NoCry: No More Secure Encryption Keys for Cryptographic Ransomware

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

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

Abstract. Since the appearance of ransomware in the cyber crime scene, researchers and anti-malware companies have been offering solutions to mitigate the threat. Anti-malware solutions differ on the specific strategy they implement, and all have pros and cons. However, three requirements concern them all: their implementation must be secure, be effective, and be efficient. Recently, Genç *et al.* proposed to stop a specific class of ransomware, the cryptographically strong one, by blocking *unauthorized* calls to cryptographically secure pseudo-random number generators, which are required to build strong encryption keys. Here, in adherence to the requirements, we discuss an implementation of that solution that is more secure (with components that are not vulnerable to known attacks), more effective (with less false negatives in the class of ransomware addressed) and more efficient (with minimal false positive rate and negligible overhead) than the original, bringing its security and technological readiness to a higher level.

Keywords: Ransomware · Malware · Cryptovirus · CSPRNG

1 Introduction

Cryptographic ransomware reached the peak of its fame after `WannaCry`'s worldwide attack, in May 2017. On victim's machine, it encrypts files, asking for a ransom (hence the name) to release the cryptographic key the victim needs to decrypt the files and re-access them. Unsurprisingly, according to a recent survey [6], 50.6% of the victims did not get any key in return, after the payment, irremediably losing the data and money.

Encryption is a strong instrument in the hands of criminals. If properly implemented, its impact is irreversible: without knowing the decryption key, recovering the contents of an encrypted file is computationally unfeasible, a very disruptive fact for the victims. However, implementing cryptography flawlessly is a difficult task, and coders of ransomware are challenged by the same issues that have been troubling security engineers in charge of implementing cryptographic applications. One of the most relevant is to generate (cryptographically secure) encryption keys and keep them safe. Failing in this makes the encryption weak

in the sense that it becomes likely to reproduce or retrieve the decryption keys, which would jeopardize the ransomware business model. In this issue, there is hope as same defence and some anti-ransomware solutions (see §6) indeed offer to recover files counting on ransomware engineering's being naïve in implementing strong cryptography.

Unfortunately, modern ransomware programs are coded more professionally than those in the past. Such professional variants (and attack relying on them are increasing and demanding higher ransom, contrarily the general trend that sees the number of ransomware attacks dropping, see [7]) are quite sophisticated, well designed, and properly implemented. Among them, there are variants of `WannaCry`, and variants of other ransomware families such as `Petya`, `NotPetya`, and `GoldenEye`, `CryptoLocker`, `Crysis`, `Cerber`, and `RAA`. They all pose serious threats. Bajpai *et al.* [2], who propose for ransomware a scale similar to the the Saffir-Simpson for hurricanes, classify them as having severity categories 5 and 6.

Are then they unstoppable? Genç *et al.* discuss a strategy in [10], called UShallNotPass. The core idea is to impede them to call *cryptographically secure pseudo-random number generators* (CSPRNGs). These functions offered by the operating system return the essential ingredients required to build cryptographically secure encryption keys: "good" pseudo-random numbers. The solution described in [10] has a sufficiently accurate detection rate (*i.e.,* 94%), but it is not yet an effective and efficient solution. What it needs is an access control system that guarantees at least three important requirements: (1) to rely on architectural components that are not vulnerable against known or arguable targeted attacks; (2) to have lower false positive rate; (3) to impose a negligible performance overhead.

Contribution. We discuss improvements to the solution proposed in [10] that satisfies requirements (1)–(3). It meets (1) by avoiding *interprocess communication* (IPC), a choice that is potentially vulnerable to named pipes hijacking (§4.1). It meets (2) by bootstrapping and maintaining a *Whitelist DB* of honest applications that also call CSPRNG (§4.2). It meets (3) by showing that, when run in respect to vanilla system, our implementation has a negligible overhead (§5) over applications that use CSPRNGs, with a relative improvement of roughly two orders of magnitude with respect the prototype presented in [10]. We also re-test the implementation against 747 active real-world ransomware samples, and measure the false negative rate.

To appreciate fully this paper's contribution, we recall UShallNotPass's in §2, security model and assumptions in §3, and the state of the art in anti-ransomware in §6. We discuss and test our implementation in §4 and in §5, arguing that our version of UShallNotPass, which we call NoCry, in antithesis to the infamous `WannaCry`, has potential to become the best defense against ransomware at the time of writing (June 2019).

2 Recalling UShallNotPass: No Random, No Ransom

UShallNotPass [10] has been proposed as a solution to stop cryptographically strong ransomware attacks. It intercepts calls made to *application programming interfaces* (APIs) of *cryptographically secure pseudo-random number generators* (CSPRNGs) and allows only authorized applications to get through, blocking and terminating all the others.

On modern *operating systems* (OSs), CSPRNG APIs are the only reliable source of cryptographically secure pseudo-random numbers that are necessary to build (cryptographically strong) encryption keys, which are the instruments that a crypto-ransomware needs to be certain that it is unfeasible for a victim to reverse the damage without paying the ransom.

Genç *et al.* showed that a proof-of-concept implementation, proving they are able to neutralize even `NotPetya`, and collect evidence that the concept works against a very large class of about five hundreds real-word active cryptographically strong ransomware samples including `WannaCry`, and other ransomware families such as `Petya` and `GoldenEye`, `CryptoLocker`, `Crysis`, `Cerber`, and `RAA`.

The goal of [10] is to prove that by controlling access to CSPRNG, ransomware can be blocked before any damage occurs. However, how the authorization is decided has not been detailed, but claimed relying on a *Whitelist* database (DB) accessible only with admin privileges and upon an undefined security policy. It suggests however two optional mechanisms for authorization: (i) digitally signed executables can call CSPRNG; and (ii) not digitally signed executables can also call CSPRNG, if the administrator decides so at run time.

The architecture of UShallNotPass and the workflow is depicted in Fig. 1. It has two separate components: Interceptor, and Controller. Interceptor captures the calls made to `CryptGenRandom` API (a CSPRNG offered by Windows OS) and dispatches the process ID to the Controller, which searches the Whitelist DB to decide whether to allow or deny access. No parameters or outputs are logged. The overhead which the proof-of-concept prototype of UShallNotPass brings to the clean system is significant. Details and benchmarks can be found in [10].

3 Security Assumptions

UShallNotPass [10] works under two assumptions, which remain valid in our implementation of the concept, NoCry: (i) at the moment in which the anti-ransomware is installed on a target system and before it becomes active and operational, the system is non-compromised; (ii) the host machine can run anti-virus software to detect, stop and neutralize common malicious actions such as keystroke logging, process injection, etc.

We also stress one key point once more. The original concept, and thus NoCry, has been conceived to work against cryptographically strong ransomware *only*. At least in the ransomware samples that we have analyzed, those are the ransomware programs that access secure random number sources. NoCry does not stop ransomware that does not follow secure development standards and, for

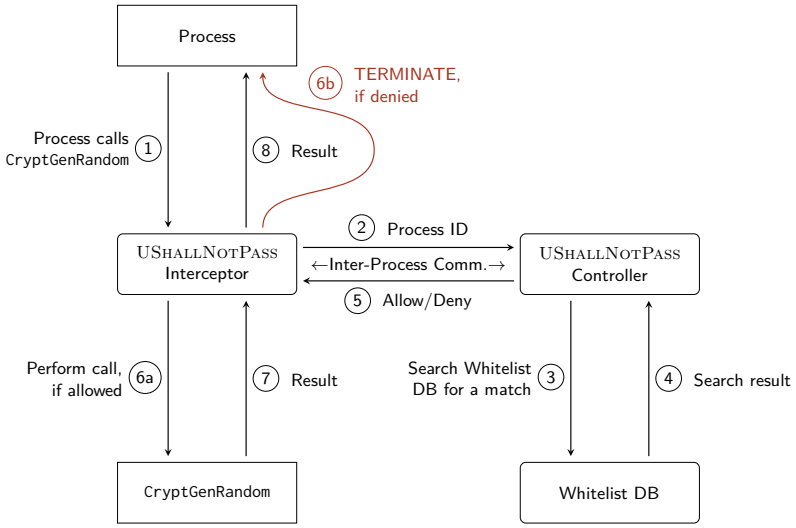

Fig. 1. Architectural view of USHALLNOTPASS [10]. When `CryptGenRandom` API is called, Interceptor identifies the caller and dispatches the process ID to Controller. If the application is authorized, the call is executed and the result is returned to the caller. Otherwise, the call is blocked and the caller process is terminated.

instance, derives keys from a *non-cryptographic pseudo-random number generators* (PRNGs), like `rand` function in `C` runtime library or `System.Random` class provided by .NET framework. In §5.3, we argue that such ransomware variants are weak, cannot achieve success in the long term, or can be stopped otherwise. Therefore, NOCRY is not all-in-one defence but meant to work side-by-side with (or even, integrated into) traditional anti-malware solutions or in combination with other anti-ransomware systems.

4 NOCRY: Requirements, Design and Implementation

We believe that an anti-ransomware application should be effective and non-invasive in at least the following meanings:

Robust Architecture. The execution and operation of the defense system should rely on architectural choices that minimize the attack surface and have no vulnerabilities against known and arguable targeted attacks. In our case, the authorization mechanism be robust against targeted attacks.

Low False Positive Rate and Minimal User Intervention. While providing the security, the defense system must also ensure (arguably and measurably) a low rate of false positive. The challenge regards our Whitelist DB. The list needs to be safely bootstrapped, and software updates should be reflected in the Whitelist DB with no interruption, inconsistency, or possibility of intrusions.

Optimized Decision Procedure. The performance impact of running an anti-ransomware should be negligible and must be imperceptible by the user. In NoCry, the overhead is due to the interception of calls to CSPRNG APIs and the time required by the access control decision procedure.

We discuss the NoCry in the reminder of the section. We refer to Windows systems, as they have been the target of most of the ransomware attacks known at today. What we discuss applies to other platforms as well.

4.1 Robust Architecture

As described in §2, UShallNotPass consists of two components: Interceptor detects the calls made to CSPRNG APIs and Controller makes authorization decisions for the caller processes. This architecture needs an active communication channel between Interceptor and Controller components. In order to fulfill this need, UShallNotPass employs *named pipes*.

A named pipe is an *interprocess communication* (IPC) mechanism which enables processes to communicate to each other using a client-server architecture [17]. In this model, the *pipe server* is the application which creates the named pipe. Once the pipe is created, *pipe clients* – the applications that connects to the pipe server – can start sending/receiving messages to/from the pipe server. In the access control system of UShallNotPass, Interceptor creates two simplex named pipes, one for dispatching the process ID to Controller and another for getting the authorization result.

That said, named pipes in Windows platform are infamous with their security issues [3]. Among them, one particular issue constitutes a critical vulnerability for UShallNotPass. Namely, a malicious application can attempt to create a named pipe before the legitimate application does, and act like the pipe server. The pipe name of UShallNotPass is static and therefore a ransomware can hijack the pipe by creating the pipe instance more quickly than Controller of UShallNotPass. This would make the attacker owner of the named pipe object, allowing the ransomware to impersonate the Controller and authorize itself.

Observing this vulnerability, NoCry is designed to be IPC-free. In this new architecture, Interceptor and Controller are moved into Unified Agent, a single module which intercepts and controls CSPRNG calls. The architectural view of NoCry is illustrated in Fig. 2. The capability of direct data exchange between Interceptor and Controller renders NoCry immune to the potential targeted attacks. Consequently, we conclude that NoCry is a more robust protection system.

4.2 Low False Positive Rate and Minimal User Intervention

We introduce two methods that NoCry offers in order to increase the usability.

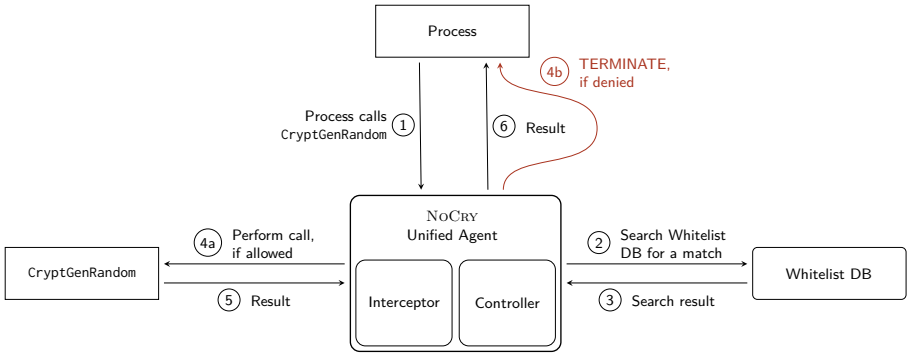

Fig. 2. Architectural view of NoCry. Interceptor and Controller reside in the same module, Unified Agent. This new construction enables robust and efficient information exchange between Interceptor and Controller for making an authorization decision.

Bootstrapping Whitelist DB UShallNotPass does not come with a predetermined whitelist of benign applications. The list, presumably, is initially empty and if access control over CSPRNG APIs were applied immediately after UShallNotPass is installed, every cryptographic application invoking these functions would be stopped: this is surely not what the authors mean to happen. Thus, benign cryptographic applications should be whitelisted before UShallNotPass is launched. To make this task as much automatic as possible we suggest in NoCry a *Training Mode*. It starts immediately after installation: the Interceptor listens the calls made to CSPRNG APIs without blocking any. Under our assumptions (§3), all access requests to CSPRNG APIs should come from honest processes. The hash of the binary executables are added to the Whitelist DB. Training Mode can only be activated once and just after the setup.

What if, against our assumption, Training Mode is run on a system that is infected by some strains of silent ransomware [13]? Some strains in fact infect computers but stay inactive until being activated by *command and conquer* (C&C) servers or simply await until a certain time has passed. This way, ransomware attempts to look like a benign application and evade behavioral analysis-based detection systems. It is unlikely that such ransomware bypass NoCry: the ransomware executable would not call CSPRNG APIs in the sleeping phase and therefore they will not be whitelisted, unless the training phase coincides with the awakening of ransomware. This may be a remote possibility, but raises our assumption of making mandatory running our Training Mode in a clean system a must, as it is usually the case for any anti-malware.

Handling Software Updates Whitelist DB can change. Programs that access CSPRNG APIs but are installed after the Training Mode has ended, must have their hashes be added to it. OS components are updated for various reasons, including patching security vulnerabilities, fixing bugs and adding new functionalities and since the update process involves replacing the existing executables with

new ones, their hash values in the Whitelist DB have to be updated consequently. User applications also regularly check for new updates and install them in the background. The hashes of these updated executables should also be reflected to the Whitelist DB.

In environments where this could potentially lead to delays, *e.g.,* due to slow human reaction, we suggest that NoCry can be configured to defer access control to keep the system stable and workflow uninterrupted. We call this *Deferred Mode*.

When working in Deferred Mode, NoCry does not immediately block calls to CSPRNG APIs coming from unknown processes. Instead, the parameters and outputs of these calls are securely logged in a protected location until administrator takes an action. Here, administrator can find the software benign, thus add the hash of the executable to Whitelist DB and dispose the logs associated with that process. Otherwise, the process is suspended and, if necessary, recovery procedure is initiated. The logging, and when necessary, recovery procedures are similar to the approach of PayBreak [14] which we discuss in §6. However, there are two notable differences in NoCry:

(i) logging is applied per unidentified process, not system-wide; and
(ii) once the administrator makes a positive decision, the logs are disposed.

The rationale of the variations above is to reduce the potential impact of logging the outputs of CSPRNG. In our approach, random numbers obtained by whitelisted processes are not logged. This eliminates the security risks which could arise due to the persistence of the generated random numbers which are potentially used for cryptographic purposes.

4.3 Optimized Decision Procedure

In UShallNotPass, the access control over CSPRNG APIs requires to make an authorization decision which cause a significant delay. Mainly, the delay is due to two factors: (i) time spent for establishing IPC; and (ii) time spent by Controller for authorization.

As discussed in [10], the IPC is the main bottleneck of the authorization procedure and causes an overhead on CSPRNG APIs calls with a factor ranging from 62 to 125. In addition to the improved the security, eliminating the IPC from access control system is another motive which led us to unify Controller and Interceptor in a new module Unified Agent in NoCry. This way, both interception and authorization tasks are carried out in one place, without needing to consume time for IPC which enables to decide and act faster.

Furthermore, in UShallNotPass, the subsequent calls from the same process are authorized independently. While this approach would provide the highest level of time-granularity in access control, it might be an overkill for the security goals and a waste of resources for many systems. It is reported in [10] that the security checks performed in Controller causes an overhead up to a factor of 5.52. NoCry, therefore, holds an authorization to be valid for the lifetime of a process.

It is reasonable to expect that the two optimizations above would bring a significant performance improvement, which we assess in the next section.

5 Methods, Experiments and Results

On NoCry, we have run a series of experiments aiming at to measure the performance overhead, and *false positive* & *false negative* rates. For each experiment we describe the methodology, then we report and discuss the result.

5.1 Performance

Methodology. We measure the time that a benchmark program spends invoking `CryptGenRandom` API repetitively for 100 000 times. We run the benchmark program first on a clean system, then on a system with NoCry. We made this experiment on Windows 7 32-bit OS, running on a VM with 2 CPU cores clocked at 2.7 GHz. Overall, this is the same setting used in [10].

Results and Discussion. Table 1 shows the results of our measurements. It also reports the result from [10], obtained using the exact same methodology.

Table 1. Time benchmarks of 100 000 iterative calls to `CryptGenRandom` API. Performance gain is calculated as $(old - new) \setminus old \times 100$. Measurements of UShallNotPass are recalculated.

Measurement Mode	Random Number Length (bits)			
	128	256	1024	2048
Clean System (sec)	0.13	0.14	0.18	0.24
UShallNotPass (sec)	15.59	15.80	15.84	16.91
UShallNotPass Overhead	11992%	11285%	8800%	7024%
NoCry (sec)	0.17	0.18	0.22	0.29
NoCry Overhead	30%	22%	18%	20%
Performance Gain	**98.9×**	**98.9×**	**98.6×**	**98.3×**

Our analysis shows that NoCry brings drastically lower overhead in terms of time for getting the output of `CryptGenRandom` API. This improvement is due to the unification of Interceptor and Controller components of UShallNotPass which enables interception and control actions to be managed by a single component, Unified Agent, and thereby removing IPC. This result is not surprising after our improvements in §4.3 and confirms our hypothesis.

Another cause of the performance increase is the use of cache mechanism during authorization. In UShallNotPass, iterative calls from the same process are authorized individually, causing a significant overhead, as much as a factor

of 5.52 [10]. With NoCry, process authorizations are valid for the lifetime of a process. That is, accessing to the Whitelist DB is performed once after the first invocation of CSPRNG API. This allows eliminating the need for accessing the Whitelist DB for authorizing subsequent calls.

Lastly, the architecture of UShallNotPass limited the maximum number of iterative calls to `CryptGenRandom` API to the order of 100 000 as the system becomes unstable beyond this point [10]. Since NoCry is IPC-free, it was able to handle a significantly larger number of requests. This makes it a better candidate for a protection system where CSPRNGs are heavily consumed.

5.2 Evaluation of False Positives

In the domain of NoCry, *false positive* describes the condition that a legitimate process calls a CSPRNG API and is stopped by NoCry.

Methodology. We have collected the Top 20 Installed Programs according to Avast PC Trends Report 2019 [1], and we look at whether they have digital signatures, the criterion which NoCry can use for authorization.

Results and Discussion. Table 2 presents the results of our findings. Among the Top 20, the only unsigned application is `7-Zip`. Being `7-Zip` is an open source software, system administrators can obtain the source code, compile themselves, and add it to the NoCry whitelist.

It is reasonable to expect that digital signatures of applications and source code availability of open source software together help system administrators maintain Whitelist DB and therefore lower the number of false positives. In the lights of these circumstances, we perceive that the false positive rate of NoCry will be at a non-invasive level.

5.3 Evaluation of False Negatives

Modern ransomware employs hybrid cryptosystems for scalability and efficiency reasons. Consequently, managing the encryption keys in a secure manner is critical for a successful ransomware campaign, as a flaw in the transport, usage or storage of the keys might allow security professionals to build a decryptor. In particular, if the victims can obtain the keys used to encrypt files, decrypting the files without paying a ransom would be feasible. This is obviously against the goals of ransomware authors so they try to obtain encryption keys securely. The analyses in previous works [2,9] recognizes the following three strategies to obtain the encryption keys: (i) using embedded keys in the binary file; (ii) generating keys on the victim's machine; and (iii) downloading keys from a certain network location.

The security analyses of key generation in ransomware are found in [2,9]. Here, we resume it. If a ransomware follows (i), keys can be extracted from the ransomware binary, and the encrypted files can be recovered. Most of the ransomware prefer to generate the keys on victim's machine. In this case, there

Table 2. Top 20 Installed Programs according to [1]. All applications in the table calls one or more CSPRNG APIs. NoCry will allow these calls automatically since the applications are digitally signed, except for 7-Zip, which is an open source software.

Rank	Program	Calls CSPRNG APIs	Digitally Signed	Source Code Open
1	Google Chrome	✓	✓	
2	Acrobat Reader	✓	✓	
3	WinRAR	✓	✓	
4	MS Office	✓	✓	
5	Mozilla Firefox	✓	✓	✓
6	VLC Media Player	✓	✓	✓
7	Skype	✓	✓	
8	CCleaner	✓	✓	
9	iTunes	✓	✓	
10	TeamViewer	✓	✓	
11	Windows Live Essentials	✓	✓	
12	7-Zip	✓		✓
13	Stream	✓	✓	
14	Dropbox	✓	✓	
15	Opera	✓	✓	
16	CyberLink PowerDVD	✓	✓	
17	CyberLink PowerDirector	✓	✓	
18	HP Photo Creations	✓	✓	
19	CyberLink YouCam	✓	✓	
20	CyberLink Power2Go	✓	✓	

are two options: to use the CSPRNG, which produces high entropy random values; or to use a non-cryptographic PRNG. The first has been largely discussed already. The second is a weak choice: PRNGs are designed to be reproducible thus their outputs are guessable. If the ransomware uses a non-cryptographic PRNG, like `rand` function in `C` runtime library or `System.Random` class provided by .NET framework, decryption is feasible. If ransomware fetches keys from a remote server (iii) then blocking the malicious IPs inhibits the ransomware, which forces ransomware developers to fallback to (i) or (ii). The only option for current ransomware to get good encryption keys is therefore to use a CSPRNG.

In order to support our argument that CSPRNG is vital for the success of a ransomware, we designed experiments: the first, `(A1)`, aims at to find out how common it is to use CSPRNG among current ransomware families. Indirectly, we also measure the false negative rate of NoCry. The second, `(A2)`, aims at to check if there exists a publicly available decryptor for those samples that did not call any CSPRNG APIs.

Methodology. Following the previous research [10], we (1) obtain malware corpus from VirusTotal[1]; (2) pick potential ransomware among them; (3) rebuilt the

[1] VirusTotal Threat Intelligence, https://virustotal.com.

same test environment, using Cuckoo Sandbox[2] to identify the active ransomware samples; and, (4) classify the families using AVCLASS [22] tool; (5) run NOCRY against them; (6) (if any) discover the reason for false negative.

Results and Discussion. We identified 747 active samples from 56 cryptographic ransomware families. Next, we installed NOCRY on the test machines and run the executables against NOCRY. Table 3 shows the results: 97.1% of the samples have been stopped by NOCRY before any user file is damaged, *i.e.,* encrypted by the ransomware program. They were the samples that attempted to call CSPRNG during the attacks, and were terminated by NOCRY as they were not present in the Whitelist DB.

Among the 2.9% of samples that cause false negative, there may be ransomware executables that either circumvented NOCRY's access control, or ransomware process did not call CSPRNG APIs. To discover the exact reason behind the false negatives, we picked random samples from the families we missed, and manually analyzed the API call tree. The missing samples from `Cryptxxx` and `Dalexis` did call `CryptGenRandom` API, however, said API could not be hooked by NOCRY. We believe this is due to a problem of our implementation. The missing samples from `Carberp`, `Cryakl`, `Crysis`, `Gator`, `Neoreklami` and `Sigma` families did not call any CSPRNG APIs. Among them, we found decryptors for `Cryakl`, `Crysis` and `Sigma` on ID Ransomware[3] platform.

6 State of the Art in Ransomware Defense

There have been several proposals from the community of information security to mitigate the cryptographic ransomware threat. NOCRY falls into the *access control* class, as Genç *et al.*'s USHALLNOTPASS [10]. We can categorize other defense systems, based on their main strategies, into three groups: *behavioral analysis*, *key escrow* and *deceptive protection*.

Behavioral Analysis. A common anti-malware strategy is to monitor the processes and terminate the ones with a suspicious behavior. The monitored behaviors include file system I/O, network connections and interaction with the OS. Among these, the fundamental characteristic of the ransomware is its aggressively encrypting victim's data, causing an unusual file system activity. Using this fact, several defense systems are proposed. One of them, Scaife *et al.*'s CRYPTODROP [21] monitors file type changes by looking file headers, compares `sdhash` [20] outputs and measures the Shannon Entropy before and after file-write operations. Another one, SHIELDFS [4] by Continella *et al.* tracks the low-level file system operations and collects the following features: folder listing, file-read/write/rename operations, file extension and average entropy of file-write operations. Comparing these characteristics with that of benign applications allows the detection of ransomware. In addition to detection, SHIELDFS creates a copy for each file

[2] Cuckoo Sandbox – Automated Malware Analysis, https://cuckoosandbox.org/.
[3] ID Ransomware, https://id-ransomware.malwarehunterteam.com/.

Table 3. List of active ransomware samples tested against NoCry. The notation x/y means that *x samples out of y could be successfully stopped.*

Family	Samples (%)
Barys	1/1 (100%)
Birele	1/1 (100%)
Bitman	152/152 (100%)
Browserio	2/2 (100%)
Bzub	1/1 (100%)
Carberp	0/1 (0%)
Cerber	60/60 (100%)
Cryakl	0/1 (0%)
Cryptxxx	0/2 (0%)
Crysis	2/3 (66%)
Dalexis	1/3 (33%)
Daws	5/5 (100%)
Delete	1/1 (100%)
Deshacop	1/1 (100%)
Dlhelper	1/1 (100%)
Enestaller	1/1 (100%)
Enestedel	1/1 (100%)
Expiro	1/1 (100%)
Gamarue	2/2 (100%)
GandCrab	1/1 (100%)
Gator	1/2 (50%)
GlobeImposter	1/1 (100%)
Godzilla	1/1 (100%)
Jaff	1/1 (100%)
Lethic	4/4 (100%)
Locky	47/47 (100%)
Midie	1/1 (100%)
Neoreklami	0/1 (0%)

Family	Samples (%)
Occamy	4/4 (100%)
OpenCandy	2/2 (100%)
Petya	2/2 (100%)
QQPass	1/1 (100%)
Razy	6/6 (100%)
SageCrypt	1/1 (100%)
Saturn	1/1 (100%)
Scar	3/3 (100%)
Scatter	2/2 (100%)
Shade	2/2 (100%)
ShadowBrokers	1/1 (100%)
Shiz	17/17 (100%)
Sigma	0/1 (0%)
Sivis	3/7 (42%)
Spigot	2/2 (100%)
Spora	2/2 (100%)
Striked	0/1 (0%)
Swisyn	0/1 (0%)
Tescrypt	5/5 (100%)
TeslaCrypt	316/316 (100%)
Tpyn	1/1 (100%)
Upatre	2/7 (28%)
Ursnif	1/1 (100%)
Vobfus	1/1 (100%)
Wowlik	1/1 (100%)
Wyhymyz	1/1 (100%)
Zerber	52/52 (100%)
Zusy	7/7 (100%)
Total:	726/747 (97.1%)

before a file-write operation, eliminating the potential damage of ransomware. Moreover, Kharraz *et al.* proposed Redemption [11] that also uses the similar metrics for identifying a ransomware activity. However, in contrast to ShieldFS, Redemption redirects file-write operations to sparse files, rather than creating a full copy of each written file. Differently, *Data Aware Defense* (DaD) by Palisse *et al.* [18] uses chi-square test to determine if the written data is close to random distribution which is indicates that the file is being encrypted. DaD computes the sliding median of this indicator on the last fifty file-write operations and suspends the corresponding process that exceeds a predetermined threshold.

Key Escrow. Key-escrow based defense allows the ransomware to complete its attack. This approach is based on the idea that the files encrypted by ransomware

can be recovered if the encryption keys can be retrieved after the attack. For this aim, logging the keys used by ransomware is first appeared in the literature by Palisse *et al.* [19] and independently by Lee *et al.* [15]. The first public implementation of this idea PayBreak, with extending the idea to cover the third party crypto libraries was given by Kolondenker *et al.* [14]. In this system, all known cryptographic API are hooked, cryptographic materials are extracted and securely stored in a key vault. In the case of a ransomware attack, the encrypted files are tried by brute-forcing to be decrypted by retrieving the keys and other necessary parameters from the key vault. A slightly different method, *Deterministic Random Bits Generator* (DRBG) is proposed by Kim *et al.* [12] to retrieve the random numbers that ransomware used after an attack. DRBG replaces the CSPRNG of the system with a back-doored PRNG. The trapdoor is known only by the user and is preferably stored in the user's mobile device. After a ransomware incident, this trapdoor is retrieved and given to the PRNG to generate the same outputs that ransomware used. Using these outputs, ransomware's operations are reverted and files are recovered.

Deceptive Protection In this strategy, carefully-crafted files are placed as a *decoy* in the file system with the user's files. These decoys are not supposed to be modified/deleted by the user, so any write request to the decoy files are treated as an indicator of ransomware activity. RWGuard, a recently proposed system by Mehnaz *et al.* [16] uses this technique –in addition to behavioral analysis– to mitigate ransomware threat in real time.

7 Critical Discussion and Conclusions

Cryptographic ransomware is a modern global crime and a large amount of public and private institutions have been attacked already. The problem is that encryption is a powerful tool in the hands of criminals, hard to fight. By encrypting critical files on the victim's machine, ransomware blocks access to information and compromises critical services, wreaking an economical and social havoc because, unless victims pay the demanded ransom to receive the correct decryption key, they might not be able to recover their files if no backup is available. Computational complexity results ensure that a properly implemented encryption is irreversible, but to realize this theoretical result in practice, ransomware has to use cryptographically secure encryption keys. Many variants choose weaker alternatives: although there could be a theoretical solution to reverse their encryption at affordable costs, such *scareware* succeed in persuading victims to pay. Other variants, implement a theoretically weak but good-enough encryption to make decryption-without-the-key sufficiently painful to convince that paying the ransom is the lesser of two evils.

But in the restricted niche of ransomware that want their damage to be computationally irreversible, one finds the most disruptive variants, for instance, `WannaCry`, `Petya`, `GoldenEye`, `CryptoLocker`, `Crysis`, `Cerber`, `RAA`, and `NotPetya`. These ransomware families need a good source of random numbers and all of

them find it in the *cryptographically secure pseudo-random number generator* (CSPRNG) available on a victim's system. Today, such functions are indeed reliable and *de facto* source of cryptographic randomness available on a computer.

To contain the threat coming from ransomware in this cryptographically strong niche, Genç *et al.* proposed in [10] to control access to CSPRNG APIs. They proved the concept by stopping a very large class of real active ransomware from doing any damage to any file —remarkably including `NotPetya`, which was till that moment believed unstoppable. But a concept, as much as promising can be, is not yet a fully-fledged application. Discussing how to implement it into an effective anti-ransomware defense, herein called NoCry, is what we have done in this paper. We solved several critical security and design issues: how to ensure that the attack surface of the architecture is reduced; how to bootstrap the Whitelist DB, honest cryptographic applications calling CSPRNG APIs and maintain it with a minimal user intervention, arguably resulting in a very low false positive rate; how to reduce the overhead that the access control imposes on the systems performance to a negligible amount. By not relying on any IPC, we removed any know-to-be-vulnerable elements from the architecture, so addressing the first issue; we addressed the last, by a better decision making that drastically improves the overhead. With respect to the previous proof-of-concept, reducing it from several thousands percent down to about 20%: quantified, the overhead is now a few *hundredth of a second*.

These are quite evident improvements, but the solution we proposed to address the second issue *i.e.,* how to manage the Whitelist DB, needs further discussion. In a system assumed uncorrupted, we bootstrap the list in *Training mode* by feeding in honest applications that call CSPRNG. In *Deferred mode* we update the list when a new version of a whitelisted application is available; we temporarily grant it the right to call CSPRNG but retaining critical data that can help recovering files in rare case where the upgrade hides a ransomware. Despite looking reasonable to us, one can still challenge our choices. For instance, one can ask why managing a Whitelist DB of applications that call CSPRNG in the first place? In fact Windows OS already offers a protection, `AppLocker`, that enables to deny non-whitelisted apps (*e.g.,* malware) from running. Cannot be ransomware dismissed as any other malware? First, we observe, this practice seems not have slowed down ransomware so we conclude that it needs more time and maturity to be widely accepted. Second, the problem with the whitelists is that they may not be complete, generating fastidious false positives. This issue, of course, affects also NoCry, but differently from a system which offers protection against generic *harmful* apps (a term that may have different interpretation). NoCry targets and operate against a very specific situation. If we imagine to defer to the user the decision about whether a potential false positive is indeed so, NoCry can precisely state that a certain application is trying to call critical functions, potentially to create strong encryption keys and unless the application is meant to encrypt data, it is better to let NoCry kill it. We fail to imagine instead stating a similar precise claim to warn about a generic harmful application. The best could be a warning message sounding like "something insecure may happen",

alert that users have learned to ignore [5]. A precise claim like that, enabled by NoCry, will help users take more informed decisions, arguably reducing the number of false positives, and we intend to test this hypothesis in a future work.

Another critic can be that by only guarding access to CSPRNG, we miss to stop ransomware that generate encryption keys using different strategies. If this critics were well founded, this would count as a serious deficiency for NoCry because would lead to false negatives. To this critic we answer first by observing that NoCry was neither designed to stop other ransomware than those calling CSPRNG APIs nor conceived to work in isolation from other anti-ransomware. Indeed, we think, the full potential of NoCry will emerge only in integration with an anti-malware that provably can reverse the damage done by ransomware that make use of the encryption keys obtained not by calling CSPRNG. Theoretical solutions exist that have the potential ability to reduce the cost of reversing encryption to a feasible time complexity and a few solutions are actually implemented [23]. These seems, indirectly, to support the argument that by not calling CSPRNG APIs, ransomware can realize only cryptographically-weak encryption. It is then by uniting different methods that we imagine a good anti-ransomware can reliably combat this crypto-crime. How to do it properly is an open problem but considering the improvements that we have herein discussed, we believe this union is possible and practical.

There is a further observation to reinforce our rebutting to the critic. We are aware that there is no silver bullet for ransomware mitigation. Each defense system has pros and cons, and NoCry may well find its beater in some next generation ransomware. As discussed in [8], ransomware applications can find other ways than calling CSPRNG to get random numbers *e.g.,* by relying on non-cryptographic sources of randomness, but we believe that the alternative choices have weak points. The fact is that all the samples and variants of ransomware in the cryptographically-hard niche that we have analyzed so far, do call CSPRNG APIs. Thus, today, these functions are the most reliable source of randomness for application in search to build cryptographically strong encryption keys. And if in the future other functions will available for the same task, the fundamental question that remains to be solved is how many of these functions are, and whether by controlling access to these APIs, we can still implement a targeted strategy as the one in NoCry that enables a decision making with an arguably low false positive rate. Investigating such research questions requires time and we leave it as future work.

Other future work still needs to be done. The argument that we have a reduced false positive rate has to be supported by experimental evidence. This means to run stress tests while running a generous number of various benign cryptographic applications under different conditions. Beyond having measured the overhead in terms of loss of performance, we still need to assess the user experience (UX) of NoCry running on different kinds of computers, included on battery powered mobile devices, to verify whether the overhead is imperceptible, as we claim, by users in their daily activities.

Acknowledgements

This work was partially funded by Luxembourg National Research Fund (FNR) under the grant agreement PoC18/13234766-NoCry PoC.

References

1. Avast: PC Trends Report 2019 (Apr 2019), retrieved June 1, 2019 from https://blog.avast.com/pc-trends-reports
2. Bajpai, P., Sood, A.K., Enbody, R.: A key-management-based taxonomy for ransomware. In: 2018 APWG Symposium on Electronic Crime Research (eCrime). pp. 1–12 (May 2018)
3. Bui, T., Rao, S.P., Antikainen, M., Bojan, V.M., Aura, T.: Man-in-the-machine: Exploiting ill-secured communication inside the computer. In: 27th USENIX Security Symposium. pp. 1511–1525. USENIX Association, Baltimore, MD (2018)
4. Continella, A., Guagnelli, A., Zingaro, G., De Pasquale, G., Barenghi, A., Zanero, S., Maggi, F.: Shieldfs: A self-healing, ransomware-aware filesystem. In: Proc. of the 32 nd A Conference on Compt. Security Applicat. pp. 336–347. ACM, New York (2016)
5. Cormac, H.: So long, and no thanks for the externalities: the rational rejection of security advice by users. In: Proc. of the 2009 New Security Paradigm Workshop (NSPW), September 8–11, 2009, Oxford, United Kingdom. pp. 133–144. ACM (2009)
6. CyberEdge: 2018 Cyberthreat Defense Report (Mar 2018), retrieved June 3, 2019 from https://cyber-edge.com/wp-content/uploads/2018/03/CyberEdge-2018-CDR.pdf
7. Gammons, B.: 4 Surprising Backup Failure Statistics that Justify Additional Protection (2017), retrieved June 3, 2019 from https://blog.barkly.com/backup-failure-statistics
8. Genç, Z.A., Lenzini, G., Ryan, P.Y.A.: Next generation cryptographic ransomware. In: Secure IT Systems - 23rd Nordic Conference, NordSec 2018, Oslo, Norway, November 28-30, 2018, Proceedings. LNCS, vol. 11252, pp. 385–401. Springer (2018)
9. Genç, Z.A., Lenzini, G., Ryan, P.Y.A.: Security analysis of key acquiring strategies used by cryptographic ransomware. In: Proceedings of the Central European Cybersecurity Conference 2018. pp. 7:1–7:6. CECC 2018, ACM, New York, NY, USA (2018)
10. Genç, Z.A., Lenzini, G., Ryan, P.Y.: No random, no ransom: A key to stop cryptographic ransomware. In: Proc. of the 2018 Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'18). Springer, Cham (2018)
11. 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 (2017)
12. Kim, H., Yoo, D., Kang, J.S., Yeom, Y.: Dynamic ransomware protection using deterministic random bit generator. In: 2017 IEEE Conference on Application, Information and Network Security (AINS). pp. 64–68 (Nov 2017)
13. KnowBe4: KnowBe4 Alert: New Strain Of Sleeper Ransomware (May 2015), retrieved June 1, 2019 from https://www.knowbe4.com/press/knowbe4-alert-new-strain-of-sleeper-ransomware

14. Kolodenker, E., Koch, W., Stringhini, G., Egele, M.: Paybreak: Defense against cryptographic ransomware. In: Proc. of the 2017 ACM on Asia Conference on Compt. and Commun. Security. pp. 599–611. ACM, New York (2017)
15. Lee, K., Oh, I., Yim, K.: Ransomware-prevention technique using key backup. In: Jung, J.J., Kim, P. (eds.) Big Data Technologies and Applications. pp. 105–114. Springer International Publishing (2017)
16. Mehnaz, S., Mudgerikar, A., Bertino, E.: Rwguard: A real-time detection system against cryptographic ransomware. In: Research in Attacks, Intrusions, and Defenses. pp. 114–136. Springer, Cham (2018)
17. Microsoft: Named Pipes (May 2018), retrieved June 3, 2019 from https://docs.microsoft.com/en-us/windows/desktop/ipc/named-pipes
18. Palisse, A., Durand, A., Le Bouder, H., Le Guernic, C., Lanet, J.L.: Data aware defense (dad): Towards a generic and practical ransomware countermeasure. In: Secure IT Systems. pp. 192–208. Springer, Cham (2017)
19. Palisse, A., Le Bouder, H., Lanet, J.L., Le Guernic, C., Legay, A.: Ransomware and the Legacy Crypto API. In: The 11th Int. Conference on Risks and Security of Internet and Syst. pp. 11–28. Springer (Sep 2016)
20. Roussev, V.: Data fingerprinting with similarity digests. In: Chow, K.P., Shenoi, S. (eds.) Advances in Digital Forensics VI. pp. 207–226. Springer, Berlin (2010)
21. Scaife, N., Carter, H., Traynor, P., Butler, K.R.B.: Cryptolock (and drop it): Stopping ransomware attacks on user data. In: 2016 IEEE 36th Int. Conference on Distributed Comput. Syst. (ICDCS). pp. 303–312 (June 2016)
22. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: Avclass: A tool for massive malware labeling. In: Int. Symp. on Research in Attacks, Intrusions, and Defenses. pp. 230–253. Springer, Cham (2016)
23. Young, M., Zisk, R.: Decrypting the negozi ransomware (2017), retrieved June 1, 2019 from https://yrz.io/decrypting-the-negozi-ransomware