AntiPatterns Regarding the Application of Cryptographic Primitives by the Example of Ransomware

Michael P. Heinl
Fraunhofer AISEC

Alexander Giehl
Fraunhofer AISEC

Lukas Graif
Technical University of Munich
Fraunhofer AISEC

ABSTRACT

Cryptographic primitives are the basic building blocks for many cryptographic schemes and protocols. Implementing them incorrectly can lead to flaws, making a system or a product vulnerable to various attacks. As shown in the present paper, this statement also applies to ransomware. The paper surveys common errors occurring during the implementation of cryptographic primitives. Based on already existing research, it establishes a categorization framework to match selected ransomware samples by their respective vulnerabilities and assign them to the corresponding error categories. Subsequently, AntiPatterns are derived from the extracted error categories. These AntiPatterns are meant to support the field of software development by helping to detect and correct errors early during the implementation phase of cryptography.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

AntiPatterns, Cryptography, Ransomware

ACM Reference Format:
Michael P. Heinl, Alexander Giehl, and Lukas Graif. 2020. AntiPatterns Regarding the Application of Cryptographic Primitives by the Example of Ransomware. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3407023.3409182

1 INTRODUCTION

A wide variety of cryptographic schemes and protocols are based on more granular cryptographic primitives [57]. These cryptographic building blocks are used with the intention to ensure security objectives such as authenticity, integrity, and confidentiality. However, their realization is not trivial: various aspects have to be taken into account for the creation and management of cryptographic keys as well as for the implementation of the actual algorithms. Both a too short time to market and insufficient knowledge of developers in regard to cryptography can lead to implementation flaws resulting in vulnerabilities which can potentially be exploited by attackers.

Some ransomware samples are good examples of insufficiently implemented cryptographic primitives. This type of malware usually employs cryptography in order to encrypt data stored on the computers of its victims, virtually 'holding them hostage'. Against payment of a ransom, those affected are then offered the cryptographic key to decrypt their data. Therefore, this type of malware is referred to as *ransomware*. Flawed implementations of the encryption mechanisms used by some ransomware samples, however, allow the decryption of data without paying the ransom. Therefore, this specific ransomware can be considered 'broken'. While there are also other types of ransomware which do not employ any cryptographic methods but rather modify the underlying operating system or master boot record (MBR) in order to block access to the computer (so-called *Lockers* [8]), this paper solely focuses on so-called *crypto ransomware* [59].

Using the example of broken ransomware samples and a self-developed categorization framework, this paper analyzes common errors during the implementation of cryptographic primitives by assigning them different categories. These error categories are derived from other publications, although none of the analyzed papers provides such an overall scheme.

Eventually, *AntiPatterns* are formulated based on the identified errors. In general, an AntiPattern describes an allegedly good and common approach which in fact is counterproductive and leads to negative consequences [63]. The reason for this can be insufficient knowledge and experience or a wrongly applied design pattern. Whilst describing bad practices during the implementation of cryptographic primitives, AntiPatterns also describe resulting consequences and recommendations to prevent such errors - similar to design patterns. In fact, the concept of AntiPatterns is mostly based on the idea of design patterns and the insight that it is also important to document unsuccessful solutions and their negative consequences in order to avoid making the same errors again [9].

This paper is explicitly not meant as a guideline to develop the 'perfect' ransomware. Rather, the results are meant to help detecting errors during the implementation of cryptographic primitives or even avoid them from the very beginning. Simultaneously, the AntiPatterns should increase the developers' awareness regarding the fragility of cryptographic implementations and that even minor errors can have a severe impact on a future product's security. Following the original idea of patterns, this paper does not discuss specific protocols or libraries but rather generic methodologies which can be wrongly applied by developers. The proper usage of publicly available protocols and libraries which are actually considered secure but turned out to be vulnerable is not regarded as an error. However, incorrect usage of such libraries by the developer, for example a weakly initialized block cipher or wrongly set status indicator, is on the other hand very well considered to be an error.

ARES 2020, August 25–28, 2020, Virtual Event, Ireland
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland*, https://doi.org/10.1145/3407023.3409182.

2 RELATED WORK

Multiple authors already discussed the topic of flawed cryptographic implementations in different contexts [17, 29, 38, 49, 52, 54, 58]. Making use of these papers, section 3 presents error categories which are in turn used to derive AntiPatterns.

In [38], the authors analyze a selection of 269 Common Vulnerabilities and Exposures (CVEs) marked as *Cryptographic Issues*. They are then categorized regarding the following aspects:

- type of error making the system vulnerable,
- part of the system containing the error,
- the error's impact on the system's security.

The categorization shows that 83% of the analyzed CVEs are caused by misuse of cryptographic libraries. The study is supported by brief descriptions of errors and exemplified by corresponding CVEs. The authors also provide recommendations for improvement which are, however, rather on a procedural level (formal verification of code, test phases etc.) than the more tangible AntiPatterns developed in the present paper.

Schneier et al. [54] discuss the problem of implementation weaknesses from the perspective of a saboteur who aims to clandestinely insert vulnerabilities in order to compromise a system's security. The authors discuss various ways how this goal can be achieved, underpinned by corresponding historic examples. The vulnerabilities are then categorized by a self-developed classification system.

Similar to the present paper, Herzog and Balmas [29] analyze implementation flaws in the area of cryptography by the example of various types of malware such as exploit kits, trojans, and ransomware. Herzog and Balmas describe the errors without giving recommendations on how to avoid them.

The study of Das et al. [58] discusses the erroneous usage of six cryptographic libraries which are analyzed regarding typical problems, such as improper default parameters or insecure code examples, and how they promote misuse by developers. Implementation flaws of the cryptographic libraries themselves are out of scope but the study gives an overview of errors possible to happen during the implementation of these very libraries. Additionally, recommendations are given in form of best practices.

Egele et al. [17] analyze implementation flaws of Android applications. For this, six rules for the implementation of cryptographic primitives are formulated and integrated into a tool which is then used to analyze approximately 11,700 Android applications in order to find out whether they violate those rules. According to the study, 88% of the analyzed applications contain cryptographically relevant implementation flaws. The recommended mitigations to decrease the error rate focus on implementation and documentation of application programming interfaces (APIs).

Gadient's work [49] builds on [17]. The work identifies and briefly describes 28 sources of error and corresponding mitigations whereby several have no relation to cryptography. One important point is that Gadient formulates so-called 'symptoms' for each source or error, indicating possible vulnerabilities in the source code. Similar to Egele et al., Gradient develops a tool which he uses to search a total of 46,000 Android applications for indications of the formerly described symptoms. The paper concludes that these symptoms are a good indicator for actual implementation flaws.

In [55], different Java applications are analyzed for security risks. Using the gained insights, a catalog containing seven AntiPatterns focussing on Java components is developed. However, these AntiPatterns are not related to cryptography.

The structure of the AntiPatterns formulated in the present paper is based on recommendations of Brown et al. [63] who give not only a comprehensive introduction to AntiPatterns but also describe different types of notation. Furthermore, example AntiPatterns focusing on software development, software architecture, and project management are provided, however, without taking security-related topics into consideration.

3 CONCEPT

This section describes the conceptual foundations of this work by detailing on the used approach and the categorization of the results. The described process is shown in Figure 1. First, relevant ransomware samples are identified during a literature survey in subsection 3.1. Also, the errors resulting in a broken ransomware for the found samples are identified. Then, a hierarchy of errors is developed for categorization of these errors in subsection 3.2. Here, individual ransomware samples are assigned to their respective error categories. The results of this section are then, consequently, used in deriving the AntiPatterns in section 4.

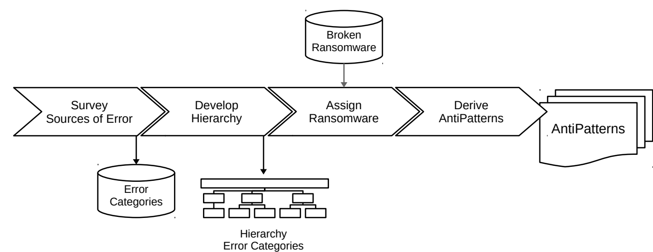

Figure 1: Conceptual process of the work.

3.1 Ransomware Literature Review

This section describes the process of surveying ransomware samples required for the description of the AntiPatterns. The criteria used to chose relevant ransomware for further studies is provided first. For the remainder of this work, only ransomware samples affecting the availability of data by encryption are discussed (cryptographic ransomware). The focus is on cryptographic ransomware affecting end user and office computers. Cryptographic ransomware for mobiles or Internet of Things (IoT) devices is out of scope.

Ransomware samples are categorized in regard to their programming errors that lead to broken cryptography. For this, it is paramount that the ransomware contains such an error. Ransomware samples which are broken because of the following reasons are therefore out of scope:

- private key leakage (*TeslaCrypt* v3-4 [37]),
- issues with non-crypto mechanisms such as manipulation of unencrypted server responses (e.g. Jigsaw v1 [11]).

In order to allow an exact categorization, the ransomware's vulnerability needs to be fully documented. Non or only partially documented ransomware is also out of scope.

The goal of this paper is to point out errors that can occur during the implementation of cryptographic primitives. Hence, only cryptography-related errors produced by the developers of the ransomware are relevant. The usage of protocols allowing attacks under certain circumstances (e.g. TLS allows padding oracle attacks when using certain configurations [62]) are not considered. The same applies for errors within cryptographic libraries used for the implementation of cryptography within the respective ransomware. The process used for conducting the literature review is shown in Figure 2.

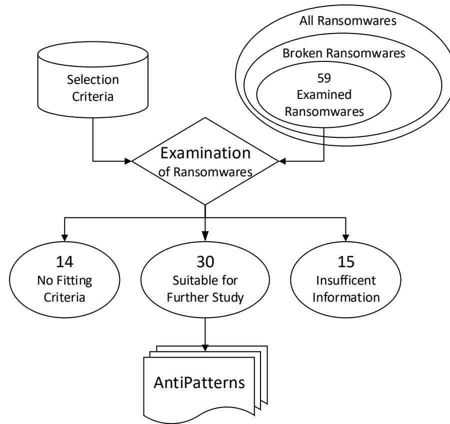

Figure 2: The process of selection and research.

First, an overview of the landscape of cryptographic ransomware is generated. Sources for this are publications of government institutions as well as scientific ransomware databases such as *ID Ransomware* which is capable of identifying 560 different ransomware samples.[1] From this, the set of ransomware (see top right-hand corner of Figure 2) is derived. Continuing from there, databases for decryption tools for ransomware provided by manufacturers of antivirus software give insight into which of those ransomware samples are effectively broken, i.e. a decryption solution is publicly available. This broken ransomware is further examined in regard to their vulnerability and the requirements stated in subsection 3.2. However, some antivirus companies do not publish detailed information about the discovered vulnerability such as how the encryption is broken or details about the implementation of the tool used to decrypt the files targeted by the ransomware. This can be explained by antivirus companies not wanting to aid the developers of ransomware in improving their 'product'.

In conclusion, a total of 59 cryptographic ransomware samples are examined. From those, 14 do not satisfy the conditions for further examination stated above. Another 15 ransomware samples are considered broken but are still unsuitable as no information on them is available during the time of this study. For the 30 remaining ransomware samples, detailed information on their respective vulnerabilities can be found.

[1]As of March 19, 2018: https://id-ransomware.malwarehunterteam.com/

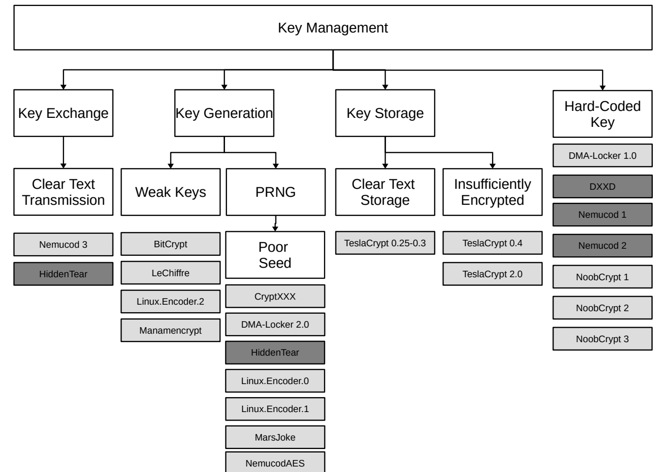

Figure 3: Key management error categories with assigned ransomware samples.

3.2 Categorization Framework

In order to formulate AntiPatterns regarding the application of cryptographic primitives, different sources of errors have to be defined first which are then in turn used to create abstract error categories. These error categories summarize similar errors of the same origin with only minor differences.

Several publications discussing errors during implementation of cryptographic primitives are used as a basis for the formulation of the following error categories [17, 29, 38, 49, 52, 54, 58]. Based on them, a system of possible sources of error has been developed which is subsequently complemented by additional categories identified during the survey of broken ransomware. The sources of error extracted from the mentioned research literature are treated similarly as the surveyed ransomware which means that errors which are not related to cryptography are not considered. Similarly, errors caused by the usage of vulnerable protocols or libraries are excluded as well. The focus especially lies on errors in the area of key management and the implementation of cryptographic primitives. In order to structure the assignment of broken ransomware to the corresponding categories, a framework is developed clustering similar error categories into main categories.

During the investigation, a total of ten error categories have been identified and divided into the two main categories *Key Management* and *Implementation* as illustrated in Figure 3 and Figure 4.

The surveyed ransomware is categorized by a step-by-step analysis employing the developed framework (ransomware is indicated by a grey box in Figure 3 and Figure 4). If a ransomware contains multiple relevant errors leading to broken encryption, it is categorized repeatedly (indicated by a dark-grey box).

3.2.1 Key Management Errors. Figure 3 shows error categories in the area of key management:

- **Clear text transmission of key exchange:** Keys generated on the victim's client are transmitted to the attacker's server in clear text due to the absence of network data encryption such as Transport Layer Security (TLS).

- **Usage of weak keys:** Files on the victim's client are encrypted using weak, easy to break keys. This category includes keys of insufficient length and keys generated by a flawed (potentially self-developed) key generation algorithm.
- **Poor initial values (seed) of pseudo random number generator (PRNG):** An insecure (deterministic) PRNG is used in combination with a predictable or constant initial value.
- **Clear text storage of keys:** The used keys are stored unencrypted on the victim's client.
- **Weak encryption of keys:** The used keys are stored either inefficiently encrypted or only obfuscated.
- **Hard-coded keys:** The used keys can be found in the ransomware's code.

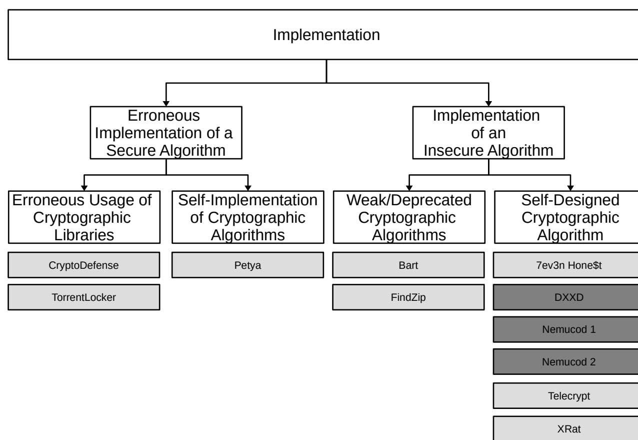

Figure 4: Implementation error categories with assigned ransomware samples.

3.2.2 Implementation Errors. Figure 4 shows error categories in the area of implementation of cryptographic primitives:

- **Erroneous usage of cryptographic libraries:** Errors likely to happen when using cryptographic libraries. Examples include wrong initialization of modes of operation, usage of insufficient default configurations, and copying of insecure example snippets. Errors caused by the library's implementation itself are explicitly excluded.
- **Self-implementation of cryptographic algorithms:** The algorithms used for data encryption are implemented by the developers themselves.
- **Usage of weak/deprecated cryptographic algorithms:** Algorithms known as weak or deprecated are used for the encryption of data.
- **Self-designed cryptographic algorithm:** An algorithm designed by the developers themselves is used for the encryption of data.

The ransomware categorization is conducted according to the framework presented above. The ransomware samples are assigned to error categories by successively narrowing down the error leading to a ransomware's vulnerability. In case a ransomware contains multiple relevant errors which result in broken cryptography independently from each other, the ransomware is assigned multiply.

Figure 3 and Figure 4 list the categorizations of the 30 ransomware samples which satisfy the requirements and are therefore analyzed in detail. While Figure 3 shows ransomware with errors in the area of key management, Figure 4 shows the ones with errors in the area of implementation.

4 DERIVED ANTIPATTERNS

In the following, the derived AntiPatterns are formulated. The sources of error presented in section 3 are the Antipatterns' structural foundation whereby very similar errors are summarized. The ransomware samples classified in the previous part of the paper are referred to in the corresponding AntiPatterns. They help to make the AntiPatterns more tangible by serving as negative examples. Hence, although AntiPatterns are generic by nature, the paragraphs describing these ransomware examples may refer to specific technology and languages for the sake of tangibility.

4.1 Clear Text Transmission

Developers are often confronted with the challenge of establishing information exchange between communication partners (e.g. between an application and the corresponding server) on insecure channels. If sensitive information has to be transmitted, this is a very important aspect during implementation. One very pragmatic approach is to transmit confidential information without additional security measures in clear text, for example by employing HTTP.

Attackers can eavesdrop the unprotected transmission and obtain sensitive information such as cryptographic key material. Furthermore, data could be modified unnoticedly, meaning that not only confidentiality but also integrity is at stake.

Several ransomware samples, such as *Nemucod 3* or *HiddenTear*, generate the keys used for encryption on their victim's client [53, 61]. The keys are then sent over the Internet to the attacker's server infrastructure in order to sell them to their victims once the encryption process is complete. However, both examples communicate unencrypted. By capturing the network traffic between the affected computer and the Internet, the key can be extracted from the corresponding message and the affected data decrypted without paying the ransom.

4.2 Weak Keys

In order to apply cryptography, usually a key of a given length is necessary which has to be generated beforehand. For this reason, similarly to the AntiPattern *Self-Implementation*, self-developed algorithms can be used. Is the generation algorithm based on constant or deterministic values or is the generated key not long enough, the algorithm's security can be compromised.

Cryptographic keys generated by insecure generation algorithms can potentially be easily recalculated. An attacker can analyze and reconstruct the used algorithm by inspecting the program's code. The usage of constant or deterministic values eases the reconstruction of keys by the attacker. The security services provided by the underlying cryptographic algorithm are therefore at stake. If keys or other cryptographic parameters are not generated with the necessary length, they can be vulnerable to brute-force attacks.

For example, when using algorithms based on the integer factorization problem, the corresponding prime factors can be found within a reasonable amount of time if the integers are too short.

The ransomware *BitCrypt* claims to use RSA-1024 for data encryption but the used RSA modulus is not a 128-byte/1,024-bit value but a 128-digit number which can be represented by 426 bit [18]. As a consequence, the data is encrypted with 'RSA-426' instead of RSA-1024 which means that the too short RSA modulus can be factorized within a couple of hours without having to employ any special hardware. The private key can then be calculated with the help of the public key embedded in the ransomware's code.

Manamencrypt uses a `.rar` archive protected by a password which is a concatenation of two parts, a SHA-1 hash incorporating three hardware features and the name of the current user [4]. If the generation algorithm is known, the password can be recalculated by extracting the corresponding user name and hardware information. The calculation of the SHA-1 digest is even negligible since it is also used as the name of the archive which further eases the password's reconstruction.

A similar case is *LeChiffre* which also uses a hash digest incorporating multiple hardware features as symmetric key [42].

4.3 Poor Seed

Random numbers are broadly used in the area of cryptography, especially when it comes to the generation of cryptographic keys. Many programming languages provide PRNG implementations which generate numbers based on an initial value (seed) and a deterministic algorithm. Although the output numbers appear to be random, the pseudo-random sequence is always the same if the PRNG is fed with the same seed.

Intuitively, such PRNGs are often used for key generation. However, if not explicitly provided by the developer, the generator might use constant or deterministic default values, for example the current system time, as seed. The generated sequence of numbers can then directly be used as a key or as input value for a KDF. Another variant could be the self-development of an algorithm for key generation based on the PRNGs how it can often be seen when it comes to ransomware.

A poor seed can easily be guessed and recalculated under certain circumstances. System times, for example, can be guessed based on modification timestamps of encrypted or newly created files. If the initial sequence is partially known (for example in the form of IVs generated before and after the time of key generation), the exact seed can be brute-forced within a short period of time. If this is not the case, a dictionary consisting of seed-key pairs can be generated and a cipher text decrypted until it results in a meaningful clear text. The seed corresponding with the successful key is then the searched initial value. Similar to keys, seeds of insufficient length can easily be found by employing this brute-force method. If a constant value is used, it can be determined by decompiling the binary.

Once an attacker has found the correct seed, she can reconstruct the exact sequence of random numbers and subsequently determine all cryptographic keys which have been generated using this sequence. Each encryption using this sequence can therefore be broken.

Linux.Encoder.1 utilizes a PRNG initialized with the system time for generating the keys and IVs used by AES-128. *CryptXXX* also uses the system time as initial value but reconstructs the original timestamps of affected files and uses a new seed for each file [32]. However, making use of the ransom note generated in each directory after having encrypted its content, the initial value can be guessed. The ransom note provides hours and minutes so that for each file only seconds and milliseconds have to be determined which results in 60,000 combinations. The PRNG gets initialized with each of these candidates in order to generate a key. This key is then used to decrypt the file's first four bytes (*magic number*) which indicate the type of file. If the magic number matches the file type indicated by the original filename extension (*CryptXXX* only attaches an additional extension), the correct key is found and the file can therefore be restored.

NemucodAES renders infected files useless by overwriting the first 2,048 bytes of each file with random values [1]. The formerly extracted bytes are encrypted with AES-256 and stored in a database. *NemucodAES*'s developers haven't initialized the used PRNG (PHP's `mt_rand`) which therefore uses an automatically generated, 32-bit long seed. For cryptographic purposes, this length is not sufficient and renders the initial value prone to brute-force attacks [46]. In order to restore the original files, the PRNG is successively initialized with all 2^{32} values until the generated sequence of numbers matches the first 2,048 bytes of the file which was (presumably) infected first. Using the found seed, the keys for all encrypted entries of the database can be computed.

4.4 Key Storage

The security of a cryptographic algorithm depends on the confidentiality of the used keys. Therefore, it is very important to appropriately store keys after their generation in order to protect them from unauthorized access.

A naive approach would be to store the keys in a hidden file and to trust the assumption that this file is neither found nor opened and read. This could be realized by giving the file a self-invented file extension and placing it in a hidden directory.

An ideal approach would be the encryption of keys before saving them. This, however, would lead to a new key which has to be stored. Another possible approach could therefore be the usage of reversible mathematical operations which would result in encoding rather than encryption but cannot be restored without the knowledge about the used operation.

When a key is stored as clear text, it can be extracted and used by an attacker, especially if there are no access restrictions enforced. Storing a key file in a hidden directory is therefore not an effective measure to protect the key. The same applies for disguising the file by using a self-invented file extension.

If a key is encrypted inappropriately, for example by applying one of the AntiPatterns formulated in this paper, it can potentially be reconstructed. Encoding the key by applying reversible mathematical operations is not considered secure encryption because the algorithm can be extracted after decompiling the respective binary.

Concluded, the desired protection objectives which are actually provided by the respective cryptographic algorithm are at stake because the key is not protected properly.

TeslaCrypt, a ransomware focusing on Windows environments, demonstrates how the mentioned approaches can result in vulnerabilities. The first versions of *TeslaCrypt* stored the key needed for the used AES-256 algorithm after its generation in a file located in the default `AppData` folder [19]. The key stays in this file until the end of the encryption process. If this process is interrupted, for example by shutting down the computer, the file persists and the key can be extracted.

TeslaCrypt 0.4 encodes the key as multiplicative inverse modulo the order of an elliptic-curve cryptography (ECC) curve before writing it into the file [19]. However, the key can still be extracted by employing the already mentioned method. It just has to be reconstructed by computing the multiplicative inverse.

TeslaCrypt 2.0 stores the generated symmetric key directly in the infected files encoded by a method similar to *Elliptic-Curve Diffie–Hellman* (ECDH) [19]. By multiplying a private key with the generator point of an elliptic curve, the public key is generated. The symmetric key is then obscured by linking it with the ECC public key. However, the coordinates of the generator points are relatively short which leads to the fact that the public key can be factorized within a reasonable amount of time, i.e., a few minutes up to a couple of days. Utilizing the recomputed factors, the private key can be reconstructed and the symmetric key subsequently restored [7, 36].

4.5 Hard-Coded Keys

Different approaches exist for providing cryptographic keys for ciphers. One of those approaches is to directly embed the key as a constant value into the program code, i.e., hard-coding it. This may be sufficient when handling public keys within an asymmetric crypto system as those keys are not required to be secret, although it may lead to problems regarding later key revocation. However, if hard-coding is used for keys within symmetric crypto systems, far-reaching consequences for the security of the cipher can occur.

A hard-coded key can be extracted by decompiling the source code. If this key is used for a symmetric cipher, the confidentiality is threatened. The same is true if the private key of an asymmetric cipher is extracted. In the case of generation of message authentication codes (MACs), the integrity and authenticity of the messages cannot be guaranteed anymore. For digital signature schemes, the integrity and authenticity as well as the accountability of all signed messages is at stake.

Each copy of the software embeds the same key meaning one broken copy affects all other available copies resulting in a broken crypto system. The revocation of a (compromised) key is also difficult as the key is provided directly via the source code and can therefore only be changed by replacing the distributed binary. If no revocation takes place, the vulnerability remains.

All three versions of the ransomware *NoobCrypt* use a hard-coded cryptographic key [25]. By examining the decompilied source code of the ransomware, the key can be extracted. After publication of the key, all files decrypted by the ransomware could be successfully decrypted as each version uses the same private key. The ransomwares *DXXD*, *Nemucod 1*, and *Nemucod 2* were broken by a similar approach [35, 48, 50].

4.6 Erroneous Library Usage

Cryptographic ciphers are often implemented using cryptographic programming libraries. These libraries provide an implementation of different ciphers via a documented API. For correct usage of the library, consultation of the provided documentation is required. Otherwise, API calls can be used incorrectly and unreasonable parameters can be handed over to the ciphers. The parameters can be outside the expected range or chosen randomly. This can, in turn, lead to function calls being set with predefined default parameters during compile time. Alternatively, the sample code included in some documentations can be used directly with only minor or completely without any modifications.

The usage of incorrectly chosen parameters when calling an API can therefore lead to unexpected behavior of the application and compromise the security of the cipher. The same applies for copying sample code without the understanding of the code's function. The usage of standard, predefined execution parameters for ciphers can enable attacks on ciphers.

The ransomware *CryptoDefense* uses RSA-2048 to encrypt files on systems running Windows [29]. The keys used by the ransomware are generated locally on the infected system using different functions of Microsoft's Crypto API such as `CryptAcquireContext`. The function call's code was adapted from a code sample. The flag indicating the private key's persistence is set to the value `0` by default. This results in the key being stored in the user's local `AppData` folder [56]. Therefore, extraction and decryption of the files is possible.

The ransomware *TorrentLocker* encrypts files with AES-256 in CTR mode by employing the library *LibTomCrypt* [33, 41]. Each file is encrypted with the same key and initial vector which allows breaking the ransomware.

4.7 Weak Algorithms

By selecting a specific cipher from the large pool of existing ciphers, a possibly insecure cipher can be selected without intent. Reasons for this can be manifold. Unawareness or preferred use of already known ciphers can, for example, result in the selection of an outdated cipher which is no longer considered secure.

Outdated or provably weak ciphers often contain severe weaknesses making these ciphers vulnerable to attacks. Some of them employ insufficient key lengths which makes these ciphers susceptible to brute-force or factorization attacks due to the increase in available computing power. Weaknesses of such ciphers are often publicly known and investigated which additionally decreases the effort for potential attackers.

Bart, a fork of the ransomware *Locky* [51], uses encrypted `ZIP` archives [48] utilizing the PKZIP algorithm which is vulnerable to known-plaintext attacks [6, 31].

4.8 Self-Implementation

As outlined, developers often use libraries to integrate cryptographic functionality into their programs. Alternatively, they can implement cryptographic algorithms completely by themselves whilst relying on published specifications. Additionally, developers can even design their own cryptographic algrithms which are often based on simple logic operators. A motivation for this could

be the desire to keep the binary as small as possible which can be a challenge regarding the storage claimed by some library's functions.

Self-implementation of secure cryptographic algorithms can result in subtle, difficult to notice errors. These errors can have severe implications on the security of the used cryptography, in the worst-case resulting in broken cryptography. In addition to this, self-developed cryptographic algorithms also show errors on a conceptual level, making broken cryptography even more likely. A (self-developed) algorithm depending on the non-disclosure of the technique itself, for example, violates one of the most important foundations of modern cryptography, namely *Kerckhoff's Principle*.

The 2016 ransomware *Petya* does not encrypt single files but rather the MBR and the Master File Table (MFT) of the infected computer's hard disk drive [29, 30] which results in an unusable operating system. The key stream of the employed stream cipher *Salsa20* is generated from an initial state of 64 bytes consisting of a 256 bit key and a 64 bit nonce among others [5, 14]. *Petya*'s developers use a self-implemented version of *Salsa20* containing several errors effectively reducing the key length by half. Thus, the employed encryption gets prone to brute-forcing [26, 30].

The ransomware *Telecrypt* communicates via the API of the messenger service *Telegram* [40]. For file encryption, a string of 10-20 characters randomly generated from a fixed set of symbols including `vo`, `pr`, `bm`, `xu`, `zt`, and `dq` is used. The encryption is implemented by byte-serial addition of key and file content. This self-developed algorithm is vulnerable to known-plaintext attacks since the key can be calculated by comparing an encrypted with an unencrypted file resulting in a broken cipher. The ransomware samples *DXXD*, *Nemucod 1*, and *Nemucod 2* also use a self-developed cipher resulting in broken cryptography [35, 48, 50].

5 CONCLUSION AND FUTURE WORK

This paper analyzes 30 broken ransomware samples for erroneous implementations of cryptography and presents a categorization framework to assign the analyzed ransomware to ten different error categories. Based on this, a total of eight AntiPatterns which can occur during the implementation of cryptography are derived. The analysis of the results shows that insufficient key management has a greater potential for error than the implementation of cryptographic primitives. Among the analyzed ransomware, hard-coded keys and insufficiently chosen initial values for PRNGs are the most common causes of vulnerabilities. Nevertheless, the set of analyzed ransomware is not large enough in order to draw a statistically sound conclusion about the general frequency of specific errors or AntiPatterns, respectively.

The categorization framework developed in this paper is based on error categories which have been extracted from previous works on implementation errors of cryptography. None of these papers, however, contains such an integrated framework which can not only be used for the categorization of ransomware. Rather, erroneous applications of all kind can be assigned to a specific error category. Based on the assigned category, the corresponding AntiPattern can be determined which helps developers to learn about common mistakes, their possible consequences, and approaches to avoid an error or even a vulnerability.

The developed AntiPatterns focus especially on the usage of cryptographic primitives. They are not limited to a specific programming language and are therefore universally applicable. The given recommendations for the avoidance or correction of errors are not meant as instructions which can be directly implemented. This would be contradictory to the general idea of an AntiPattern. Rather, they are intended to be proposed solutions for the specified (and similar) problems.

This paper explicitly addresses lawful purposes only. It is not the goal to promote the development of a 'perfect' ransomware. The AntiPatterns are primarily intended to support the field of software development. No matter if libraries, applications, or embedded systems, the AntiPatterns can help to detect and avoid errors during the implementation of cryptography at an early stage.

A specific example is the development of industrial control systems (ICS) [23]. Apa et al. [39] analyze several devices used for the automation of production lines and found a vulnerability in the area of key generation, significantly reducing the search space of brute-force attacks. This enables an attacker to obtain the correct cryptographic key within a short period of time and use it in order to physically tamper with the production plant. Making use of AntiPatterns, this vulnerability could have been detected and corrected during the development phase of the device. Other common problems in ICS security are transmission in clear text and the use of wrong cryptographic libraries [22]. Considering the respective AntiPatterns, these error sources can be mitigated.

Possible future work can include the analysis of a larger set of ransomware but also other, lawful software in order to identify additional error categories, extend the existing scheme, and consequently derive corresponding AntiPatterns. By increasing the number of research objects, a more quantitative and statistical conclusion can be drawn about which errors are the most common. Based on this insight and existing methodologies [24, 27, 28], a metric can be developed, ordering the possible errors in accordance with their frequency and impact on the actual product in terms of vulnerabilities. Similar to the OWASP Top Ten [45], this metric can then be used during code reviews in order to detect and correct vulnerabilities, qualitatively making use of the accompanied AntiPatterns.

As already discussed earlier, there are approaches to develop tools automatically analyzing Android applications in order to find vulnerabilities [17, 49]. Applying this concept to ransomware could help to support manufacturers of antivirus software in their continuous research for decryption tools. Decompiling the binaries first, the actual analysis could then start on the assembly level whereby it has to be considered that parts of the code might be obfuscated and/or encrypted which would impede the analysis.

ACKNOWLEDGMENTS

The presented work is part of the German national security reference projects IUNO (grant number 16KIS0324) and IUNO InSec (grant number 16KIS0933K). The authors would like to thank Patrick Wagner for providing valuable feedback.

REFERENCES

[1] Adam Caudill. 2017. *Breaking the NemucodAES Ransomware*. https://adamcaudill.com/2017/07/12/breaking-nemucodaes-ransomware/

[2] Elaine Barker. 2016. *NIST Special Publication 800-57 Part 1 Revision 4: Recommendation for Key Management*. Technical Report. National Institute of Standards and Technology (NIST). https://doi.org/10.6028/NIST.SP.800-57pt1r4

[3] Elaine Barker and Allen Roginsky. 2012. *NIST Special Publication 800-133: Recommendation for Cryptographic Key Generation*. Technical Report. National Institute of Standards and Technology (NIST). https://doi.org/10.6028/NIST.SP.800-133

[4] Sabrina Berkenkopf. 2016. *Manamecrypt – a ransomware that takes a different route*. GDATA. https://www.gdatasoftware.com/blog/2016/04/28234-manamecrypt-a-ransomware-that-takes-a-different-route

[5] Daniel J. Bernstein. 2008. *The Salsa20 Family of Stream Ciphers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 84–97. https://doi.org/10.1007/978-3-540-68351-3_8

[6] Eli Biham and Paul C. Kocher. 1995. A known plaintext attack on the PKZIP stream cipher. In *Fast Software Encryption (Lecture Notes in Computer Science)*, Bart Preneel (Ed.), Vol. 1008. Springer, Berlin, Heidelberg, 144–153. https://doi.org/10.1007/3-540-60590-8_12

[7] BloodDolly and Lawrence Abrams. 2015. *Decryption Guide for TeslaCrypt Encrypted Files*. https://up2sha.re/file?l=C5ag0MrQNqAb.pdf

[8] Bryan Lee. 2017. *Ransomware: Unlocking the lucrative business model*. PaloAlto Unit 42. https://www.paloaltonetworks.com/resources/research/ransomware-report

[9] D. Budgen. 2003. *Software Design*. Pearson/Addison-Wesley.

[10] Bundesamt für Sicherheit in der Informationstechnik. 2018. *BSI TR-02102-1: Kryptographische Verfahren: Empfehlungen und Schlüssellängen*. Technical Report. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile

[11] Check Point Software Technologies. 2016. *Jigsaw Ransomware Decryption*. https://blog.checkpoint.com/2016/07/08/jigsaw-ransomware-decryption/

[12] Lily Chen. 2009. *NIST Special Publication 800-108: Recommendation for Key Derivation Using Pseudorandom Functions (Revised)*. Technical Report. National Institute of Standards and Technology (NIST). https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf

[13] Lily Chen. 2011. *NIST Special Publication 800-56C: Recommendation for Key Derivation through Extraction-then-Expansion*. Technical Report. National Institute of Standards and Technology (NIST). https://doi.org/10.6028/NIST.SP.800-56c

[14] Daniel J. Bernstein. 2005. Salsa20 design. *Department of Mathematics, Statistics, and Computer Science. The University of Illinois at Chicago. Chicago*. https://cr.yp.to/snuffle/design.pdf

[15] M. J. Dworkin. 2001. *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation - Methods and Techniques*. Technical Report. National Institute of Standards and Technology (NIST). https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[16] Claudia Eckert. 2014. *IT-Sicherheit: Konzepte - Verfahren - Protokolle* (9 ed.). De Gruyter Oldenbourg, Berlin. https://doi.org/10.1515/9783486859164

[17] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/2508859.2516693

[18] Fabien Perigaud and Cedric Pernet. 2014. *Bitcrypt broken*. Airbus CyberSecurity. https://airbus-cyber-security.com/bitcrypt-broken/

[19] Fedor Sinitsyn. 2015. TeslaCrypt 2.0 disguised as CryptoWall. https://securelist.com/teslacrypt-2-0-disguised-as-cryptowall/71371/

[20] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. 2015. *Cryptography Engineering*. John Wiley & Sons, Ltd, Hoboken, NJ, USA. https://doi.org/10.1002/9781118722367

[21] International Organization for Standardization. 2010. ISO/TR 14742:2010 Recommendations on cryptographic algorithms and their use. https://www.iso.org/standard/54951.html

[22] Alexander Giehl and Sven Plaga. 2018. Implementing a Performant Security Control for Industrial Ethernet. In *2018 International Conference on Signal Processing and Information Security* (Dubai, United Arab Emirates). IEEE, 4. https://doi.org/10.1109/CSPIS.2018.8642758

[23] Alexander Giehl and Norbert Wiedermann. 2018. Security verification of third party design files in manufacturing. In *10th International Conference on Computer and Automation Engineering Proceedings* (Brisbane, Australia). ACM, New York, NY, USA, 8. https://doi.org/10.1145/3192975.3192984 Best Presentation Award.

[24] Alexander Giehl, Norbert Wiedermann, and Sven Plaga. 2019. A framework to assess impacts of cyber attacks in manufacturing. In *2019 11th International Conference on Computer and Automation Engineering Proceedings* (Perth, Australia). ACM, New York, NY, USA, 8. https://doi.org/10.1145/3313991.3314003

[25] Karsten Hahn. 2016. *The Rise of Low Quality Ransomware*. GDATA. https://www.gdatasoftware.com/blog/2016/09/29157-the-rise-of-low-quality-ransomware

[26] Hasherezade / Malwarebytes Labs. 2016. *Uncovering the secrets of malvertising*. https://www.virusbulletin.com/uploads/pdf/conference_slides/2016/hasherezade-vb-2016-ransomware.pdf

[27] Michael P. Heinl. 2019. *A metric to assess the trustworthiness of certificate authorities*. Master's thesis. University of Ulm. https://doi.org/10.18725/OPARU-12173

[28] Michael P. Heinl, Alexander Giehl, Norbert Wiedermann, Sven Plaga, and Frank Kargl. 2019. MERCAT: A Metric for the Evaluation and Reconsideration of Certificate Authority Trustworthiness. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop* (London, United Kingdom) *(CCSW'19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3338466.3358917

[29] Ben Herzog and Yaniv Balmas. 2016. *Great Crypto Failures*. Technical Report. Check Point Software Technologies. https://blog.checkpoint.com/wp-content/uploads/2016/10/GreatCryptoFailuresWhitepaper_Draft2.pdf

[30] Ben Herzog and Yaniv Balmas. 2016. Great Crypto Failures. https://www.youtube.com/watch?v=loy84K3AJ5Q

[31] Kyung Chul Jeong, Dong Hoon Lee, and Daewan Han. 2012. An Improved Known Plaintext Attack on PKZIP Encryption Algorithm. In *Information security and cryptology - ICISC 2011*, Howon Kim (Ed.). Lecture Notes in Computer Science, Vol. 7259. Springer, Berlin, 235–247. https://doi.org/10.1007/978-3-642-31912-9_16

[32] Josh Reynolds. 2016. CryptXXX Technical Deep Dive. Cisco. https://blogs.cisco.com/security/cryptxxx-technical-deep-dive

[33] Taneli Kaivola, Patrik Nisén, and Antti Nuopponen. 2014. *TorrentLocker Unlocked*. SANS Digital Forensics Incident Response. https://digital-forensics.sans.org/blog/2014/09/09/torrentlocker-unlocked

[34] B. Kaliski. 2000. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898. RFC Editor. http://www.rfc-editor.org/rfc/rfc2898.txt

[35] Lawrence Abrams. 2016. *Decryptor Released for the Nemucod Trojan's .CRYPTED Ransomware*. Bleeping Computer. https://www.bleepingcomputer.com/news/security/decryptor-released-for-the-nemucod-trojans-crypted-ransomware/

[36] Lawrence Abrams. 2016. TeslaCrypt Decrypted_ Flaw in TeslaCrypt allows Victim's to Recover their Files. Bleeping Computer. https://www.bleepingcomputer.com/news/security/teslacrypt-decrypted-flaw-in-teslacrypt-allows-victims-to-recover-their-files/

[37] Lawrence Abrams. 2016. *TeslaCrypt shuts down and Releases Master Decryption Key*. Bleeping Computer. https://www.bleepingcomputer.com/news/security/teslacrypt-shuts-down-and-releases-master-decryption-key/

[38] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why Does Cryptographic Software Fail? A Case Study and Open Problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (Beijing, China) *(APSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 7, 7 pages. https://doi.org/10.1145/2637166.2637237

[39] Lucas Apa and Carlos Mario Penagos Hollman. 2013. Compromising Industrial Facilities from 40 Miles Away. https://media.blackhat.com/us-13/US-13-Apa-Compromising-Industrial-Facilities-From-40-Miles-Away-WP.pdf

[40] Malwarebytes Labs. 2016. *TeleCrypt – the ransomware abusing Telegram API – defeated!* https://blog.malwarebytes.com/threat-analysis/2016/11/telecrypt-the-ransomware-abusing-telegram-api-defeated/

[41] Marc-Etienne M.Léveillé. 2014. TorrentLocker: Ransomware in a country near you. ESET. https://www.welivesecurity.com/wp-content/uploads/2014/12/torrent_locker.pdf

[42] McAfee. 2016. *McAfee Labs Unlocks LeChiffre Ransomware*. https://securingtomorrow.mcafee.com/mcafee-labs/mcafee-labs-unlocks-lechiffre-ransomware/

[43] Microsoft. 2016. *RNGCryptoServiceProvider-Klasse (System.Security.Cryptography)*. https://msdn.microsoft.com/de-de/library/system.security.cryptography.rngcryptoserviceprovider(v=vs.110).aspx

[44] Open Web Application Security Project (OWASP). 2017. *Cryptographic Storage Cheat Sheet*. https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet#Providing_Cryptographic_Functionality

[45] Open Web Application Security Project (OWASP). 2020. *OWASP Top Ten*. https://owasp.org/www-project-top-ten/

[46] Openwall. [n.d.]. php_mt_seed - PHP mt_rand() seed cracker. Openwall. http://www.openwall.com/php_mt_seed/

[47] Oracle. 2014. *Class SecureRandom (Java Platform SE 8)*. https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html

[48] Dorka Palotay. 2016. Ransomware Cryptonite. https://hsbp.org/tiki-download_wiki_attachment.php?attId=207&download=y

[49] Pascal Gadient. 2017. *Security in Android Applications*. Master's Thesis. http://scg.unibe.ch/archive/projects/Gadi17.pdf

[50] Roland Dela Paz. 2016. *Nemucod Adds Ransomware Routine*. Fortinet. https://blog.fortinet.com/2016/03/16/nemucod-adds-ransomware-routine

[51] Proofpoint Staff. 2016. Doh! New "Bart" Ransomware from Threat Actors Spreading Dridex and Locky. Proofpoint. https://www.proofpoint.com/us/threat-insight/post/New-Bart-Ransomware-from-Threat-Actors-Spreading-Dridex-and-Locky

[52] S. Rahaman and D. Yao. 2017. Program Analysis of Cryptographic Implementations for Security. In *2017 IEEE Cybersecurity Development (SecDev)*. 61–68. https://doi.org/10.1109/SecDev.2017.23
[53] ReaQta. 2016. Nemucod meets 7-Zip to launch ransomware attacks. https://reaqta.com/2016/04/nemucod-meets-7zip-to-launch-ransomware/
[54] Bruce Schneier, Matthew Fredrikson, Tadayoshi Kohno, and Thomas Ristenpart. 2015. *Surreptitiously Weakening Cryptographic Systems.* Technical Report. https://eprint.iacr.org/2015/097.pdf
[55] Marc Schönefeld. 2010. *Refactoring of Security Antipatterns in Distributed Java Components.* Ph.D. Dissertation. University of Bamberg. https://fis.uni-bamberg.de/bitstream/uniba/224/2/Dokument_1.pdf
[56] Mark H / Shearwater. 2014. *Cryptodefense infection, some lessons learned.* https://isc.sans.edu/forums/diary/Cryptodefense+infection+some+lessons+learned/18165/
[57] Nigel P. Smart and Rodica Tirtea. 2014. *Algorithms, key sizes and parameters report 2014* (november 2014 ed.). Technical Report. European Network and Information Security Agency (ENISA). https://doi.org/10.2824/36822
[58] Somak Das, Vineet Gopal, Kevin King, and Amruth Venkatraman. 2014. *IV = 0 Security Cryptographic Misuse of Libraries.* Technical Report. MIT. https://courses.csail.mit.edu/6.857/2014/files/18-das-gopal-king-venkatraman-IV-equals-zero-security.pdf
[59] Trend Micro. 2017. *Ransomware - Definition.* https://www.trendmicro.com/vinfo/us/security/definition/ransomware
[60] Meltem Sönmez Turan, Elaine Barker, William Burr, and Lily Chen. 2010. *NIST Special Publication 800-132: Recommendation for Password-Based Key Derivation - Part 1: Storage Applications.* Technical Report. National Institute of Standards and Technology (NIST. https://doi.org/10.6028/NIST.SP.800-132
[61] Utku Sen. 2015. Destroying The Encryption of Hidden Tear Ransomware. https://utkusen.com/blog/destroying-the-encryption-of-hidden-tear-ransomware.html
[62] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS .... In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT '02).* Springer-Verlag, Berlin, Heidelberg, 534–546.
[63] William Brown, Raphael Malveau, Hays McCormick, and Thomas Mowbray. 1998. *AntiPatterns: Refractoring Software, Architectures and Projects in Crisis.* John Wiley & Sons, Ltd. https://doi.org/10.1016/B978-1-85573-259-9.50003-4

A APPENDIX: MITIGATION STRATEGIES

This section gives an overview on mitigation strategies for each AntiPattern. The mitigation strategies are composed of a summary of established best practices in the area of secure implementation of cryptography.

A.1 Clear Text Transmission

The transmission of sensitive information such as passwords or cryptographic keys always has to be encrypted. The implementation can thereby be realized on different levels of the TCP/IP Reference Model. In the tansportation layer, for example the Secure Shell Protocol (SSH) or Transport Layer Security (TLS) can be used in order to secure application protocols such as HTTP by end-to-end encryption [16]. Especially the latter is widely spread and can be implemented by using libraries such as *OpenSSL*. On the network layer, the transmission of data can be protected by employing IPSec's Encapsulating Security Payload (ESP) Protocol which builds a tunnel between communication partners. Similarly to TLS, ESP also provides confidentiality, integrity, and authenticity.

If confidential information only has to be transmitted once, for example the key in case of ransomware, one might want to keep the effort as low as possible and therefore not implement one of the more sophisticated protocols mentioned above. In this case, the key's confidentiality could also be protected by encrypting it before transmission employing asymmetric encryption. This way, only the owner of the private key is able to decrypt the key, although the corresponding public key is hard-coded into the ransomware.

A.2 Weak Keys

It is not advisable to employ self-developed key generation algorithms, especially if constant or deterministic values are used. A much better alternative is to use standardized *key derivation functions* (KDFs) which derive cryptograpic keys of a specific length based on a secret input value [12, 13, 57].

A special version of KDFs are *password-based KDFs* (PBKDFs) which take a user-defined password as well as a *salt* as input values [60]. The salt's purpose is to slow down attacks based on precomputed tables of possible key values. Another way to make such attacks more difficult is to increase the number of iterations during the key derivation. These parameters, however, have to be chosen very carefully since they otherwise do not offer any additional security. The salt should be generated by an RNG instead of using a constant value [17, 60]. RFC 2898 (PKCS #5) [34] defines a minimum length of 64 bit whereas the *National Institute of Standards and Technology* (NIST) recommends 128 bit [60]. Furthermore, the password should be of high entropy and the number of iterations not below 1,000. Alternatively, keys can also be generated by making use of (P)RNGs (see AntiPattern *Poor Seed*).

Various authorities such as the German BSI [10] or NIST [2] constantly publish guidelines containing recommendations about respective key lengths which should be considered during the implementation of cryptographic primitives.

A.3 Poor Seed

Although seeming to be an unremarkable aspect, the generation of cryptographic keys by PRNGs can have a severe impact on the security of the used cryptography and consequentially on the whole product. The ideal solution would therefore be *true RNGs* (TRNG) utilizing physical processes such as thermal resistance noise in order to generate non-deterministic sequences [16]. TRNGs don't need a seed but usually dedicated hardware which regularly renders them unavailable. Another possibility to generate true random numbers are *non-physical non-deterministic RNGs* (NPTRNGs) [10, 16]. The values generated by them are based on intrasystem values such as the content of random access memory (RAM), keyboard and mouse interactions by the user, and audio driver noise. Linux provides such an NPTRNG in form of the device file `/dev/random`. When using NPTRNGs, it has to be ensured that attackers are not able to interact with, i.e. manipulate, it.

In order to use deterministic PRNGs for cryptographic purposes, it has to be initialized with a truly random seed [10, 38]. If there is no TRNG available for the generation of the seed, NPTRNGs can be used as well. Since `/dev/random` is blocking if there is not enough entropy available, the generation of random values can be heavily delayed [10]. Therefore, the combination of letting NPTRNGs generate a seed of suffcient length which is then fed to a PRNG can result in a better output rate. The German BSI recommends an entropy of n bit in order to ensure a security of n bit [10].

As demonstrated by the formerly mentioned ransomware examples, a PRNG's seed can partially be determined by the sequences generated by the PRNG. Hence, non-secret values such as IVs for block ciphers should be generated independently from cryptographic keys by initializing the PRNG with a new, truly random seed each time.

Several programming languages and APIs do not only offer PRNGs but also *cryptographically secure PRNGs* (CSPRNGs) such as Java's `SecureRandom` [47] or Microsoft's `RNGCryptoService-Provider` [43].

A.4 Key Storage

Cryptographic keys shall under no circumstances be stored in clear text. Key files therefore always have to be encrypted. The needed key can be derived from a password, for example by utilizing a hash function or PBKDF as described in subsection A.2. This approach is for example employed by *Pretty Good Privacy* (PGP) in order to protect the private key needed to decrypt email messages [16]. Furthermore, the key file should only be made accessible for legitimate users by appropriately defined file system permissions.

Another valid possibility to securely store keys are so-called *hardware security modules* (HSMs), such as smart cards or USB tokens [16, 20], where the key is stored in a read-only memory (ROM) and additionally protected by a personal identification number (PIN). Attacks against such mobile media are usually quite expensive.

A similar solution are *trusted platform modules* (TPMs) which are chips persistently connected with a computer's hardware and able to generate and store keys [16]. Microsoft's *BitLocker* for example utilizes TPMs to securely store keys used for full disk encryption.

A.5 Hard-Coded Keys

The security of a crypto system depends on the confidentiality of the used key(s). Therefore, no (private) keys and in general any private information must be hard-coded into the source code since hard-coded information can easily be extracted by attackers. It is recommended to generate the required keys on the target system instead. This also prevents the same key from being used for all copies of the software. It is also paramount to securely store the generated keys after its generation.

If a shared key is required, e.g., for MACs, the keys can be negotiated via key exchange protocols, e.g., Diffie-Hellman. Alternatively, the key can be generated by one of the communication parties, encrypted via an asymmetric crypto system, and then distributed to the other party. Public keys can also be distributed via certificates, allowing attestation of its owning entity's identity and also guaranteeing its integrity [3].

A.6 Erroneous Library Usage

Since the usage of cryptographic libraries can lead to errors, special care must be taken during their implementation. The documentation should be studied *prior* to their usage and especially the choice of parameters, values, and flags has to be considered carefully. Code samples provided within the documentation should only be used for orientation as they may lead to erroneous configurations of a cipher. Especially for block ciphers, errors can occur more easily. The EBC mode, for example, should only be used for plaintexts *shorter* than the block length of the employed cipher [16, 17].

The choice of an initial vector is also critical as each mode has different requirements concerning the vector. For the CBC mode, no predictable or constant IVs must be used as otherwise attacks on the used cipher are possible [17, 52]. In OFB mode, IVs must only be used exactly once [15, 20] and also in CFB mode, they must not repeat [10, 20, 58]. Hence, in order to improve the security of cryptographic primitives, IVs must be generated randomly, especially for ciphers in CBC mode [10, 52, 58]. Studying the cipher or mode prior to implementation is highly recommended. Especially default configurations provided by the libraries should not be assumed secure a priori [58].

A.7 Weak Algorithms

Only ciphers that are verified by experts and considered secure should be used. Ciphers proven to be vulnerable must not be used under any circumstances. It is worth noting that the definition of a *secure* cipher can change over time. Therefore, guidelines and recommendations such as *NIST SP 800-57* [2], *ISO TR 14742* [21], or publications by authorities such as the German BSI [10] or ENISA [57] which discuss the security of ciphers should be regularly consulted in order to be aware of updates.

A.8 Self-Implementation

Self-implementation of cryptogrpahic ciphers has to be avoided at any time regardless how simple this may appear for experienced developers. Instead, often used and regularly inspected implementations of ciphers, for example in the form of cryptographic libraries, have to be used [44]. If the self-implementation of a cipher cannot be avoided, rigorous testing of the implementation is necessary. For this, test vectors provided by experts or official institutions have to be used [38].

Under no circumstances should self-designed algorithms be used. It is crucial to follow the principle *Never roll your own cryptography*. A variety of tested and reliably secure ciphers are publicly available and only those must be used, preferably in the form of tested implementations.