Cuckoo Sandbox Developers Team. 2017. Cuckoo. https://cuckoosandbox.org/. (2017).
[30] Adam Young and Moti Yung. 1996. Cryptovirology: Extortion-based security threats and countermeasures. In *Proceedings 1996 IEEE Symposium on Security and Privacy*. IEEE, 129–140.