No Random, No Ransom: A Key to Stop Cryptographic Ransomware

Ziya Alper Genç(✉), Gabriele Lenzini, and Peter Y. A. Ryan

Interdisciplinary Centre for Security Reliability and Trust (SnT),
University of Luxembourg, Luxembourg City, Luxembourg
{ziya.genc,gabriele.lenzini,peter.ryan}@uni.lu

Abstract. To be effective, ransomware has to implement strong encryption, and strong encryption in turn requires a good source of random numbers. Without access to true randomness, ransomware relies on the pseudo random number generators that modern Operating Systems make available to applications. With this insight, we propose a strategy to mitigate ransomware attacks that considers pseudo random number generator functions as critical resources, controls accesses on their APIs and stops unauthorized applications that call them. Our strategy, tested against 524 active real-world ransomware samples, stops 94% of them, including WannaCry, Locky, CryptoLocker and CryptoWall. Remarkably, it also nullifies NotPetya, the latest offspring of the family which so far has eluded all defenses.

Keywords: Ransomware · Cryptographic malware · Randomness Mitigation

1 Introduction

Ransomware is a malware, a malicious software that blocks access to victim's data. In contrast to traditional malware, whose break-down is permanent, ransomware's damage is reversible: access to files can be restored on the payment of a ransom, usually a few hundreds US dollars in virtual coins.

Despite being relatively new, this cyber-crime is spreading fast and it is believed to become soon a worldwide pandemic. According to [24], a US Government's white paper dated June 2016, on average more than 4,000 ransomware attacks occurred daily in the USA. This is 300% increase from the previous year and such important increment is probably due to the cyber-crime's solid business model: with a small investment there is a considerable pecuniary gain which, thanks to the virtual currency technology, can be collected reliably and in a way that is not traceable by the authorities.

The cost of ransomware attacks on individuals, enterprises, and societies is huge. It has exceeded $5 billion US dollars in 2017, and estimated to raise to $11.5 billion by 2019 [8]. Attacks like the one perpetrated by WannaCry—which infected 300 thousands computers of, among others, hospital, manufacturing,

C. Giuffrida et al. (Eds.): DIMVA 2018, LNCS 10885, pp. 234–255, 2018.
https://doi.org/10.1007/978-3-319-93411-2_11

banks, and telecommunication companies in about 150 countries—suggests that such predictions are not an exaggeration.

Ransomware applications come in different flavours, but *Cryptographic ransomware*, the family studied in this paper, encrypts a victim's files using *strong cryptography* [5]. In this approach, decrypting without the key is infeasible, so the only hope of recovering the files, in the absence of backups, is to pay the ransom.

In the absence of an effective cure for the threat, official recommendations suggest prevention. The US Government, for instance, suggests "to have appropriate backups, so [...] to restore the data from a known clean backup" and in addition "to verify the integrity of those backups and test the restoration process to ensure it is working" [24]. Keeping backups however is a solution that does not scale if the threat becomes world-wide: it is an expensive practice that not all companies implement whereas private users are likely not to follow the practice at all. Not surprisingly, a survey on the practice [11] reports that only 42% of ransomware victims could fully restore their data.

Security experts have looked into the problem. For example the EUROPOL's European Cybercrime Centre and the Dutch Politie together with Kaspersky Lab and McAfee have founded an initiative called "No More Ransom"[1] whose goal is, we quote, "to disrupt cybercriminal businesses with ransomware connections" and "to help victims of ransomware retrieve their encrypted data without having to pay the criminals". But, in case of infection, the initiative warns that "there is little you can do unless you have a backup or a security software in place". Other professionals are offering applications that are capable of some protection, but these anti-ransomware systems leverage from existing antivirus/antimalware strategies rather than re-thinking afresh how to solve the problem. At the time of writing (May 2018), no silver bullet exists to convincingly contain the threat.

Security researchers have also worked to slow down the threat (see Sect. 8), The have approached the problem from a *cryptographic* perspective, proposing strategies that enable decrypting the files. Since decrypting without the key is computationally hard, those works look for smart ways to place in escrow the encryption keys, and use them later to attempt decrypting the files. Let us call approaches of this nature *i.e.,* that attempt to recover the files after damage is done, "*ex post*", and as "*ex ante*" approaches that attempt to prevent a ransomware from encrypting files in the first place.

Our Contribution. We approach the problem from a cryptographic perspective. Our solution, UShallNotPass, has an *ex ante* nature. It would be the first *cryptographic ex-ante* defense against ransomware that we know about.

Its strategy relies on two fundamental observations. First, the keys-for-money exchange on which ransomware based the success of their business works only if the victim has no other ways to recover the files but paying for the release of the key. To achieve this goal, a ransomware must properly implement *strong cryptography*, which means:

[1] https://www.nomoreransom.org/.

– *robust encryption*, that is, well-established encryption algorithms;
– *strong encryption keys*, that is, long and randomly generated strings.

Second, if these are the tools that ransomware necessarily need, one can try to prevent any unauthorized use of them. Nothing can be done to prevent ransomware from using robust encryption: these algorithms are public and ransomware can implement them directly. Thus, we have to focus on denying access to a source of strong encryption keys.

In Sect. 2 we observe that current ransomware gets random numbers from a small set of functions, called Cryptographically Secure Pseudo Random Number Generator (CSPRNG), that modern Operating Systems (OSes) offer to applications. In Sect. 3, we explain the UShallNotPass's essential idea, which is *to guard access to those functions*, let only the authorized applications (*e.g.,* certified or white-listed) use the resources, and stop all the others. In Sect. 4, we discuss how to implement the access control mechanism and its enforcement strategy, while in Sect. 5, we describe all the technical details of our implementation, discussing how we hook the calls to CSPRNG functions and terminate the caller. In Sect. 6, we benchmark our solution for robustness and performance. We ran our solution against 524 *active* ransomware samples obtained from a pool of 2263 real-world malware samples. We stopped 94% of samples in the test set, which includes, among many others, WannaCry, TeslaCrypt, Locky, CryptoLocker, and CryptoWall. Reverse engineering the remaining 6% samples shows that the samples do actually call CSPRNG; so, with a better implementation we should be able to stop them too. Notably, we stop also Bad Rabbit and NotPetya, which came out after we designed our solution. Because of this, UShallNotPass may then have the potentiality to be effective against zero-day threats.

In Sect. 7 we discuss the limitation of our approach. Although having found one common strategy that is capable to block all the hundreds of instances of real ransomware in our possession is in our opinion a considerable finding, we point out that we have not yet proved that we can stop *only* all current ransomware (*i.e.,* no false positive). This investigation requires a different experimental set up than the one taken in this work and is actually our on-going research. We argue that it should not be hard to upper bound the number of false positives to a reasonable quantity. In Sect. 9 we conclude the work: we critically compare what we think are the novel aspects of our solution against the state of the art, given in Sect. 8, and we try to imagine how future ransomware could overcome our solution.

1.1 Requirements

The requirements that inspired and, a posteriori, characterize the security quality UShallNotPass (we use here the terminology as suggested by the RFC 2119 [4]).

(R1) it MUST stop all currently known ransomware;

(R2) it SHOULD be able stop zero-day ransomware;
(R3) it MUST NOT log cryptographic keys and thus:
  – it should not introduce the risk of single point of failure that smarter ransomware can try to break;
  – it should not endanger the level security of benign applications (*e.g.,* TLS session keys);
(R4) it SHOULD be easily integrated in existing anti-virus software, OSes, and access control tools;
(R5) it MAY be implemented directly in an OS's kernel or in hardware.

2 On Ransomware and Randomness

We answer a fundamental question: why does ransomware need random numbers and from which sources they must necessarily obtain it? The answer will help understand the rationale of USHALLNOTPASS's *modus operandi*.

As any other software virus, a *ransomware*, say R, runs in the victim's computer. On that machine, R finds the files, F, that it will attempt to encrypt. As we saw in Sect. 1, to work properly R needs two tools: a robust encryption algorithm and a means to create strong key, k. With those tools, R has all it needs to encrypt F. The encrypted files will replace F irreversibly and irremediably until the ransom is paid, triggering the release of the decryption key. The diagram in Fig. 1 shows this simple work-flow in picture with some detail that we are going to discuss.

First let us see how R typically acquires the tools it needs. Strong encryption algorithms are publicly available. Current ransomware just makes use of those robust encryption algorithms, either by statically linking third party networking and cryptographic libraries, for example the NaCl or the OpenSSL or by accessing them via the host platform's native Application Programming Interfaces (APIs).

However, to obtain strong keys, R has to access *secure* randomness sources. R has a few alternatives for doing so, but only one is secure. In fact:

(1) R can have a strong k hard-coded, precisely in a section of its binary code, but this solution leaves k exposed. R can probed and have k extracted from it *e.g.,* by Binary Analysis.
(2) R can download a strong k from the Internet. Occasionally ransomware samples employ this technique and download encryption keys from their command and conquer (C&C) servers, but also this option exposes the key. It can be eavesdropped *e.g.,* by Intrusion Prevention System (IPS). Note that although ransomware will likely use secure communication (*i.e.,* an encrypted channel), the problem of establishing the session key remains, looping the argument (*e.g.,* if the key is hard-coded R, there is a way to reverse engineering it, *etc*).

The remaining alternative is to let R generate its own k. But for k to be strong, k must be randomly chosen from a data set with sufficient entropy to

make brute-force attacks infeasible and be kept safe. Where, in a computer, R can find that randomness it requires to build strong keys? True randomness is generally unavailable and thus ransomware must resort to those few deterministic processes that return numbers which exhibit *statistical* randomness. These processes are known as *random number generator (RNG)* functions. R can implement them. But, being deterministic algorithm, RNG are always at risk to be error-prone. If they produce predictable outputs the cryptographic operations build on them cannot be considered secure [9] because with a predictable "randomness" all hybrid encryption schemes would be vulnerable to plain-text recovery [3]. History proves that this concern is legitimate. To give a few examples, in the Debian–OpenSSL incident, random number generator was seeded with an insufficient entropy which resulted generation of easily guessable keys for SSH and SSL/TLS protocols [1]. Moreover, DUAL EC random number generator of Juniper Networks found to be vulnerable, allowing an adversary to decrypt the VPN traffic [2]. These incidents shows that extreme care should be taken when dealing with randomness. *Non-cryptographic* random number sources have weaknesses [22], and they should be avoided in cryptography.

The safest way (*i.e.,* the way to avoid that risk of being error-prone in generating pseudo random numbers) is to use well tested and robust functions called Cryptographically Secure Pseudo Random Number Generator *(CSPRNG)*.

In the OSes of the Microsoft (MS) Windows family, CSPRNG functions are available through dedicated APIs (analogous solutions do exist in other OS families, although the name of the functions will change). User mode applications call cryptographic APIs to get secure random values of desired length. Historically, Windows platform has provided the following APIs:

- `CryptGenRandom`: Appeared first in Windows 95 via MS Cryptographic API (MS CAPI), now deprecated.
- `RtlGenRandom`: Beginning with Windows XP, available under the name `SystemFunction036`.
- `BCryptGenRandom`: Starting with Windows Vista, provided by Cryptography API Next Generation (CNG).

Legacy applications call the function `CryptGenRandom` to obtain a random value or, as modern applications do, call `BCryptGenRandom`. When developers do not need a context, they can also directly call `RtlGenRandom` to generate pseudo-random numbers. Moreover, `CryptGenRandom` internally calls into `RtlGenRandom`. While the implementation of `RtlGenRandom` is not open-sourced, a relevant documentation [12] states that various entropy sources are mixed, including: (i) The current process ID; (ii) The current thread ID; (iii) The ticks since boot; (iv) The current time; (v) Various high-precision performance counters; (vi) An MD4 hash of the user's environment block, which includes username, computer name, and search path; (vii) High-precision internal CPU counters, such as RDTSC, RDMSR, RDPMC; (viii) Other low-level system information[2].

[2] For the complete list, please see Chap. 8 of [12].

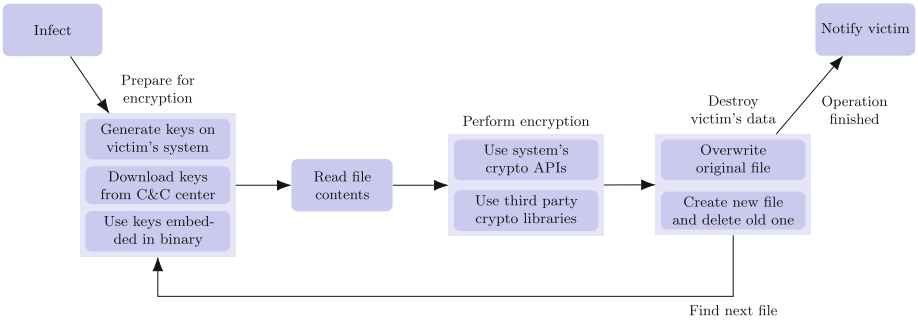

Fig. 1. Generic ransomware functionality.

To be sure the key used is strong, current ransomware takes advantage of the CSPRNG functions that the host OS provides.

Note that, for the same reason, those functions are also used in: (i) Initialization Vectors (IVs): used by both stream and block ciphers (ii) Salts: used in Key Derivation Functions (KDFs) (iii) Paddings: block ciphers (in ECB or CBC modes of operation) and public key encryption algorithms.

3 UShallNotPass' Rationale

From these consideration it should be clear why the idea of this paper of guarding access (*i.e.,* of intercepting incoming calls) to (the APIs of) CSPRNG functions works: any strong ransomware must call those functions sooner or later. All 524 samples we have analyzed call these reliable functions of the OS. Future ransomware might find ways to create keys without calling any CSPRNG function, but that is the topic for future research.

Keys generated by alternative methods may not be so strong and files encrypted with them could be decrypted by *ex-post* defenses like "NoMoreRansom": we plan to test this hypothesis in future work (see Sect. 9).

Thus it should be clear that CSPRNG functions are *security-critical resources*, and hence only authorized processes should have access to them. This means that deciding which processes should be authorized is critical, but is not within the scope of this paper and it will be addressed in future work. Generically speaking, we suggest that authorized applications are those which have been *whitelisted* or *certified*. The process of authorizing an application can be as simple as let the user (or the system administrator) decide about whether s/he trust the application (*e.g.,* as done by [20]), or it can result from an agreement protocol between the operating system kernel's owners (*e.g.,* Microsoft, Apple) and the developers of cryptographic applications, as happens for apps that available in the Apple Store. Whatever the strategy, similarly to what happens in Europe about applications that process personal data, application developers have to gain their authorization/certification. Ransomware, developed for the

illegal software market, should therefore be excluded. The third and last consideration is that we suggest that unauthorized requesters of *CSPRNG functions* are terminated.

Thus UShallNotPass prevents ransomware damaging files in the system and no recovery is necessary. In Sect. 6 we will see how this strategy is essential for stopping Not Petya.

Assumptions. UShallNotPass targets ransomware families that follow secure development strategies and utilize strong cryptography. We will deal only with the strongest amongst current ransomware, that is, we ignore insecurely designed and badly implemented ransomware families, for instance those which call `rand` to generate keys or those which encrypt files with home-brew algorithms. For these ransomware we already have solutions able to mitigate their effects.

Currently, UShallNotPass runs as a software component of the host OS and relies on the security of the host. Therefore we assume that the OS on which our system runs is up-to-date. In particular, we require that ransomware does not exploit any zero-day vulnerabilities to escalate privilege. It should be noted that this requirement is inherent to every defense software runs on any OS. Furthermore, an outstanding feature of our strategy is its being obfuscation agnostic, *i.e.,* UShallNotPass targets all ransomware samples from non-obfuscated to highly-obfuscated ones.

4 UShallNotPass' Design

We now describe the inner mechanism of our technique in more detail.

4.1 High Level Description

Essentially, UShallNotPass is an access control mechanism over the CSPRNG of the host system: it intercepts requests to the CSPRNG and queries the ID of the caller. Once the ID is determined, UShallNotPass reaches a decision according to a *system policy*. If the caller process is authorized, it obtains the pseudo-random number. Otherwise, UShallNotPass takes action according to the system policy. In our implementation, the caller is terminated.

Intercepting Requests to CSPRNG. As we argued in Sect. 2, ransomware requires to use CSPRNG of the host system. In the current architecture of modern OSes, there are limited number of resources which provide cryptographically secure pseudo-random numbers. It is feasible to intercept the calls made to CSPRNG functions of the host system and redirect the control to the decision making component of UShallNotPass.

System Policy and Managing Access Control. When a request is made to access the CSPRNG of the system, to reach a decision to grant or deny access UShallNotPass follows a *system policy*, a set of rules, for instance, determined by the system administrator. The system policy can be specified in various ways, depending on the needs and the nature of the host system. Our current design implements it as a *whitelist*, *i.e.,* list of applications allowed to access CSPRNG which a system administrator determines immediately after UShallNotPass is installed. It can be more complex thought, such as determined by the OS companies in agreement with developers of cryptographic applications and based on accrediations, granted after established security checks.

Further security measures can be necessary. Here we mention two in particular:

- *Digital signatures*: Code signing is a technique to verify the integrity of the executable and the origin of the source. Digitally signed software has therefore higher trust score when evaluated by anti-malware products and OSes. For example, Microsoft uses Authenticode [18] verify the signature of the executables and kernel drivers. Following the same approach, we design UShallNotPass so that it can be configured to allow applications with digital signatures to access to CSPRNG of the host system.
- *Human interaction*: It may be desired to have a minimal whitelist, and extend it when necessary. So that when an application requests a cryptographically secure pseudo-random number for the first time, it is put on hold and the decision will be made on that time. A similar measure has been described in [20], but it involves the user. Considering this choice unsafe, UShallNotPass instead interacts exclusively with the administrator. UShallNotPass's system policy can be set to force to ask the exclusive permission of the system administrator when an application calls CSPRNG for the first time.

Once the whitelist is created, UShallNotPass will start intercepting the access requests to CSPRNG of the host system. For each request, identity of the owner will be determined and UShallNotPass will decide whether to grant access. If the result is positive, the process is allowed to obtain the pseudo-random number. Otherwise, the request is blocked and the process is terminated.

Needless to say, it is therefore of uttermost important to secure the system policy itself from unauthorized modifications (*e.g.,* stored in a directory accessible only with administrator privileges).

5 Implementation

We implemented a prototype of UShallNotPass which targets Windows 7 OS. On Windows 7, user-mode processes invoke `CryptGenRandom` to get cryptographically secure pseudo-random numbers. Therefore, our implementation intercepts each invocation of `CryptGenRandom` API and determines the identity of caller process. To this end, UShallNotPass consists of two components:

- *Interceptor (INT)* which intercepts calls made to `CryptGenRandom` API, collects and transmits the identity of the caller process to controller, and takes the appropriate action that controller commands.
- *Controller (CTR)* which gets information from the Interceptor and returns grant/deny commands according to the system policy.

5.1 Intercepting Calls to CSPRNG

There are various ways of intercepting calls on Windows platform, including patching System Service Dispatch Table (SSDT), modifying Import Address Table (IAT) and injecting a Dynamic Link Library (DLL) to target process. We followed the DLL injection technique and used Detours library of Microsoft Research for this purpose. The Interceptor of USHALLNOTPASS is hence a DLL module which is loaded into target process on the system. For ease of prototyping, we load the Interceptor into processes using AppInit DLLs technique [17]. Once loaded, it hooks `CryptGenRandom` function, that is, whenever `CryptGenRandom` is called by a process, program flow is routed to the Interceptor.

5.2 Decision of Authorization

The Interceptor calls `GetModuleFileName` to obtain the full path to the module of the caller process, which can point to a DLL or an executable. The file path information is passed to the Controller, whose response is forwarded to the Interceptor. Controller computes the SHA256 digest of the binary file of the module and checks whether it is in the whitelist.

If the result is positive, a GRANT command is returned to Interceptor, or a DENY command otherwise. Once the decision is received from Controller, Interceptor executes it. If the decision was to grant access to secure random API, Interceptor calls `CryptGenRandom` with the intercepted parameters and returns the result and control to the caller process. If the decision of Controller was to deny the request, then Interceptor calls `ExitProcess`, which causes the caller process to end[3].

5.3 Maintaining the Whitelist

Whitelist is implemented as a file which contains the list of SHA256 digests of the binary executables. The integrity of the whitelist is protected by a keyed-hash value, appended to the end of the list. As another security precaution, the whitelist is located in a directory which only administrators has write permission.

The Controller component of USHALLNOTPASS has a graphical user interface (GUI) which provides the basic functionality to the user, such as adding an entry to the whitelist or removing one from it. Controller also logs relevant information about the call events to `CryptGenRandom` API, including time, SHA256 digest of the caller and the action taken.

[3] Calling `ExitProcess` can as well cause process to crash, which, eventually ends it.

6 Experimental Results

We tested our UShallNotPass with the aim to verify whether it complied with the requirements we stated in Sect. 1. Compliance with R3 does not need to be tested. It follows from the design: UShallNotPass does not store cryptographic keys (R3). Instead we test compliance with R1 and R2 indirectly by answering the following questions about UShallNotPass:

- **Q1** Does it stop ransomware before they encrypt any files?
- **Q2** Can it protect against zero-day ransomware?

Furthermore we are interested in knowing what is UShallNotPass's performance in time and space resources. A defense system that is not practical to deploy is considered useless.

- **Q3** What is overhead cost in resources of UShallNotPass?

The answer this third question gives evidence for compliance to R4 and R5: if UShallNotPass proves be efficient, it can be easily integrated with existing anti-virus software as an additional run-time control (R4). Its simplicity also suggests that controlling the access to critical functions can be implemented at least at level of OS kernel (R5).

Instead, we have not yet thought about the possibility to implement this mechanism at lower level, such as in hardware.

6.1 Experimental Setup

We conducted a series of experiments to test the robustness of UShallNotPass against cryptographic ransomware. We obtained real world cryptographic ransomware samples from well known sources including VirusTotal[4] and ViruSign[5]. In order to collect executables, we performed a search on these sources with the keywords *ransom, crypt* and *lock* which generally appear in the tags determined by submitters and antivirus vendors. Furthermore, we populated our collection by downloading samples from the links provided by Malc0de[6].

Our initial test set had 2263 malware samples which is labeled by anti-virus engines as ransomware.

Collecting a malware sample is one thing, determining its type is another. A malware sample tagged "ransomware" may not necessarily be an active cryptographic ransomware. Therefore, we needed to check the obtained malware samples one by one and select the active cryptographic ransomware in order to build a valid sample set. For this aim, we utilized Cuckoo Sandbox[7] open source automated malware analysis system. We created a virtual machine (VM)

[4] VirusTotal, https://www.virustotal.com.
[5] ViruSign, https://www.virusign.com.
[6] Malc0de, http://malc0de.com.
[7] Cuckoo Sandbox, https://cuckoosandbox.org.

in KVM[8] and performed a clean install of Windows 7 OS. Next, we created a user environment on the VM and performed actions which reflects the existence of a real user, *e.g.,* we installed various popular applications such as third party web browsers (and select plug-ins), office and document software, utilities etc. Moreover, we placed a number of files on the VM that typical ransomware families targets, such as office documents, images and source codes. When possible, we also removed traces of the virtualization, *e.g.,* changed default device names of VM, tuning RDTSC, etc. Finally, we took the snapshot of the VM and finalized the configuration of Cuckoo for managing the VM.

After the test environment was set, we submitted the malware samples to Cuckoo which executed them one-by-one, on the clean snapshot of the VM. Although majority of ransomware samples attack the system immediately after infection, *i.e.,* encrypts the victim's files, we allowed them to run 20 min unless the detection occurs earlier. After each analysis, we inspected if any alteration/deletion of the decoy files observed on the test machine. We call a malware sample as an active ransomware if any of the decoy files has a new SHA256 hash after the analysis is completed. If Cuckoo does not detect any activity or hashes of decoy files are same until the timeout happens, we exclude the sample from our list of active ransomware.

To compare our results to the previous research, and to reason on the techniques used by malware authors, we identified the family of each ransomware sample. For this purpose, we employed AVclass [21], an automatic malware labeling tool which performs plurality vote on the labels assigned by AV engines.

We excluded the vast majority of the samples from our test set as they did not show up any malicious activity during the analysis. There are several reasons behind this outcome. Firstly, it is a well known fact that malware authors try to avoid being analyzed and thus malware samples behaves benign if they detect that they are run in a virtual environment. Ransomware authors also follow this strategy. Another reason of inactivity is that malware design may involve a C&C server which may be down for some reason. Finally, ransomware may require certain conditions met before start attacking, *e.g.,* regional settings, wait for a specific date.

To sum up, we built a test set which contains 524 active samples from 31 cryptographic ransomware families to test against UShallNotPass.

6.2 Robustness

In this section, we will analyze the outcome of the experiments to find the answer of **Q1** and **Q2**.

To begin with, UShallNotPass stopped ransomware samples from *all families* in our data set, which includes famous and powerful ransomware families The details are reported in Table 1, where we also report for each family the average number of bytes per calls and the numbers of call, figures that support

[8] Kernel-based Virtual Machine, https://www.linux-kvm.org/page/Main_Page.

our argument that employing cryptographically secure pseudo-random numbers is a common property of all the ransomware.

Table 1 shows that UShallNotPass successfully stopped 94% of cryptographic ransomware in our test set, including WannaCry, Locky and TeslaCrypt and remarkably the unmitigated *NotPetya*. The remaining 6% of missed elements looks like be false negative but we have evidence that this is not the case: quite likely we missed them because our implementation of the Interceptor is not perfect. In fact, a dynamic analysis we performed on each representative for all the missed family (*i.e.,* Cryptolocker, Filecryptor, SageCrypt and Yakes) has revealed that the ransomware actually invoke `CryptGenRandom`. Thus, in principle, they should have been stopped. The only conclusion we can draw is therefore that *our* implementation missed to intercept those call for some not obviously apparent technical reason. We looked into that and in Sect. 7 we discuss technical detail about how to improve Interceptor's capacity of intercepting.

That said, we need to comment that UShallNotPass was implemented before the Bad Rabbit and NotPetya ransomware families emerged. Therefore, until proven otherwise, we have at least one evidence that supports R2 that UShallNotPass can be effective on zero-day ransomware.

Case Study: *NotPetya.* We find it remarkable that UShallNotPass was effective against NotPetya, a particular debilitating ransomware that in 2017 was used for a global cyberattack against Ukraine, Germany, Russia, Italy, France and Poland[9]. NotPetya is a ransomware which encrypts victim's disk at boot time (NotPetya has other malware characteristics such as the propagation, exploitation and network behaviors, but those are out of the scope of this paper.) Upon execution, NotPetya generates random numbers to use in the encryption, modifies the Master Boot Record (MBR) of the system disk which allows it to load its own kernel in the next reboot. Next, it restarts the system and shows a fake `chkdsk` screen to the user. Meanwhile, the malicious kernel encrypts the Master File Table (MFT) section of the disk which renders the data on that disk unusable. Since NotPetya loads its own kernel, the solutions proposed by [7,13,14] is bypassed and therefore cannot protect the victim. Moreover, [15] logs the random numbers that NotPetya uses to derive the encryption keys. Nonetheless, the key vault becomes inaccessible as well as other data after the reboot as the MFT is encrypted. On the other hand, UShallNotPass stops NotPetya once it calls `CryptGenRandom` and terminates it before any cryptographic damage occurs.

6.3 Performance

We measured the overhead of UShallNotPass on computing and storage resources to answer **Q3**. Our assessment focuses two points: (i) API level overhead, *i.e.,* the extra time to access secure randomness, (ii) application level

[9] https://en.wikipedia.org/wiki/Petya_(malware).

Table 1. Measurements of CSPRNG usage. Next to *Family*, recalling the ransomware's family name, column *Sample* reports the number of elements in the family and the number of samples that UShallNotPass stopped. *CGR Usage* column shows the need of using CSPRNG among ransomware and contains two subcolumns: *Bytes*, the average number of bytes that a sample of ransomware obtains from calling `CryptGenRandom`, and *#Calls*, the number of calls to the function.

Family	Samples (%)	CGR usage	
		Bytes	# Calls
Androm	7/7 (100%)	4125257	178
Bad Rabbit	1/1 (100%)	52	2
Cayu	1/1 (100%)	4216212	20261
Cerber	149/149 (100%)	22393	2786
Crilock	1/1 (100%)	3456637	15
Critroni	1/1 (100%)	4755304	392
Crowti	3/3 (100%)	5231466	14
Crypmod	1/1 (100%)	2167813	20118
Crypshed	1/1 (100%)	5137296	13
Cryptesla	8/8 (100%)	5125627	14
Cryptolocker	8/17 (47%)	2805603	10
Cryptowall	1/1 (100%)	2242370	10
Dynamer	2/2 (100%)	3954293	20118
Enestaller	3/3 (100%)	2127036	82
Enestedel	5/5 (100%)	3871449	61
Filecryptor	3/4 (75%)	64	1
Genkryptik	3/3 (100%)	2506214	11
Kovter	1/1 (100%)	160	3
Locky	55/55 (100%)	5672894	23940
NotPetya	1/1 (100%)	92	2
Ransomlock	1/1 (100%)	2312373	12
Razy	2/2 (100%)	3955	2851
SageCrypt	4/7 (57%)	3417095	9
Scatter	6/6 (100%)	5626959	560
Shade	2/2 (100%)	2900347	12613
Teslacrypt	82/82 (100%)	4351264	14
Torrentlocker	1/1 (100%)	2642555	388
Troldesh	2/2 (100%)	3500127	11
WannaCry	2/2 (100%)	5615288	162
Yakes	23/39 (59%)	2450372	9
Zerber	115/115 (100%)	5542697	70
Total:	495/524 (94%)		

Table 2. Performance impact of USHALLNOTPASS on 100 000 iterative calls to `CryptGenRandom`

Measurement Mode	Random number length (bits)			
	128	256	1024	2048
USHALLNOTPASS off (seconds)	0.12	0.15	0.20	0.27
USHALLNOTPASS on (seconds)	15.59	15.80	15.84	16.91
Time spent in IPC (seconds)	14.90	15.05	15.05	16.00
IPC discarded (seconds)	0.69	0.75	0.79	0.91
Total overhead (factor)	125.42	105.68	77.69	61.77
IPC discarded overhead (factor)	5.52	5.00	3.89	3.32

overhead, namely, the latency perceived by the users. We conducted the assessments on a Windows 7 OS running on a VM with 2 CPU cores clocked at 2.7 GHz.

Benchmarks in API Level. We measured the time cost of invoking the `CryptGenRandom` API on the clean machine. For this aim, we wrote a benchmark program that invokes `CryptGenRandom` to generate 128 bits of random number, repetitively for 100 000[10] times and outputs the total time spent for this action. We observed that it took 0.12 s to complete this task. Then we run the benchmark program on the system that USHALLNOTPASS runs. This time it took 15.59 s to complete the same task. The results states that USHALLNOTPASS introduces an overhead with a factor of 125. According to our analysis, the main reason behind this impact is the significantly slow communication between Interceptor and Controller components of USHALLNOTPASS. We also observed that, if the overhead of communication is discarded, the performance impact happens to be a factor of 5.52. We remark that the observations made on an unoptimized prototype of USHALLNOTPASS. More efficient techniques of IPC and dynamic decision making for access control would result in better performance figures.

Our measurements on API level overhead and detailed results are illustrated in Table 2. It should be also noted that as the length of the pseudo-random number increases, the cost ratio of access control gets lower.

Impact in Application Level. Another important performance criterion is the slowdown in functionality of the software due to USHALLNOTPASS. On our test system, we installed latests versions of select applications which are common in home and office users. Next, we whitelisted and run the applications while USHALLNOTPASS is active. We inspected whether any slowdown occurred during the use of each application and logged the CSPRNG consumption, if any.

[10] We have chosen to set the limit of trials to 100 000 as with the current implementation of Inter-Process Communication (IPC), our setup becomes instable beyond this limit.

The test set contains the following applications: 7zip, Acrobat Reader, Chrome, Dropbox, Firefox, Foxit Reader, Google Drive, Internet Explorer, LibreOffice, Microsoft Office, Putty, PyCharm, Skype, Slack, Spotify, Teamviewer, Telegram Desktop, TeXstudio, Visual Studio, VLC, WinRar and WinZip. Among those that called `CryptGenRandom`, we present our observations on the following five:

- **Acrobat Reader.** We created a new digital signature and signed a PDF document. During this period, Acrobat Reader called `CryptGenRandom` 13 times and obtained 64 bytes of random value in total.
- **Chrome.** We observed Chrome's CSPRNG usage by connecting a website over HTTPS. For this purpose, we connected https://www.iacr.org/. Once the TLS connection is established, we stopped monitoring. We recorded 2 calls to `CryptGenRandom` and 32 bytes of usage in total.
- **Dropbox.** After creating a new account, we put 5 files with various sizes, 20 MB in total. During the synchronization of these files, Dropbox invoked `CryptGenRandom` 61 times, obtaining 16 bytes of data in each.
- **Skype.** We monitored Skype when making a video call for 60 s. During this period, Skype performed 13 calls to `CryptGenRandom` and obtained 16 bytes in each call.
- **Teamviewer.** Among the tested applications, Teamviewer was the clear winner in pseudo-random number consumption. In our test, we connected to a remote computer and keep the connection open for 60 s. We observed 128 calls to `CryptGenRandom` which yield 2596 bytes in total.

We did not notice any slowdown or loss in the functionality of any applications nor a program instability.

7 Discussion: Limitations and Improvements

History suggests that malware mitigation is a never ending race: a new defense system is responded with new attacks. We are no exception; cyber-criminals will develop new techniques to bypass UShallNotPass. In this section, we first discuss how they could achieve this goal due to the limitations of our approach. Next, we review the issues may arise during the use of UShallNotPass.

7.1 Alternative Randomness Sources

The results of our experiments suggests that cryptographic ransomware can be efficiently mitigated by preventing access to CSPRNG APIs of the host system. Ransomware authors will try to find alternatives sources for randomness. We anticipate that the first place to look for would be the *files* of victims. Generating encryption keys from files is known as *convergent encryption* [10] and already a common practice in cloud computing. That being said, the feasibility and security of maintaining a ransomware campaign (from point of cybercriminals) based on this approach needs to be studied.

Alternatively, ransomware authors may try to fetch cryptographically secure random numbers (or encryption keys) from C&C servers instead of requesting access to CSPRNG API. As we discussed in Sect. 2 ransomware cannot establish a secure channel with the remote server in this scenario. Such a ransomware may still communicate with a randomness source on the Internet, over an unsecure channel. In this case, however, the random numbers would be exposed to the risk of being obtained by IPSes. This would make it difficult for a ransomware to be successfull in the long term. Having said that, more feasible defense strategies should be developed for home users who will likely not be in the possession of advanced network devices like an IPS.

Lastly, ransomware may statically link a random number generator and use a seed gathered from user space. However, this approach would require higher implementation effort and be error-prone. Again, feasibility and security of this risky approach should be studied.

We leave these challenges as open problems for future works.

7.2 Implementation Related Issues

DLL Injection Method. AppInit DLLs mechanism loads the DLL modules specified by the `AppInit_DLLs` value in the Windows Registry. For ease of development, we utilized AppInit DLLs technique to load Interceptor component of UShallNotPass into target processes. However, AppInit DLLs are loaded by using the `LoadLibrary` function during the `DLL_PROCESS_ATTACH` phase of `User32.dll`. Therefore, executables that do not link with `User32.dll` do not load the AppInit DLLs [17]. Concordantly, UShallNotPass cannot intercept and control any calls made from these executables. During the experiments, we encountered 29 ransomware samples that do not link to `User32.dll`. However, dynamic analysis of these samples shows that they all indeed call `CryptGenRandom` function. This finding suggests that more powerful hooking techniques would yield protection against these sample. We highlight that this limitation only concerns our current prototype, *i.e.,* it is not inherent to the approach, and leaves room for improving the implementation of UShallNotPass as a future work.

Whitelisting Built-in Applications. Modern OSes are installed with components including administrative tools and system utilities. Depending on the nature of the tasks, certain built-in applications may utilize the CSPRNG APIs. To keep the OS stable and secure, and maintain its functionality, these applications should be whitelisted before UShallNotPass launched. To determine which built-in Windows applications call CSPRNG APIs, we performed a clean install of Windows 7 32-bit on a VM, monitored the calls to CSPRNG APIs and identified the caller processes. During this experiment, we executed typical maintenance operations on the clean system, such as defragging hard disks, managing backups, installing drivers and updating the OS.

We detected invocation of CSPRNG API by Explorer (`explorer.exe`) and Control Panel (`control.exe`) which are two of the most frequently used Windows applications. Moreover, Windows Update (`wuauclt.exe`) and Windows

Update Setup (`WuSetupV.exe`) are the only signed applications that consumed secure randomness. Therefore, if UShallNotPass is configured to allow the signed applications to access CSPRNG APIs, these two applications do not need to be whitelisted. Furthermore, Local Security Authority Process (`lsass.exe`) was the only application which calls `BCryptGenRandom`, while others called `CryptGenRandom`. The complete list of applications[11] that called CSPRNG APIs during the experiment is given in Table 3.

Table 3. Windows applications that calls CSPRNG APIs. Most of the applications listed below are located at `%WINDIR%\System32`.

Executable name	File description	Digitally signed
explorer.exe	Windows Explorer	✗
lsass.exe	Local Security Authority Process	✗
SearchIndexer.exe	Microsoft Windows Search Indexer	✗
svchost.exe	Host Process for Windows Services	✗
dllhost.exe	COM Surrogate	✗
wmiprvse.exe	WMI Provider Host	✗
SearchFilterHost.exe	Microsoft Windows Search Filter Host	✗
SearchProtocolHost.exe	Microsoft Windows Search Protocol Host	✗
control.exe	Windows Control Panel	✗
TrustedInstaller.exe	Windows Modules Installer	✗
VSSVC.exe	Microsoft Volume Shadow Copy Service	✗
WMIADAP.EXE	WMI Reverse Performance Adapter Maintenance Utility	✗
wuauclt.exe	Windows Update	✓
WuSetupV.exe	Windows Update Setup	✓
mmc.exe	Microsoft Management Console	✗
MpCmdRun.exe	Microsoft Malware Protection Command Line Utility	✗
dfrgui.exe	Microsoft Disk Defragmenter	✗

Handling Sofware Updates. OS software or installed applications may be updated for various reasons, including patching security vulnerabilities, fixing bugs and adding new functionalities. The update process may also involve replacing the existing executables with newer ones and thus altering their hash values. Therefore, if an OS component or an application which has access rights to CSPRNG API is updated, Whitelist of UShallNotPass must also be updated accordingly to prevent false positives. More precisely, the old hash value should be removed from the Whitelist and the new hash value should be added.

[11] The list of applications may vary on different versions of Windows OS.

Abuse of Digital Signatures. While Code Signing aims to help verifying the software origin, cyber criminals frequently used stolen certificates to sign malware in order to penetrate this defense [6,23]. Furthermore, there is an incidence *i.e.,* a ransomware sample with a valid digital signature [25], which proves that ransomware authors also have this capability. Such a clandestine ransomware sample may evade access control feature promised by our system. Namely, if UShallNotPass is configured to allow digitally signed applications to access CSPRNG of the host system, and the ransomware binary has a valid signature (*e.g.,* the stolen certificate is not revoked yet or Certificate Revocation List (CRL) is not up to date), then the victim's files would be encrypted. Note that utilization of digital signatures is optional and meant to improve practicality and applicability of our system. System administrators should decide enabling this feature according to their systems' needs and capabilities. When ultimate security is desired, this option should be left as disabled so that even digitally signed ransomware would not cause harm on data.

User Interaction. As we discussed above, software applications on host system may be updated or replaced with another one. To prevent interruption in the work flow, UShallNotPass may be configured to ask user permission in case previously unseen process requests access to CSPRNG of the host system. This brings the risk of infection, as the user is involved in the decision making, and may not concentrate well each time. We remark that user interaction is an optional feature of UShallNotPass and is an example of security/usability trade off. If disabled, it would not pose any risk against security.

7.3 Improvements

Our prototype currently hooks into only `CryptGenRandom` API, as our initial findings suggested us that it is widely used by ransomware. To evade detection, ransomware may restrict itself to utilize other CSPRNG APIs such as `RtlGenRandom` and `BCryptGenRandom`. However, adding new hooks is only an implementation effort, that we plan to undertake in a future work.

8 Anti-ransomware: A Critical Review

In Sect.1 we distinguished anti-ransomware defenses according to their *ex-ante* or *ex-post* nature. We also separated *non-cryptographic* from non *cryptographic* approach. In addition, there are two main defense strategies which seem driving the most famous works: *behavioural analysis* and *key escrow*.

Behavioral Analysis. Solutions in this sub-category monitor an application's activity in real-time, searching for indicators (*e.g.,* a process' interactions with its environment, file system activity, network connections and modifications on OS components) that may justify counter-actions such as blocking the application's execution. Approaches differ because of what is observed, how the observation process is designed and executed. Thus, Unveil [13] by Kharraz *et al.* generates

Table 4. Comparison of ransomware defense systems

Feature	UNVEIL	CryptoDrop	ShieldFS	PayBreak	Redemption	UShallNotPass
Mode of operation	Proactive	Proactive	Proactive	Key-escrow	Proactive	Access control
Obfuscation resilience	✓	✓	✓	✗	✓	✓
Disk I/O agnostic	✗	✗	✗	✓	✗	✓
Stops NotPetya	✗	✗	✗	✗	✗	✓

an artificial user environment and monitors the potential ransomware there for desktop locks, file access patterns and I/O data entropy. The software decides whether certain activities hide an ransomware by comparing the monitored features with those of benign applications of reference and by applying a similarity threshold of obtained from a precision-recall analysis. Differently, CryptoDrop by Scaife *et al.* [20] operates in the real environment and observes file type changes and measures file modifications. Malicious changes to file are detected by similarity-preserving hash functions and measuring Shannon Entropy. Continella *et al.* developed ShieldFS [7] that monitors low-level file system activities and collects the following features: *folder listing, file read/write/rename, file type* and *write entropy*. A ransomware is recognized by comparing these characteristic activity patterns with that of benign applications. ShieldFS also monitors cryptographic primitives through searching the memory space of a suspicious process for a precomputed key schedule to increase detection speed. Lastly, Kharraz and Kirda developed Redemption [14] which monitors the same indicators as above, but redirects write calls to sparse files. By this way, malicious changes reverted more efficiently than previous defenses.

Works in this category are not cryptographic according to our definition and can have either *ex ante* or *ex post* nature. Which one depends on whether their monitoring happens in a safe virtual environment (so having the possibility to stop the real damage from happening) on in the real system (competing with the ransomware while it has started encrypting).

Key Escrow. Systems adopting this strategy also run in real-time and in the real system so they have an *ex-post* nature. They create the conditions to easy the decryption of the infected files mainly by holding in escrow the encryption keys that the system generates on request. These are many, but in case some requests come from ransomware the keys to decrypt files should be among them. This proactive "protection" is applied only after a ransomware has finished its work.

To the best of our knowledge, the approach of using *key-backup* to combat ransomware is first proposed by Palisse *et al.* in [19] and *independently* by Lee *et al.* in [16]. Later, Kolodenker *et al.* presented the first proof-of-concept of this technique with the PayBreak [15] system. It intercepts calls made to APIs of cryptographic libraries, extracting the parameters in those calls and storing them in a secure key vault. To detect statically linked third-party cryptographic libraries in order to extract encryption keys, the system use fuzzy function signatures. In the case of infection, the system tries to decrypt the encrypted files

using the stored keys and parameters. Since this defense strategy does not involve any file system trace analysis to construct and evaluate the behavior of a process, PayBreak achieves superior performance than the real-time protection systems in the previous category.

Limitations of Current Defenses. To begin with, none of the previous defenses stops NotPetya ransomware. NotPetya performs a disk encryption after the system is booted into its own malicious kernel, thereby bypassing on-line protections.

Besides, solutions that rely on a virtual environment, like Unveil, miss ransomware that recognize the presence of artificial system. Such smart ransomware become malicious only when put in real systems while remaining innocuous and bypassing controls otherwise. Anti-ransomware with an *ex-post* nature, like CryptoDrop, may recognize and stop the ransomware when it is too late. In their experiments over 5100 files, CryptoDrop's authors report that ransomware could encrypt up to 29 files. The median of this statistics reported as 10. Like other behavioural analysis based solutions, ShieldFS comes with an overhead that has been estimated to exceed 40% while being 26% in average. PayBreak, also *ex post*, needs to correctly recognize the cryptographic functions employed by the ransomware to log the encryption keys and the parameters. While this is feasible for built-in cryptographic functions on the host system, ransomware that utilizes third-party libraries can bypass detection through *obfuscation*. In addition, there are some issues with the logging of crypto APIs. PayBreak logs every key, including private keys of TLS and SSH connections. Both protocols offers *forward secrecy* which is build upon employing ephemeral keys. All schemes which counts on application level security (Layer 7 of OSI Model) may become vulnerable in this case. PayBreak is designed in such a way that all keys are stored in one place. This may bring the risk of single point of failure as well as a new target for cyber-criminals. Table 4 compares UShallNotPass against the related works herein commented.

9 Conclusion and Future Work

Cryptographic ransomware applications encrypt files and offer to decrypt them after the payment of a ransom. They are getting better and stronger but need randomness to implement strong encryption. So, a strategy to block them is to control access to randomness sources. We propose UShallNotPass, a system that implements this strategy and terminates unauthorized requests to (CSPRNG)'s APIs provided by the host Operating System. On testing, UShallNotPass stopped 495 active real-world samples of cryptographic ransomware (out of 524, so missing only 6%) from 31 different families. UShallNotPass has minimal overhead on system performance which makes it practical to be used in real-world applications.

There is of course room to extend our approach: to improve the intercept capabilities (for example to confirm our conjecture as to why our implementation missed 6%); improve the performance of our decision making method;

studying and preventing other ways that ransomware could generate encryption keys, circumventing calls to CSPRNG, evading our controls; build a practical and automatic white-listing strategy with low false positive rates (an issue that we have only partially assessed in this paper, since it requires a different experimental set up, and we leave this work for the future).

The approach described here has been shown to be highly effective against the current generation of ransomware, but doubtless, (having read this paper), the authors of ransomware will devise new strategies to evade our approach. The race between ransomware and anti-ransomware will continue.

Acknowledgments. We sincerely thank Clémentine Maurice for reviewing our paper. We also appreciate the anonymous reviewers for their constructive feedbacks and comments. This work is supported by a pEp Security SA/SnT partnership project "Protocols for Privacy Security Analysis".

References

1. Debian Security Advisory: DSA-1571-1 OpenSSL - predictable random number generator, May 2008. http://www.debian.org/security/2008/dsa-1571. Accessed 17 July 2017
2. Juniper Networks: Out of cycle security bulletin, December 2015. https://kb.juniper.net/InfoCenter/index?page=content&id=JSA10713. Accessed 17 July 2017
3. Bellare, M., Brakerski, Z., Naor, M., Ristenpart, T., Segev, G., Shacham, H., Yilek, S.: Hedged public-key encryption: how to protect against bad randomness. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 232–249. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10366-7_14
4. Bradner, S.: Key words for use in RFCs to Indicate Requirement Levels. BCP 14, RFC Editor, March 1997. http://www.rfc-editor.org/rfc/rfc2119.txt, http://www.rfc-editor.org/rfc/rfc2119.txt
5. Bromium: Understanding Crypto-Ransomware (2015). https://www.bromium.com/sites/default/files/rpt-bromium-crypto-ransomware-us-en.pdf
6. Chen, T.M., Abu-Nimeh, S.: Lessons from stuxnet. Computer **44**(4), 91–93 (2011)
7. Continella, A., Guagnelli, A., Zingaro, G., De Pasquale, G., Barenghi, A., Zanero, S., Maggi, F.: ShieldFS: a self-healing, ransomware-aware filesystem. In: Proceedings of the 32Nd Annual Conference on Computer Security Applications, pp. 336–347. ACSAC 2016. ACM, New York (2016)
8. Cybersecurity Ventures: Ransomware Damage Report (2017). https://cybersecurityventures.com/ransomware-damage-report-2017-part-2/
9. Dodis, Y., Ong, S.J., Prabhakaran, M., Sahai, A.: On the (im)possibility of cryptography with imperfect randomness. In: 45th Annual IEEE Symposium on Foundations of Computer Science, pp. 196–205, October 2004
10. Douceur, J.R., Adya, A., Bolosky, W.J., Simon, D., Theimer, M.: Reclaiming space from duplicate files in a serverless distributed file system. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002), pp. 617. ICDCS 2002. IEEE Computer Society, Washington, DC, USA (2002)
11. Gammons, B.: 4 Surprising Backup Failure Statistics that Justify Additional Protection, January 2017. https://blog.barkly.com/backup-failure-statistics. Accessed 17 July 2017

12. Howard, M., Le Blanc, D.: Writing Secure Code. Developer Best Practices, 2nd edn. Microsoft Press, Cambridge (2004)
13. Kharaz, A., Arshad, S., Mulliner, C., Robertson, W., Kirda, E.: Unveil: a large-scale, automated approach to detecting ransomware. In: 25th USENIX Security Symposium (USENIX Security 2016), pp. 757–772. USENIX Association, Austin, TX (2016)
14. Kharraz, A., Kirda, E.: Redemption: real-time protection against ransomware at end-hosts. In: Dacier, M., Bailey, M., Polychronakis, M., Antonakakis, M. (eds.) Research in Attacks, Intrusions, and Defenses, pp. 98–119. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66332-6_5
15. Kolodenker, E., Koch, W., Stringhini, G., Egele, M.: Paybreak: defense against cryptographic ransomware. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, pp. 599–611. ASIA CCS 2017. ACM, New York (2017)
16. Lee, K., Oh, I., Yim, K.: Ransomware-prevention technique using key backup. In: Jung, J.J., Kim, P. (eds.) Big Data Technologies and Applications, vol. 194, pp. 105–114. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-58967-1_12
17. Microsoft: Working with the AppInit_DLLs registry value, November 2006. https://support.microsoft.com/en-us/help/197571/working-with-theappinit-dlls-registry-value
18. Microsoft Corporation: Windows Authenticode Portable Executable Signature Format. Technical report, March 2008. http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx
19. Palisse, A., Le Bouder, H., Lanet, J.-L., Le Guernic, C., Legay, A.: Ransomware and the legacy Crypto API. In: Cuppens, F., Cuppens, N., Lanet, J.-L., Legay, A. (eds.) CRiSIS 2016. LNCS, vol. 10158, pp. 11–28. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54876-0_2
20. Scaife, N., Carter, H., Traynor, P., Butler, K.R.B.: Cryptolock (and drop it): stopping ransomware attacks on user data. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), pp. 303–312, June 2016
21. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: AVclass: a tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 230–253. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_11
22. Soeder, D., Abad, C., Acevedo, G.: Black-box assessment of pseudorandom algorithms. Black Hat USA (2013). https://media.blackhat.com/us-13/US-13-Soeder-Black-Box-Assessment-of-Pseudorandom-Algorithms-WP.pdf
23. Szor, P.: Duqu-Threat Research and Analysis, November 2011. https://securingtomorrow.mcafee.com/wp-content/uploads/2011/10/Duqu.pdf
24. US Department of Justice: How to Protect your Networks from Ransomware (2016). https://www.justice.gov/criminal-ccips/file/872771/download
25. VirusTotal: Scan report, June 2017. https://virustotal.com/en/file/81fdbf04f3d0d9a85e0fbb092e257a2dda14c5d783f1c8bf3bc41038e0a78688/analysis/