SoK: Use of Cryptography in Malware Obfuscation

Hassan Jameel Asghar
Macquarie University
Australia
Email: hassan.asghar@mq.edu.au

Benjamin Zi Hao Zhao
Macquarie University
Australia
Email: ben_zi.zhao@mq.edu.au

Muhammad Ikram
Macquarie University
Australia
Email: muhammad.ikram@mq.edu.au

Giang Nguyen
Macquarie University
Australia
Email: duclinhgiang.nguyen@hdr.mq.edu.au

Dali Kaafar
Macquarie University
Australia
Email: dali.kaafar@mq.edu.au

Sean Lamont
and Daniel Coscia
Defence Science and Technology Group
Australia
Email: sean.lamont2@dst.defence.gov.au
daniel.coscia1@dst.defence.gov.au

***Abstract*—We look at the use of cryptography to obfuscate malware. Most surveys on malware obfuscation only discuss simple encryption techniques (e.g., XOR encryption), which are easy to defeat (in principle), since the decryption algorithm and the key is shipped within the program. This SoK proposes a principled definition of malware obfuscation, and categorises instances of malware obfuscation that use cryptographic tools into those which evade detection and those which are detectable. The SoK first examines easily detectable schemes such as string encryption, class encryption and XOR encoding, found in most obfuscated malware. It then details schemes that can be shown to be hard to break, such as the use of environmental keying. We also analyse formal cryptographic obfuscation, i.e., the notions of indistinguishability and virtual black box obfuscation, from the lens of our proposed model on malware obfuscation.**

1. Introduction

Malware developers often obfuscate their programs in the hope that the program goes undetected by malware detectors, e.g., antivirus software. Many researchers have documented obfuscation techniques used in malware in the wild [1], [2], [3]. Of particular interest is the use of cryptographic tools to obfuscate malware, e.g., encrypting parts of the program via a block cipher. Use of cryptography may give the impression that the said technique is provably resistant to de-obfuscation. However, as we shall see shortly, most use of cryptography by malware authors to obfuscate their programs amounts to simple encoding. On the other hand some techniques can be shown to require substantial computing power to de-obfuscate [4]. A key question to ask is how do we evaluate if a malware obfuscation technique is resistant to de-obfuscation or not? In this survey, we propose a definition of malware obfuscation, and then categorise malware obfuscation techniques, used by malware in the wild as well as proposed by malware analysts and computer security researchers, under this framework into those that are easily detectable versus those that avoid detection.

The reader may find some similarity of the topic to cryptographic program obfuscation [5]. Although cryptographic program obfuscation can be used in malware obfuscation, it is helpful to distinguish between the goals of malware obfuscation and that of program obfuscation in general. In case of the latter, given a program P, the goal is to create an obfuscated program $\hat{P}$ which is functionally equivalent to P but which is harder to reverse-engineer [6]. The programmer therefore wishes to hide how the program is implemented. On the other hand, the goal of malware obfuscation is to avoid being detected, and therefore labelled as malware, before it has run its functionality at least once on a target machine. Thus, simply obfuscating the program will not necessarily meet the goals of malware obfuscation, as the resulting program, by virtue of being functionally equivalent to the original program, can still be detected as malware through its input/output behavior, e.g., via dynamic analysis. Having said that, often times a malware author simply aims to avoid detection via inspection of the program code, i.e., via static analysis only.

Perhaps the most common example of the use of cryptography in obfuscation is *string obfuscation* [2]. Under this type of obfuscation, strings such as URLs, paths and constants are encrypted using a cryptographic cipher such as AES or DES [2]. Through string obfuscation, the malware author may wish to hide URLs which may have been flagged as malicious by anti-malware programs. Note that regardless of the security of the underlying cipher, e.g., DES vs AES, such obfuscation can in principle be undone. This is because the decryption logic of the program, including the decryption key, has to be provided within the application [2], [7]. Thus, such strings can be de-obfuscated by a sufficiently sophisticated de-obfuscation tool or a skilled programmer using, for example, dynamic analysis.

One of the goals of this paper is to decouple the use of cryptographic tools in the manner exemplified in string obfuscation, which essentially amounts to *security by obscurity*, versus a more principled approach whereby robustness against de-obfuscation is guaranteed not merely because the detector was unable to locate the decryption/decoding routine within the program. At the outset we need to be clear about the opposing goals of malware obfuscation and detection. The goal of a malware author is to ensure that his/her program runs on the target machine. For this to happen, the malware should "trick" any anti-malware software running on the target system to believe it is a benign program. Obfuscation, then, is one of the techniques towards that end. On the other hand, malware defense would like to avoid such mistakes. On the precautionary side, we could label any use of obfuscation in a program as indicative of malicious intent. However, the fact that obfuscation is used by benign programs as well [2], e.g., to protect intellectual property, shows that this will inevitably block the execution of benign programs. This shows that we cannot simply label a program malware on the basis of obfuscation alone. A good detector should be able to distinguish between an obfuscation of a benign program versus that of a malware.

Our Contributions. We give a formal model of malware obfuscation in Section 2, where we define obfuscation to be successful if it degrades the combined false positive and negative rates of a malware detector. Under this formal model, we show that evasion is not possible if the obfuscated program is functionally equivalent to the original program. We then relax the requirement of functional equivalence to requiring the obfuscated program to run on selected target machines. We discuss few of the prominent techniques from cryptography used in malware obfuscation, but which do not evade detection under our model in Section 3. We then give a description of an obfuscator which provably evades detection under our model in Section 4, but which is not useful in the sense that the program's functionality may not even run on the target machine. We analyse a real-world alternative of such a scheme based on environmental keying [8] from the lens of our model in Section 5. In Section 6, we discuss the use of deniable encryption in malware obfuscation and its shortcomings. Section 7 discusses cryptographic notions of obfuscation, such as indistinguishability obfuscation [5], and its relation to malware obfuscation defined in this paper. We analyse the prevalence of cryptographic malware obfuscation in the real-world in Section 8, and discuss related work in Section 9. Finally, we discuss some limitations of our treatise and avenues for future work in Section 10.

2. Formal Model and Implications

A program P is modelled as a probabilistic polynomial time Turing (PPT) machine. We consider a set S of programs with two disjoint subsets: Malware and Benign. We assume the two subsets to be mutually exclusive. We shall often use M to denote a generic malware, i.e., a member of Malware. We assume that S is sampled from a joint probability distribution $\mathcal{D}$ of programs and their labels (malware or benign). We assume that the set S is of polynomial size. Thus, the set is not supposed to capture all possible benign and malicious programs, but rather a representative sample, on which we can test the capabilities of obfuscators and detectors. A malware detector D, is a program, which takes as input a program P, and outputs 1 if it is malware and 0 otherwise. The type-I and type-II errors (false positives and false negatives, respectively), associated with D are defined as:

$$\begin{aligned} \alpha_D &= \Pr[D(P) = 1 \mid P \in \mathsf{Benign}], \\ \beta_D &= \Pr[D(P) = 0 \mid P \in \mathsf{Malware}]. \end{aligned} \tag{1}$$

Here the probability is over the distribution $\mathcal{D}$ and any randomness employed by the detector D.[1] Note that probability over the distribution $\mathcal{D}$ means that we sample a program uniformly at random from S to calculate the two errors. When we talk about the total error or simply the error of a detector D, we mean the quantity $\alpha_D + \beta_D$. Without loss of generality, we can assume that $\alpha_D + \beta_D \leq 1$ [9]. Because otherwise we have $\alpha_D + \beta_D > 1$, and we can instead use a detector which flips the output of D. The error of this detector is given by:

$$1 - \alpha_D + 1 - \beta_D = 2 - (\alpha_D + \beta_D) < 1.$$

We shall call D trivial if it makes its decisions independent of the program, i.e., benign or malware.

Proposition 1. *We have*

1) *If D is trivial then $\alpha_D + \beta_D = 1$.*
2) *Conversely, if for any D, $\alpha_D + \beta_D = 1$, then there exists a trivial D' such that $\alpha_{D'} = \alpha_D$ and $\beta_{D'} = \beta_D$.*

Proof. For part (1), assume D is trivial. Let $P \in S$. Then:

$$\begin{aligned} 1 &= \Pr[D(P) = 1] + \Pr[D(P) = 0] \\ &= \Pr[D(P) = 1 \mid P \in \mathsf{Benign}] + \Pr[D(P) = 0 \mid P \in \mathsf{Malware}] \\ &= \alpha_D + \beta_D. \end{aligned}$$

For part (2), let D' be a detector which with probability α_D outputs 1, else outputs 0. Then

$$\alpha_{D'} = \Pr[D'(P) = 1 \mid P \in \mathsf{Benign}] = \Pr[D'(P) = 1] = \alpha_D,$$

and

$$\begin{aligned} \beta_{D'} &= \Pr[D'(P) = 0 \mid P \in \mathsf{Malware}] = \Pr[D'(P) = 0] \\ &= 1 - \alpha_D = \beta_D, \end{aligned}$$

as required. □

In light of the proposition, we shall call any D with $\alpha_D + \beta_D = 1$ as being trivial.

Corollary 1. *If D is non-trivial then $\alpha_D + \beta_D < 1$.* □

1. The notation $D(P)$ covers both static and dynamic analysis. In the former, the detector only makes its decision based on taking the description of the program P as a string. In the latter, the program can run P as a subroutine, feeding it with required inputs. This engulfs sandboxed environments.

In particular, any D is trivial if $\alpha_D = 1$ or $\beta_D = 1$. For instance, if S is a set of all programs that use an encryption algorithm, and D is the malware detector that outputs 1 if a program uses an encryption algorithm. Then we have $\alpha_D = 1$, even though $\beta_D = 0$. This is exactly the example we highlighted in the preamble. The case when $\alpha_D = \beta_D = 0$ for all possible sets of programs S is not possible since malware detection is undecidable [10], [11]. Thus, in practice, any D is expected to give a tradeoff. We shall assume that there is at least one non-trivial detector D for the set S of programs. Otherwise, obfuscation is pointless.

Remark 1. *As mentioned, the programs D and P are PPT algorithms. This means that the size of P is bounded by a polynomial. Also, if D simply runs P as a subroutine, it can only evaluate P on polynomially many inputs from a possibly larger space.*

Functional Equivalence. We say that two programs P_1 and P_2 are functionally equivalent if for all inputs x, we have $P_1(x) = P_2(x)$. Otherwise they are functionally inequivalent. We assume that any benign program is functionally inequivalent to a malware and vice versa. We also assume that functional inequivalence can be checked in polynomial time. That is, given a polynomial number of inputs, one can check whether two programs are functionally equivalent or not. In case of a pair of functionally inequivalent programs, one obtains at least one input-output pair in polynomial time in which their output differs. On the other hand, the benign programs (resp., malware) in S may be functionally equivalent to one another.

Malware Obfuscation. A program obfuscator $\mathcal{O}$ is a PPT compiler that takes as input a program P and outputs a program P'. We call P' the obfuscation of P under $\mathcal{O}$.

Definition 1. *Let $\mathcal{O}$ be a non-trivial malware obfuscator. Let D be a malware detector. Define:*

$$\beta_D^{\mathcal{O}} = \Pr[D(M') = 0 \mid M' \leftarrow \mathcal{O}(M), M \in \mathsf{Malware}]$$

and,

$$\alpha_D^{\mathcal{O}} = \Pr[D(P') = 1 \mid P' \leftarrow \mathcal{O}(P), P \in \mathsf{Benign}]$$

We say that $\mathcal{O}$ evades detection from the malware detector D, if $\alpha_D^{\mathcal{O}} \leq \alpha_D$ implies $\beta_D^{\mathcal{O}} > \beta_D$. □

Trivial Obfuscators and Naive Detectors. The above definition is lenient: it calls the obfuscator a successful one if it evades detection, on average. Thus, for some subsets of malware it might perform worse. If it performs worse for all malware, then obviously the said technique is useless. The definition weeds out the trivial "identity" obfuscator, which simply prints out its input program as the output. This obfuscator will necessarily have $\alpha_D^{\mathcal{O}} = \alpha_D$ and $\beta_D^{\mathcal{O}} = \beta_D$, and hence it does not evade detection. More importantly, the definition penalises any brute-force or "dumb" way of detecting malware, by stating that there will necessarily be a tradeoff in the form of increased false positives. Consider for instance the set S of all programs that do not use a cryptographic library. Let $\mathcal{O}$ be an obfuscator, that encrypts some components of the program. Let D_2 be the following malware detector that uses the (non-trivial) malware obfuscator D as a subroutine:

```
$D_2(P)$:
if $P$ imports a cryptographic library then
    Output 1
else
    $D(P)$
end if
```

Clearly, we have $0 = \beta_{D_2}^{\mathcal{O}} \leq \beta_{D_2}$. However, we also have $1 = \alpha_{D_2}^{\mathcal{O}} > \alpha_{D_2}$, since D is non-trivial and before obfuscation, none of the programs import a cryptographic library. Thus, this obfuscator evades detection against D_2 under our definition (as it should). Another detector in this line of detectors is the one that flags any use of obfuscation as an indicator that the program may be malicious. This may very well be the case in the real-world: obfuscation may be more prevalent in malware than benign programs. However, since we are interested in the strength of the obfuscation technique, and the capabilities of any detector against an obfuscation technique, we have modelled our definition as a challenge to the detector to distinguish the same technique applied to a malware versus benign program. Furthermore, benign programs also use obfuscation techniques for a variety of reasons such as protection of intellectual property [2], [12]. Definition 1 is formed after the definition of the ideal distinguisher in [13].

Utility and Functional Equivalence. From a malware author's point of view, the above definition does not say anything about the utility of the obfuscated malware M'. For instance, the obsfuscator could simply remove all "malicious functionality" in M, and therefore (rightfully) achieve a higher error rate $\beta_D^{\mathcal{O}}$, saying nothing about the detection capabilities of D. On the one extreme, we may have functional equivalence. That is, $M(x) = M'(x)$ for all inputs x. However, this requirement means that there is no obfuscator which can evade detection from a malware detector which predominantly bases its decision on blackbox input-output behaviour (e.g., dynamic analysis). We prove this in the following.

Proposition 2. *Let $\mathcal{O}$ be an obfuscator that preserves functionality. Then there exists a non-trivial detector such that $\mathcal{O}$ does not evade detection from it.*

Proof. Let D be a non-trivial detector for S guaranteed by assumption. This means that $\alpha_D + \beta_D < 1$. The idea is to construct a detector D' that runs D over the (unobfuscated) programs in S, marks its decisions, and then creates input-output fingerprints for each program in S. After obfuscation is applied, since the programs remain functionally equivalent, D' will be able to identify them through the input-output fingerprints, and hence retain the error rate of D (as the labels are from D). The only subtlety is if some programs in S are functionally equivalent, which we address in the following.

We construct a detector D' from the set S of programs and labels, and the detector D as follows. For each program in S, D' runs D, and stores the label output by D. Next, at each round, D' chooses as input $\{0,1\}^q$, where q is a polynomial. D' then runs each program in S on this input, storing the respective outputs.[2] After at most polynomially many inputs Q, the sequence of input-output pairs of each program labelled as benign is different from the sequence of input-output pairs of each program labelled as malware, according to the assumption on functional inequivalence. However, as noted before there may be some benign programs that are functionally equivalent to other benign programs. The same goes for malware programs.

As long as D labels them uniformly, there is no issue. However, if D labels a subset of programs with the same input-output sequence differently, then D' will not be able to distinguish between their obfuscated versions given only input-output sequences. Let $S_1, S_2, \ldots, S_t$ be subsets of benign programs from S, such that each program in S_i is functionally equivalent, and for each i, not all programs in S_i have the same label under D. Take one such S_i. Since each program in S_i is benign, let $j(i)$ be the number of programs in S_i mislabelled by D as malware. Similarly we have a collection of subsets of malware programs such that within each subset they are functionally equivalent, but D mislabels at least one of them as benign. Due to symmetry, the ensuing analysis is applicable to this case as well. Therefore, without loss of generality, we use the benign case only. Let $S(t) = S_1 \cup S_2 \cup \cdots \cup S_t$.

After the programs in S are run through $\mathcal{O}$, our detector D' does as follows. Given any program P, it runs the program on the inputs from Q. If the input-output sequence identifies it as a program not in $S(t)$, D' simply outputs the label previously stored from D. Otherwise, it identifies the set S_i such that $P \in S_i$. Note that the programs in different S_i's have different input-output "fingerprints." After this, D' returns the label 1 (malware) with probability $j(i)/|S_i|$, else it outputs the label 0 (benign). We now calculate the type-I errors of the two detectors. Since D is run once by D' (before obfuscation), let us calculate its *empirical* type-I error α_D^{Emp} defined as:

$$\begin{aligned}\alpha_D^{\text{Emp}} &= \frac{|D(P)=1 \mid P \in \mathsf{Benign}|}{|\mathsf{Benign}|}\\ &= \frac{|D(P)=1 \mid P \in \mathsf{Benign}, P \notin S(t)|}{|\mathsf{Benign}|}\\ &+ \frac{|D(P)=1 \mid P \in \mathsf{Benign}, P \in S(t)|}{|\mathsf{Benign}|}\\ &= \frac{j + \sum_{i=1}^{t} j(i)}{|\mathsf{Benign}|},\end{aligned}$$

where j is the number of benign programs not in $S(t)$, mislabelled by D. Now, for D' we have:

$$\begin{aligned}\alpha_{D'}^{\mathcal{O}} &= \Pr[D'(\mathcal{O}(P)) = 1 \mid P \in \mathsf{Benign}]\\ &= \Pr[D'(\mathcal{O}(P)) = 1 \mid P \in \mathsf{Benign}, P \notin S(t)]\\ &\times \Pr[P \in \mathsf{Benign}, P \notin S(t)]\\ &+ \Pr[D'(\mathcal{O}(P)) = 1 \mid P \in \mathsf{Benign}, P \in S(t)]\\ &\times \Pr[P \in \mathsf{Benign}, P \in S(t)]\\ &= \frac{|D(P)=1 \mid P \in \mathsf{Benign}, P \notin S(t)|}{|\mathsf{Benign}| - |S(t)|} \cdot \frac{|\mathsf{Benign}| - |S(t)|}{|\mathsf{Benign}|}\\ &+ \sum_{i=1}^{t} \frac{|S_i|}{|\mathsf{Benign}|} \cdot \frac{j(i)}{|S_i|}\\ &= \frac{j}{|\mathsf{Benign}|} + \frac{\sum_{i=1}^{t} j(i)}{|\mathsf{Benign}|} = \alpha_D^{\text{Emp}}\end{aligned}$$

Thus, D' reproduces D's type-I error. A similar analysis holds for type-II error, i.e., $\beta_{D'}^{\mathcal{O}}$. Hence, we conclude that D' is non-trivial and hence $\mathcal{O}$ cannot evade detection from it.

□

Note that the proof above uses the assumption of being able to check functional inequivalence in polynomial time. This is needed to check input-output behaviour on the same set S of programs before and after obfuscation. If functional inequivalence cannot be determined in polynomial time, then it can also not be determined in polynomial time after obfuscation is applied. In other words, the detector will make the error anyway, on the first pass. In light of the above proposition, we have a milder (and realistic) requirement that there is some target subset X of inputs x, for which we have $M'(x) = M(x)$. Given this we have the following definition of utility.

Definition 2 (Utility)**.** *For a program P, let $X(P)$ denote a target set of inputs; a subset of the domain of P. An obfuscator is useful if for all programs P and target input sets $X(P)$, we have $P'(x) = P(x)$ for all $x \in X(P)$. If $X(P)$ is exactly the input domain of P, we say that the obfuscator preserves functionality.*

The above definition means that for some inputs (not in the target set), the obfuscated malware M' might not even behave maliciously. This naturally models the use of environmental variables to identify target machines on which the malware is supposed to run (as opposed to every machine) [8]. We will discuss this technique in Section 5.

On the Obfuscator. The obfuscator itself is benign, in the sense that if we run the obfuscator on any benign program, the resulting obfuscation is not malicious (in the global sense, irrespective of the point of view of any detector). The utility aspect is important, because in principle we could write a malware which evades detection against any (polynomial-time) detector as we shall show in Section 4. This essentially means that the malware may never run its functionality on its target.

2. If the programs are probabilistic, we assume that the input contains the input to the random tape, i.e., coin tosses [14, §7.1].

3. Non-Evasive Obfuscation Techniques

We first summarise some techniques that do not evade detection in our model. The main reason being that the decryption routine is shipped with the program. The obfuscation technique is simple: encrypt part(s) of the malware using an encryption algorithm. The encrypted components are decrypted on run-time. Thus, such obfuscation (in principle) is detectable both via static and dynamic analysis, since detection amounts to finding the decryption routine within the program. In our model, this would mean that the error of the detector will remain the same after such obfuscation is applied. Since these are the predominant techniques used by real-world malware authors [3], we nonetheless enlist some of them. Example usage of these techniques are in string encryption [2], [7]. Here, only the string (such as a URL) is encrypted. Once again, the decryption key is provided within the program so that the string can be decrypted at run-time. Similarly, they can be used in class encryption: e.g., DEX file encryption [7]: this again suffers from the same aforementioned problem. These techniques in essence are similar to the use of "packing" in malware, which uses compression to evade detection. Needless to say that decompression is done at run-time.

3.1. Base-64 Encoding

Base64 encoding is an encoding scheme that converts binary data into text. A non-malicious use of Base64 encoding is sending images over email. Base64 encoding is also frequently used to obfuscate malware, e.g., to obfuscate file names and file content [15]. This is related to cryptography only at a rudimentary level, as Base64 encoded text, once detected, can readily be decoded. This is obvious, since this is merely an encoding scheme and not an encryption scheme. The following is an example of Base64 encoding in JavaScript taken from [15]:

```
this.jdField_b_String
= a(b("L1RodW1icy5kYg=="));
```

Here, `b()` Base64 decodes the input string, and `a()` reads the resource. The decoded string is `Thumbs.db` which itself contains Base64 encoded malicious data [15].

3.2. XOR Obfuscation

The underlying idea behind XOR obfuscation is its use in one-time pad. Given a plaintext represented in bits, if it is XORed with a random key of equal length, then the resulting encryption is perfectly secure. If the plaintext represents a malware, then one can encrypt the malware by using a key of equivalent length. However, this requires huge key sizes. As a result, malware obfuscators normally reuse a short key, e.g., a single byte, and encrypt equivalent sized blocks. The use of a single byte is known as single-byte XOR encoding [16, §13]. Due to short keys used in XOR encoding based obfuscation, there are various tools that can deobfuscate the program or find the key. These tools look for expected text at a given location in the program, in an attempt to find the key (e.g., PE files) [16, §13]. Finding the key does not mean that these detectors necessarily try to look for the decryption routine within the malware to find the key [16, §13]. For a list of few other notable schemes that belong to the this category of encryption using short keys, we refer the reader to [16, §13].

3.3. Obfuscation with Stronger Encryption

Many obfuscation tools also offer obfuscation via block ciphers such as AES, DES and TEA [1], [2]. However, even though block ciphers such as AES in an appropriate mode can provide strong encryption, their use in malware obfuscation is rudimentary in our model. The decryption routine together with the key is normally part of the code which obfuscates the code [2], [17]. Note that these ciphers are also employed to encrypt traffic between the malware and a remote server (command and control traffic) [1]. But once again since the decryption keys are hardcoded, the traffic can be decrypted to see what information is being exchanged [17]. One could also use asymmetric encryption algorithms, e.g., RSA, to generate session keys to encrypt communication as in the case of the Waledac malware family [17]. The RSA public-private key is generated at run-time by the malware, and hence the private key is also readily available to the detector (via dynamic analysis). For a detailed account of block ciphers used by commercial obfuscation tools to obfuscate Android and Java apps in the manner described above, please see [2].

3.4. Use of Hash Functions

At first glance, it is not clear how cryptographic hash functions could be used in obfuscation, as their one-wayness property implies that deobfucation would be impossible even for the malware itself. The use of hash functions such as MD5 and SHA in malware have been documented for generating unique identities of command and control bots such as in the Waledac malware family [1], [2]. This is obviously not an application of obfuscation. However, as we show next, one can use hash functions to construct a scheme that provably evades detection in our model, albeit with an unavoidable utility tradeoff.

4. Towards Evasive Techniques: The Hash-then-Decrypt Construct

We now turn our attention to obfuscation techniques that evade detection (under our model). The construct given in this section is derived from a similar construct based on environmental variables which first appears in [8]. Let Enc and Dec be the encryption and decryption functions of a semantically secure symmetric key encryption scheme. Let H be a cryptographic hash function. Given the program P and its block B, to be obfuscated, the obfuscator $\mathcal{O}$ does the following:

$$k \leftarrow \text{random key}, [\![B]\!] \leftarrow \text{Enc}_k(B),\ h_k \leftarrow H(k),$$

where the key is selected from the key space of the symmetric encryption scheme. The obfuscator then creates the program P', which is the same as P except that B is replaced by the block:

```
$x \leftarrow$ KeyFinder()
**if** $H(x) = h_k$ **then**
    $B' \leftarrow \text{Dec}_x([\![B]\!])$
    $B'$
**end if**
```

Here the routine KeyFinder() simply samples a random key from the key space. We now discuss the security and utility of this obfuscator.

Proposition 3. *If the symmetric key cryptosystem is semantically secure, then the random key-based hash-then-decrypt obfuscator $\mathcal{O}$ evades detection against any detector D in the random oracle model.*

Proof. Let S be a set containing two programs: one benign and other malware, denoted P and M, respectively. We assume that P and M are identical except for the blocks B_P and B_M in P and M, respectively. This implies that M is benign if the block B_M is replaced with B_P. Let D be a non-trivial detector achieving errors α_D and β_D, respectively, with both strictly less than 1. Let $\mathcal{O}$ denote the above obfuscator. Let $\mathcal{A}$ denote the semantic security adversary. Adversary $\mathcal{A}$ chooses $m_0 = B_P$ and $m_1 = B_M$ as its choices of the two messages.[3] When $\mathcal{A}$ receives $[\![m_b]\!]$, it inserts this as $[\![B]\!]$ in the program. Adversary $\mathcal{A}$ then gives the resulting program P' to D. Whenever D makes a hash query, the adversary samples a uniform random string and gives it to D. For repeat hash queries, the adversary returns the same random string as previously sent. Let b' be the bit returned by D. Adversary $\mathcal{A}$ outputs b' as its guess. It is clear that the adversary perfectly simulates the obfuscation except for negligible probability (e.g., in case D finds a collision). Now, assume that the obfuscator does not evade detection against D. This means that from Definition 1, we have simultaneously:

$$\alpha_D^{\mathcal{O}} \leq \alpha_D \text{ and } \beta_D^{\mathcal{O}} \leq \beta_D.$$

Then, except for negligible probability, we have:

$$\begin{aligned} \Pr[b' = b] &= \Pr[b' = 1 \mid b = 1] \Pr[b = 1] \\ &\quad + \Pr[b' = 0 \mid b = 0] \Pr[b = 0] \\ &= (1 - \alpha_D^{\mathcal{O}}) \cdot \frac{1}{2} + (1 - \beta_D^{\mathcal{O}}) \cdot \frac{1}{2} \\ &\geq (1 - \alpha_D) \cdot \frac{1}{2} + (1 - \beta_D) \cdot \frac{1}{2} \\ &= 1 - \frac{1}{2}(\alpha_D + \beta_D) > \frac{1}{2}, \end{aligned}$$

where the last step follows due to the non-triviality of D, i.e., $\alpha_D + \beta_D < 1$ (Corollary 1). This is clearly a contradiction, as $\mathcal{A}$ breaks the semantic security of the cryptosystem. □

3. We assume they are of the same length. Otherwise, we pad the shorter program with more benign code, e.g., print statements.

Unfortunately (or fortunately), this obfuscator is not useful.

Proposition 4. *The random key-based hash-then-decrypt obfuscator $\mathcal{O}$ is not useful.*

Proof. Pick any input $x \in X(M)$ in the target set of inputs. On this input, necessarily (unobfuscated) M runs the malicious block B (otherwise it is not the target set of inputs). On the other hand, on the same input, the probability that the obfuscated version M' will run the block B is negligible (in the size of the key space). Through the union bound, over polynomially many inputs in $X(M)$, the probability that M' will run the block B remains negligible. Hence with overwhelming probability $M'(x) \neq M(x)$ for all inputs in $X(M)$, and hence the malware obfuscator is not useful. □

5. Environmental Keying

While the obfuscated malware construction in Section 4 is not useful (according to our definition), variants of it have appeared in theory and practice. On the one extreme, we have the cryptographic key uniformly at random selected from a key space, which means that the decryption routine cannot be run in feasible time, by the malware as well as the detector. On the other extreme, the malware writer could provide the key within the obfuscated program. However, under our model, this technique is not secure, as it is a matter of time before some detector will be able to detect keys.[4] In between these two extremes we have a spectrum of difficulties:

1) Use of a random cryptographic key so that the KeyFinder routine just takes long enough to evade dynamic analysis [18]. This means use of smaller key-sizes than is considered cryptographically secure. This also includes use of other time-consuming options, e.g., hashing multiple times [18]. However, for the obfuscated malware to be useful, the KeyFinder should output the key in feasible time. Hence, this technique does not evade detection as the correct key will be found on all machines (the obfuscated malware is functionally equivalent).
2) Instead of embedding it in the hash, the key can be retrieved from an external source, e.g., a server [16], [18], [19]. However, the link to the server itself may disclose maliciousness, e.g., a known malicious domain. On the other hand, there are examples of malware who retrieve keys from trusted hosts, such as GitHub, Dropbox, and Google Docs, thus evading such reputation-based detection [20], [21], [22]). Furthermore, a growing trend in malware is communicating over TLS/SSL with remote command-and-control hosts [21]. Encrypted communication could also be used to download

4. This is one of the reasons why XOR-based obfuscation techniques are easily detected and de-obfuscated by common program analysis tools.

part of the malware hosted on mainstream cloud platforms, such as GitHub, which may not raise suspicion [22]. However, if a trusted host is used to host the key and the key is communicated to the malware via an encrypted channel, the detector will still be able to label the program as malware using dynamic analysis, as it will fetch the key on any machine. Thus, such techniques only delay the inevitable.

3) By far the most resilient technique is environmental keying [8]. Instead of brute-forcing the key, the program constructs the key from the environment variables (e.g., user name). The main point being that the malware would run (after a mini-bruteforce search) on target computers (whose environment variable values are known beforehand). However, on non-target machines, the KeyFinder routine may not terminate as the environment variables never take on the pre-determined values "hidden" within the hash digest.

The technique of environmental keying has been known at least since 1998 [8]. We present a detailed account of this technique in the hash-then-decrypt construct in the related work section (Section 9). Here we look at the use of environmental keying with the help of an example.

5.1. The Ebowla Framework: Case Study

We consider the Ebowla framework [23] for encrypting malware payloads via environmental keying which has been presented at several white hat offensive security conferences. As mentioned earlier, environmental keying uses values stored in "environment variables" to derive the key. The idea being that the attacker knows the values taken by these environment variables in his/her target machines, and hence reproducing the key will be faster on the target machines versus other machines (including those employed by malware detectors as sandbox environments). The framework describes the following environment variables, although it leaves room for writers to define more variables:

- Environmental variables: e.g., username, user domain, computer name, and number of processors.
- Path variables: e.g., `C:\Windows\temp`. This includes a starting location, e.g., `C:\Windows`.
- External IP ranges, e.g., `100.0.0.0`.
- System time range with the grarnualrity of year, month or a particular day, e.g., 20210000, 20211200, 20211201.

One or more of these variables, called tokens, are then used to create the key k in the hash-then-decrypt construct. An important consideration here is how much logic is pre-built in the hash checking part to ensure a reasonable tradeoff between key reconstruction by the malware, versus by the detector. For instance, if the key is derived by constructing one variable each from the above mentioned variables, and the hash checking routine constructs values for these tokens in sequence, then this reduces the *entropy* of the key space, versus if the hash checking routine needs to check all possible combinations (order of concatenation). We shall return to the entropy of the key space in environmental keying shortly. For now, we focus on the particular version of hash-then-decrypt construct, employed in the Ebowla framework. The obfuscation routine is as follows:

$$k \leftarrow H(\text{environmental key}), [\![B]\!] \leftarrow \text{Enc}_k(B),\ h_B \leftarrow H(B),$$

where as before we assume that B is the malicious block that needs to be obfuscated (the so-called payload). The obfuscated program P' then contains the following block instead of B:

```
$x \leftarrow$ KeyFinder(environment variables)
$B' \leftarrow \text{Dec}_x([\![B]\!])$
**if** $H(B') = h_B$ **then**
    $B'$
**end if**
```

where

```
KeyFinder(environment variables):
environmental key $\leftarrow$ Concatenate values of environment variables
**return** $H$(environmental key)
```

Note that the above technique can be used both by a malware (to obfuscate its malicious payload) or a benign program, e.g., to check if only the rightful user is able to run the protected part of the program [12]. Two main differences between this hash-then-decrypt construct and the one discussed in Section 4 are that (a) the key is obtained as the hash of the constructed environmental variable string – this is obviously done to increase entropy of the key, and (b) the hash of the block B, i.e., h_B, is also provided in the program P'. This second difference, however, means that in our model the obfuscator does not evade detection. The proof is simple: any non-trivial detector D on the unobfuscated set of programs S can keep hashes of the benign and malicious blocks. Thus, the detector retains its advantage (type-I and type-II errors) over the obfuscated variants of the two types of programs. In practice, what this means is that any detector that keeps signatures of past programs will be able to detect this obfuscation. To avoid this, the authors suggest that not all of the block be used for hashing (by using an offset, e.g., by discarding the last few bytes before hashing). This can then evade pre-computed signatures. However, the offset is part of the decryption routine, and hence can be used by a detector to recompute hashes for malware that have already been detected. This "vulnerability" can be removed if instead we use the following routine based on the hash-then-decrypt construction of Section 4. Namely, the obfuscator first constructs:

$$k \leftarrow H(\text{environmental key}), [\![B]\!] \leftarrow \text{Enc}_k(B),\ h_k \leftarrow H(k),$$

and then replaces the block B with:

```
$x \leftarrow$ KeyFinder(environment variables)
**if** $H(x) = h_k$ **then**
    $B' \leftarrow \text{Dec}_x([\![B]\!])$
    $B'$
    end if
```

Notice the double application of the hash function, so that $h_k = H(k) = H(H(\text{environmental key}))$. The program does not store k. Only h_k and the encrypted block $[\![B]\!]$ are hardcoded into the program.

Remark. The construction from Ebowla shown above is a simplification of the one shown in [23], as we ignore compression and encoding of the encrypted block, which is likely there to increase portability of the payload, rather than for security reasons. One important aspect however is how the environment key is derived and how it is reconstructed.

5.2. Finding the Environmental Key – Target vs Non-Target Machines

The main idea behind environmental keying is that finding the key should be easy on the target machines, but hard if the detector does not know the target machines. There is an implicit assumption that the environmental variables used are sufficiently unique (high entropy) among different machines. This problem makes more sense if the attacker's target is a much smaller set of machines than the total pool of machines. Without knowing the identity of the target machines (and hence the state of their environment variables), a detector's only choice may be to run a brute-force search on the environment variables used.

Translated to our formal model, the detector D, given the obfuscation P' of a program using the environmental-key based hash-then-decrypt obfuscation technique of Section 5.1, would like to flag it as a malware or benign program. The detector has no advantage in classifying it as one or the other without finding the environmental key. To model environmental variables and their values taken up by machines, we consider a universal set $\mathcal{C}$ of all environmental values taken up by any machine. The set is bestowed with a probability distribution, modeling the probability that a given environmental variable can take on a specific value. We assume this distribution to be public, and hence also known to the detector. At each time step, we assume that the detector can sample one profile at a time and hence potentially run the given obfuscated program on this environmental profile.[5] Let $\mathcal{T}$ be the target set of profiles from $\mathcal{C}$. That is, those profiles, on which the environmental key matches the hash, and therefore decryption is successful. Let $p_{\mathcal{T}}$ denote the probability of sampling such profiles. We assume that these profiles are hardwired into $\mathcal{O}$ (via the hash digest), and hence the same target profiles work for all programs obfuscated by $\mathcal{O}$.

We are interested in finding:

$$\beta_D^{\mathcal{O}} = \Pr[D(M') = 0 \mid M' \leftarrow \mathcal{O}(M), M \in \mathsf{Malware}]$$

and,

$$\alpha_D^{\mathcal{O}} = \Pr[D(P') = 1 \mid P' \leftarrow \mathcal{O}(P), P \in \mathsf{Benign}],$$

5. This can of course be generalized to polynomially many profiles.

where we assume that D is given access to $\mathcal{C}$. Furthermore by assumption, D is non-trivial, i.e., $\alpha_D + \beta_D < 1$. The observation is that if $p_{\mathcal{T}}$ is a non-negligible function in the size of the profiles, then the detector will find the key in polynomially many samples.[6] In this case, both error rates are $\beta_D^{\mathcal{O}} = \beta_D$ and $\alpha_D^{\mathcal{O}} = \alpha_D$, and hence the obfuscator does not evade detection. On the other, hand, if $p_{\mathcal{T}}$ is a negligible function in the size of the profiles, then the detector cannot find the key in polynomially many samples. The detector then outputs 1 with some fixed probability p, else it outputs 0. This implies that $\alpha_D^{\mathcal{O}} + \beta_D^{\mathcal{O}} = (1-p)+p = 1$, regardless of p. Combining with $\alpha_D + \beta_D < 1$, we see that if $\alpha_D^{\mathcal{O}} \leq \alpha_D$, then necessarily $\beta_D^{\mathcal{O}} > \beta_D$, and hence $\mathcal{O}$ evades detection against D.

Thus, from the malware author's point of view, the goal is to ensure that $p_{\mathcal{T}}$ is negligible. One way to achieve this is to ensure that the set of profiles $\mathcal{C}$ has high entropy. That is, there is a large number of possible values that can be taken up by the accessed environmental variables across all machines.

On the Key Finder Routine. The Key Finder routine takes as input a set of environment variables and "extracts" a string which is the purported key to be used in the hash-then-decrypt construct. The use of the word extract is intentional, as this may not be a simple concatenation of the environment variables. For instance, in the Ebowla framework, the Key Finder routine starts by loading the current value in the `PATH` variable, and then traverses the file system from this value (outputting the current value at each point). Note that we are assuming that the key finding routine is itself not obfuscated. We will return to the case when this routine may be obfuscated as well in Section 7. For now, we assume it to be in the clear. This means that the malware obfuscator cannot hide the environmental variable(s) used to extract the key, as well as the entire routine.

Environment Variables and Entropy. We can define environment variables as showing the current state of the system. They can be categorized into two main types:

1) *Time invariant:* These remain static over time. Examples include: `USER`, `HOME`, `PATH`, and IP address.
2) *Time variant:* These change over time. A simple example is current system time.

For entropy, an important consideration is how many computers take on a particular value of an environment variable. Consider a universe of n computers. Let A be an environment variable, and let $N(A = a)$ denote the number of computers having the state $A = a$ (having their environmental variable A set at a). Denote by $P_A(a) = N(A = a)/n$ as the empirical probability of having the environment variable A set to a. Then, we can see that if A is the system time, and a is the current time, then $P_A(a) = 1$. On the other hand if A is the username (the variable `USER`) then $P_A(a)$

6. We use the usual definition of a negligible function, i.e., one that grows slower than the reciprocal of any polynomial.

is close to 0. A key difficulty is to have estimates of these empirical probabilities in the real-world, as there is a lack of datasets due to obvious privacy concerns. However, we can guess the entropy through what is known about similar variables via other means. For instance:

- *User:* In many organizations, computer login names are a combination of the first and the last name to ensure uniqueness across the organisation. In the US alone, there were 6,299,033 unique surnames according to the 2010 US Census [24]. A different study curated a list of 4,250 unique first names [25]. The combination of these two alone gives a total space of $\approx 2^{35}$. This is arguably a crude lower bound as it is only confined to the US. Furthermore, computer names are more involved, as they may include abbreviations and/or additional characters. However, if the malware is targeting a group of users with similar user names, then the search space reduces drastically. As we have mentioned earilier, the malware will not be able to hide the fact that it is extracting a certain subset of user names.
- *IP Addresses:* There are a total of 2^{32} IP addresses under IPv4. The use of IPv6 will further increase this space. Malware authors might be interested in a certain IP prefix, in which case the space is reduced.
- *System Time:* Time-based activation may start based on the granularity of a second, a minute, an hour, a day, a week, or a year. This gives a granularity of about 2^{25} a year. Once again, broadening the range of time when the malware is triggered reduces the space.

Thus, a combination of all three increases the search space for the detector. But this increase in entropy comes at a tradeoff: the higher the entropy, the more targeted the attack, and the less likely it will be detected by the detector. On the other hand, the lower the entropy, the wider the net cast by the malware, but the easier it is for the detector to identify it as malware.

Time on Non-Target Machines. Given that the Key Finder routine is in the clear, is it possible to make the routine take longer on non-target machines without making it take longer on the target machine regardless of the number of targets? For instance, in the Ebowla framework described above, the code traverses the file system to extract strings, as well as tries all combinations of the different environment variables. Obviously, this is done on both the target and other machines, and hence the excess time also penalizes the target machine. The answer to this question is mixed. Consider the following construct:

1) Denote by a the string obtained by concatenating the values of one or more environmental variables in the target computer (denoted by A).
2) Let m be a large positive integer. Pick a *small* number i from $\{1, 2, \ldots, m\}$.
3) The environmental key is $H(a||i)$.
4) The Key Finder routine is then: concatenate the values of the environment variables A, and concatenate it with $j = 1, 2, \ldots, m$ until a hash match is found.

The idea here is that on any machine that does not exhibit a as the concatenated value of the corresponding environment variable, the Key Finder routine will likely run through all values until m, by which time the detector will discard the machine as not being the target. If m is large enough, this can be substantial time. On the other hand, i can be chosen to be small so that Key Finder returns the key after only going till i on the target machine. A drawback of this construct is that the detector might only run the Key Finder for a feasible number of steps before discarding the current machine as not being the target machine (as the intention of the malware writer would not be to run indefinitely on the target machine). Even if we make m exponentially large, if the number of possible environmental values are small (low entropy, or in other words, the size of profiles $\mathcal{C}$ is polynomially bounded), then the detector can run the Key Finder routine in parallel for several profiles at a time. In our Turing machine language, this means using dovetailing [26] to output the key in polynomial time.

Another possibility is to accept a certain trade-off. Finding the key on the target machine may take time t, but this will also increase the work of the detector by at least t-folds. In the Ebowla framework, this is done by making the Key Finder go through combinations of environmental variables, or traversing through the file system without specifying depth. However, a simpler construct based on hash puzzles (from cryptocurrencies) may do the same trick [27]. In this case, we can build the puzzle so that it takes "on-average" a certain amount of time before the key is found on the target machine. On the non-target machine, this will take roughly the same amount of time per computer profile by the detector before it can decide the search to be futile, as the detector knows the possible number of combinations. A benefit of this approach is that the program author does need more environmental variables to increase search space of the detector.

This argument also shows that it is not possible to devise a Key Finder that will make the detector take exponential time while running for polynomial time on the target machine. This follows from the dovetailing trick [26]. As long as the detector has the target profile in its polynomially sized set of profiles, the detector will eventually find the key by running the same routine. Thus, the only possibility is to ensure that the number of profiles is of exponential size. This can be achieved only via increasing the entropy of the environmental variables used.

6. Deniable Encryption

Environmental keying does raise suspicion in the sense that a detector knows the presence of encrypted code, even if the encrypted payload may not be decrypted by the detector.[7] But what if there are multiple possible decryptions of the same encrypted payload, and only one of them malicious? If the malware detector is only able to decrypt the benign versions then it is likely to flag the resulting program as benign. On the other hand, on the target computer, if it is more likely that the malicious version will be decrypted, then we have a malware that potentially avoids detection, albeit in a more steganographic sense. An encryption system that has such a property is called *deniable encryption* [29].

The possible use of deniable encryption in malware has been discussed in [23], [30]. A simple XOR-based scheme illustrates the concept. Suppose we have a benign program P and a malware M. Abusing notation, we use P and M to denote the corresponding strings. We also assume that $|P| = |M|$; otherwise, one can use padding to make the two equal. Then, one constructs a key K_1 of length equal to $|P| = |M|$ and then computes:

$$C = M \oplus K_1,$$

and further computes:

$$K_2 = C \oplus P.$$

By the property of XOR, we have

$$C = M \oplus K_1 = P \oplus K_2.$$

Thus, C could either be the encryption of M under the key K_1 or the encryption of P under the key K_2. Note that, since K_1 is random, so too is K_2 (without conditioning on the other). The idea is to deny C being an encryption of a malware, and presenting the key K_2 if need be to show that the underlying payload was P [30].

There are some issues with this approach:

- The key length needs to be the same as the malware program, which can be quite big. Alternative deniable encryption techniques are presented in [29], based on both public-key and shared-key cryptography. However, they are rather inefficient in terms of length of ciphertext per plaintext. One such example is given below.
- How does the malware author guarantee that the correct key is used in the target computer? One way is to use an environmental key. But since the key K_2 is dependent on the first key, this would make finding the second key hard on non-target computers as well, which defeats the purpose of deniable encryption.
- Another way of providing the two keys is to host them on a server, which is then retrieved by the program in runtime (e.g., HTTP keying). The malware author initially only uploads the key K_2, and later at a specific time it posts K_1. Thus, the detector will not be able to decrypt the malware block until the relevant key (K_1) is uploaded.
- There is dependency between the two keys. Once the key K_2 is obtained, K_1 is not random. An ideal technique (from the malware author's point of view) should be where the two keys are independent.
- The scenario of deniable encryption is not entirely suited to the obfuscation. Deniable encryption assumes a sender and a receiver. The adversary can coerce either one of them to reveal the key. There are also sender-and-receiver deniability schemes, but they assume some intermediaries through which the encrypted communication is transmitted. In the obfuscation scenario, the sender and receiver are both the obfuscated program. Therefore, some public-key encryption schemes described in [29] do not work in this scenario as the trapdoor information has to be in the code.

7. Indeed, presence of high entropy code can be flagged as indication of malicious intent [28].

Another symmetric key encryption scheme from [29] can be summarised as follows: the obfuscator encrypts a benign program P and malware M separately with two encryption keys. The obfuscated program contains a routine to search for keys (say they are environmental keys). The hash-then-decrypt construct matches both hashes, and if any of them match, it decrypts the program matching the hash. Given that in most computers, the benign routine is always decrypted, the malware writer may use plausible deniability. However, the unencrypted code will always raise suspicion. And the detector would not hastily label this as benign.

7. Cryptographic Notions of Obfuscation

There are some key differences between the goal of cryptographic obfuscation and real-world use of cryptographic tools in malware obfuscation. The former considers a program P, and requires its obfuscated version, $\hat{P}$, to be functionally equivalent, while at the same time (informally) nothing should be learned about P from the description of $\hat{P}$. However, this does not directly translate to the goal of malware obfuscation (as defined in this paper), which aims to evade detection. In particular, since $\hat{P}$ preserves functionality, the detector D can still label it as malware based on its input-output behaviour, as we shall see shortly. The above informal description of cryptographic obfuscation captures the concept of virtual black box (VBB) obfuscation, defined as follows.

Definition 3 (VBB Obfuscation [5])**.** *A program obfuscator $\mathcal{O}$ is a virtual black box (VBB) obfuscator, if it preserves functionality and for all PPT algorithms D, there exists a PPT simulator R such that for all programs P*

$$\left|\Pr\left[D(P', 1^\lambda) = 1\right] - \Pr\left[R^P(|P|, 1^\lambda) = 1\right]\right| < \textit{negl}(\lambda) \quad (2)$$

where $P' \leftarrow \mathcal{O}(P, 1^\lambda)$.

In the above, λ is the security parameter, and we require any algorithm taking it as input to run in polynomial time in its size as well. By now it is well-known that VBB obfuscation is impossible in general [5]. As a result, alternative "weaker" notions have been proposed of which the notion of indistinguishability obfuscation (iO) shows the most promise.

Definition 4 (iO [5]). *A program obfuscator $\mathcal{O}$ is an indistinguishability obfuscator if it preserves functionality, and for all functionally equivalent programs P_1 and P_2 of equal size*

$$\left|\Pr\left[D(P_1', 1^\lambda) = 1\right] - \Pr\left[D(P_2', 1^\lambda) = 1\right]\right| < negl(\lambda) \tag{3}$$

where $P_i' \leftarrow \mathcal{O}(P_i, 1^\lambda)$, for $i = 1, 2$.

Note that functionally equivalent programs means that $P(x) = P'(x)$ for all inputs in the domain of P (and P'). Our informal statement above can now be stated formally.

Proposition 5. *Let $\mathcal{O}$ be a VBB or iO obfuscator. Then there exists a detector D such that $\mathcal{O}$ does not evade detection from D.*

Proof. Let D be the detector of Proposition 2. Then, since $\mathcal{O}$ preserves functionality, the result follows from Proposition 2. □

We can get around this limitation if we let go of the "functionality preserving" requirement of the obfuscated program. Namely, we only want the program to act as a malware on a limited (target) set of computers (see Definition 2). In this case, one may devise a malware that runs as a malware only if the current computer's name equals a predefined list of names (a use of environmental keying). However, there does not seem to be much benefit of obfuscation here over the hash-then-decrypt construct. If the program reads the environmental variables, it is necessary interacting with its environment, and hence we can learn the variables accessed by it through dynamic analysis. This brings us back to the same issue with environmental keying and the hash-then-decrypt construct.

Obfuscating the Key Finder Routine. But the advantage here is that the Key Finder routine itself can be obfuscated. For instance, the Key Finder routine could read a bunch of environment variables on the machine, and only use a small subset of them to decide whether to trigger the malware behavior or not. The remaining variables can be discarded. In this case, since the Key Finder routine is obfuscated, the detector will not know which environmental variable is needed, and hence load a larger environmental profile than is actually being used by the program.

On VBB and iO. Cryptographic obfuscation, both VBB and iO, is a widely studied problem. There are works that show that VBB obfuscation is still possible under *idealized models*, e.g., the generic graded encoding model [31]. On the iO end, various constructions have beep proposed via multilinear maps (or graded encoding schemes) including [32], [33], [34]. Unfortunately, many of these schemes have been broken, e.g., [34]. An account appears in [35]. There are also constructions of iO schemes not based on multilinear maps or graded encoding schemes, e.g., [36]. Finally, there are also implementations of iO schemes [37] as reported in [38], and [39]. Thus, while this is still a field with ongoing research, a successful and practical scheme may have an impact in terms of malware obfuscation, e.g., by obfuscating the Key Finder routine in the hash-then-decrypt construct.

8. Obfuscation using Cryptographic Tools in the Wild

Datasets showing the prevalence of cryptographic obfuscation in malware are few and far between, partly because it is difficult to programmatically infer if a cryptographic library is being used for obfuscation or for other tasks such as encrypting communication, or encrypting local files (in the case of ransomware). In Section 9, we discuss some works in literature that have surveyed obfuscation techniques in the wild. However, since their coverage is not limited to obfuscation via cryptographic techniques (evasive or otherwise), we, therefore, run our own analysis to get an idea of cryptographic malware obfuscation. As a first step, we need to have a ground-truth dataset to identify malware that use cryptographic obfuscation. For this, we leverage labelled samples (binaries) from the Dike dataset.[8] The labels for this dataset are obtained by parsing the categories returned by antivirus vendors aggregated by VirusTotal. Among the categories is that of an "encryptor," which corresponds to programs that use obfuscation. Taking this smaller dataset as ground truth, we extract EMBER features [40] to train a machine learning model to predict cryptographic obfuscation over the much larger SOREL dataset [41].

The specific label from the Dike dataset we use to characterize programs using cryptographic obfuscation are 6,197 samples flagged as "encryptors" and not "ransomware." The remaining 3,735 samples are considered as not using cryptographic obfuscation. Our decision to exclude ransomware from encryptors is due to the high likelihood that ransomware will use encryption libraries for tasks other than obfuscation. With a training/testing split of 80%/20%, a random forest classifier with 100 trees is trained with the EMBER features as inputs and the labels defined above. The resulting classifier yields an AUC of 0.966 in distinguishing between the two defined classes: encryptors and non-encryptors. The confusion matrix with the breakdown of original classes and errors is provided in Figure 1. With these results we are satisfied in the ability of the classifier to infer the encryptor label on the unlabelled EMBER features of the SOREL dataset. We sample 200,000 Malware labelled samples, and 200,000 Benign labelled samples from the SOREL dataset. The classifier identified 10.9% (21,834) of the Malware samples as using encryption obfuscation. On the other hand, 6.5% (13,087) of Benign samples were also flagged as using encryption obfuscation.

To validate these predictions on the SOREL dataset, we analyze a smaller sample of 1,000 samples each with the highest and lowest confidence values labelled by the classifier as using cryptographic obfuscation, from both Malware and Benign samples as labelled in the SOREL dataset. With these 4,000 samples, we obtain a VirusTotal (VT) report detailing the sample's behaviour and consider

8. Dike Dataset: https://github.com/iosifache/DikeDataset

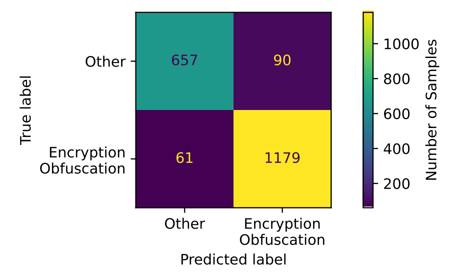

Figure 1. Confusion matrix for cryptographic obfuscation detection on the test set from the Dike Dataset; AUC was 0.966, Accuracy was 92.4%.

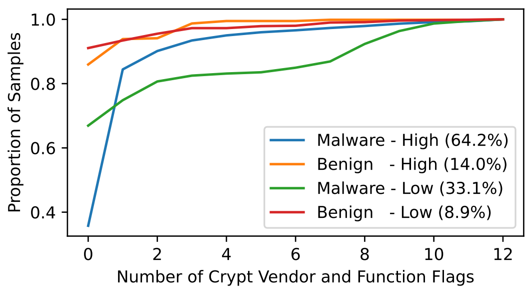

Figure 2. Cumulative distribution of SOREL samples with VT reports demonstrating encryption keywords found in imported function names and AV vendor reports. The four groups consist of 1000 samples of the Highest and Lowest probability of using cryptographic obfuscation. The percentage shown is proportion of samples containing a search term.

heuristics of their behaviour that may have warranted the classification. VT is an anti-virus aggregator; when a user submits a binary for analysis, VT scans the binary with a selection of anti-virus vendors. If more than a threshold number of these vendors flag the binary as malicious, VT will report the sample as malicious. Within each sample's VT report, each vendor also reports a short indicator of what sort of maliciousness exists. For example, Kaspersky may report `Packed.Win32.Krap.iu`, which indicates that the sample belongs to the family of malware which protects against reverse engineering, i.e., obfuscation. Additionally, the VT analysis extracts any imported functions from the binary. From both these sources, we use a list of search terms that may indicate obfuscation using encryption. This is the same list used in Dike Dataset and includes: "crypt", "cryp", "coder", "pack", and "krypt".

The cumulative distribution of samples which contained VT reports are depicted in Figure 2. More than 64% of the Malware samples labelled as encryptors with high confidence contain at least one vendor labelling it as using cryptographic obfuscation, compared to 33% among the malware samples labelled as such with low confidence. This sits in contrast with benign samples; only 8.9 and 14% were flagged as using cryptographic obfuscation by any vendor. This discrepancy is likely due to the "safe" nature of benign samples: AV vendors not detecting any concerning behaviour, will not label them as belonging to any malware family; hence no "crypt" search term. In contrast, our classifier operating on the EMBER features may have captured encryption behaviour otherwise not flagged by the AV vendors. Nevertheless we are confident that the classifier correctly detects encryptors in malware, indicating that 10% of them as using some form of cryptographic obfuscation.

9. Related Work

Enviromental Keying. The concept of environmental keying was first proposed by Riordan and Schneier in [8]. Further investigation of the topic in the academic circle appears in [19] and the work on secure triggers [42]. A related concept is used in the mesh design, hash-then-decrypt method proposed by Nate Lawson [12]. This uses the same concept of secure triggers albeit for different applications including gaming (unlocking higher levels, if the user has collected certain items), as well as protecting software. Glynos looks into different ways in which environmental keys can be derived [43]. Offensive security practitioners have also looked into environmental keying [23], [44]. Most importantly, the technique has been used in real-world malware to encrypt part or whole of the payload.[9] The most notable example being the Gauss malware [45]. To the best of our knowledge, the encrypted payload of this malware, detected in 2011, has still not been decrypted [46]. The malware consisted of a reconaissance component which would collect system information. The system information was then used by attackers to derive environmental keys [46]. As noted in [46], the payload could only be decrypted by the unique target in an air-gapped network, and therefore, researchers have failed to decrypt it even after trying millions of combinations and publicly releasing technical details. Such sophisticated use of malware obfuscation can most likely be attributed to state-based actors. In general, encryption may also be used for other purposes, such as concealing the communication with a remote server [22]. A report by CISCO states that an analysis of 400,000 malware binaries revealed about 70% of them as using some encryption in 2017 [20].

Theoretical Modeling. Some researchers have also looked at formalising the concept of programs using environmental keying and the properties they must satisfy. Futoransky et al [42] proposed the notion of secure triggers, which is similar to the environmental keying based hash-then-decrypt construct first proposed in [8]. The difference being that instead of using the hash function to check if the key is valid, they use the encryption of the all 0 string as a predicate. They show that the construct is secure under the universal composability framework [47], in the sense that the adversary (detector in our case) does not learn the contents of the encrypted block if the predicate is from a large enough space and under the semantic security of the encryption scheme. Blackthorne et al. [4] define security of environmental keying under an analyst (detector) with different capabilities, such as not knowing the target versus after the malware has infected its target. They also look at malware that could change its behaviour after sensing

9. See for instance: https://attack.mitre.org/techniques/T1480/001/

if its environment has been altered by the analyst (e.g., sandboxed environments) [48]. The authors of [49] look at malware that derives its environment keys not just from the target but also from the nodes in the network in its path. Decryption is only successful if the correct path order is followed. In this paper, we have proposed a definition independent of the cryptographic technique being used for obfuscation. Our model defines the detector (variously, the analyst or the adversary) in terms of its ability to distinguish between malware and benign programs, instead of pinning it to environmental keying.

Surveys on Malware Obfuscation. Several works have studied the prevalence of obfuscation techniques used by malware in the wild. You and Yim [50] give a brief taxonomy of malware obfuscation techniques, amongst which they mention encrypted malware. Such malware consists of an encrypted part and a decryptor. The drawback of such malware from the authors' point of view is that antivirus scanners may detect such malware through the commonly used decrypting routine. As mentioned in our paper, we also consider this technique as non-evasive, but mainly due to the fact that malware can be detected on run-time. Aligot [1] is a tool that identifies malware samples that obfuscate the use of block ciphers, hash functions and public key encryption. The authors test their tool on a couple of known families of malware that use cryptographic tools. However, their main experiments are done on synthetically generated samples. The tool may not be able to differentiate the use of cryptographic functions for obfuscation versus for other purposes. The authors in [2] propose the StringHound tool to detect string obfuscation. They enlist some encryption algorithms used for string encryption, e.g., XOR encoding and AES encryption. As the name suggests, their survey is limited to string obfuscation. Several obfuscation techniques for Android malware are discussed in [3], of which string and class encryption are related to the topic of this SoK. The goal of their survey however is to show the affect of these obfuscation techniques on the unobfuscated programs and whether or not anti-malware engines are able to detect them. Wermke et al look at different obfuscation techniques employed on Android apps via popular obfuscation software [7]. Related to cryptography are string and class obfuscation, which we categorise as non-evasive techniques. A layered taxonomy of obfuscation techniques is presented by Xu et al [38] to help developers protect their software from intellectual theft in a more systematic way. Among the techniques they mention the hash-then-decrypt construct from [19] as well as cryptographic obfuscation (indistinguishability obfuscation). However, they do not view the techniques in light of their efficacy in evading detection against a model such as the one proposed in our paper.

Other Work Related. There are some other works worth mentioning in the use of cryptographic tools in malware. Some of these works are modifications of environmental keying, and are therefore left out as they retain the basics of that construct. For instance, the Bradley virus [30] uses nested encryption, where the key for the first encryption is derived from the environment variable. If the hash of this derived key matches the hardcoded hash, then the program continues to decrypt the next segment of the program. Otherwise it deletes the whole malware. Perhaps, the most notable area of work marrying cryptography and malware is cryptovirology [51], which deals with the study of malware that use cryptography to encrypt operations on the victim's computer. Ransomware are the most notable example of such malware. However, note that the use of cryptography in such programs is not meant for obfuscation, as encryption is used mainly for denying access to files with only the malware author having the means to decrypt them.

10. Limitations and Open Problems

- Our definition of malware detection does not take distributional aspects into account. For instance, cryptographic libraries may be used more by malware than benign programs. Thus, the definition does not rule out detectors who base their decisions on such distributions. Likewise, mere presence of any obfuscation may indicate that the program is more likely to be malware rather than benign. However, our definition challenges the detector in being able to differentiate the same obfuscation technique being applied on malware and benign programs, which is arguably more principled.
- The only obfuscation techniques using cryptographic tools that are shown to be hard in this paper are hash-then-decrypt construct and deniable encryption using environmental keys. Both use environmental keys and therefore can only evade detection if targeting only a small subset of machines. This appears to be a fundamental limitation of malware obfuscation. It appears that any technique that runs the malicious payload on a large number of machines after a small amount of time can in principle be detected; the detector only needs to wait for sufficiently long before it is able to detect malware.
- On the experimental side, while we have shown some results on the real-world use of cryptographic obfuscation by malware programs, this needs to be further validated and refined into different classes of techniques for obfuscation. In particular, we do not know how prevalent is the use of environmental keying in malware obfuscation.
- Many real-world malware obfuscation techniques are only used to avoid specific types of malware analysis, e.g., use of packing to avoid static analysis, or assessing the run-time environment to detect whether a program is being dynamically analysed in a virtual environment. In our paper, we have not considered these specifics as these techniques do not provide provable guarantees of evading detection.

Ethical Considerations

All instances of malware obfuscation detailed in this paper are in the public domain. As such, we have not introduced any malware obfuscation technique that may jeopardize the security of systems. The aim of this paper is to simply categorise them from the point of view of difficulty of detection.

Acknowledgments

This work was partially supported by the Australian Defence Science and Technology (DST) Group under the Next Generation Technology Fund (NGTF) scheme.

References

[1] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: Cryptographic function identification in obfuscated binary programs," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 169–182.

[2] L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonysamy, and M. Mezini, "Hidden in plain sight: Obfuscated strings threatening your privacy," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 694–707.

[3] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.

[4] J. Blackthorne, B. Kaiser, B. Fuller, and B. Yener, "Environmental authentication in malware," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2017, pp. 381–400.

[5] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Annual international cryptology conference*. Springer, 2001, pp. 1–18.

[6] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[7] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in google play," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 222–235.

[8] J. Riordan and B. Schneier, "Environmental key generation towards clueless agents," in *Mobile agents and security*. Springer, 1998, pp. 15–24.

[9] T. Baigneres, P. Junod, and S. Vaudenay, "How far can we go beyond linear cryptanalysis?" in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2004, pp. 432–450.

[10] F. Cohen, "Computer viruses," Ph.D. dissertation, University of Southern California Doctoral dissertation, 1986.

[11] D. Evans, "On the impossibility of virus detection," *Retrieved from*, 2017.

[12] N. Lawson, "Mesh design pattern: Hash-and-decrypt," root labs rdist, https://rdist.root.org/2007/04/09/mesh-design-pattern-hash-and-decrypt/, 2007.

[13] T. M. Cover, *Elements of information theory*. John Wiley & Sons, 1999.

[14] S. Arora and B. Barak, *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[15] A. Apvrille, "Cryptography for mobile malware obfuscation," in *RSA Conference Europe*, 2011.

[16] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[17] J. Calvet, C. R. Davis, and P.-M. Bureau, "Malware authors don't learn, and that's good!" in *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2009, pp. 88–97.

[18] 0xPat, "Malware development part 5 – tips and tricks," https://0xpat.github.io/Malware_development_part_5/, 2020.

[19] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation." in *NDSS*. Citeseer, 2008.

[20] C. Cisco, "Annual cybersecurity report, 2018," https://www.cisco.com/c/dam/m/hu_hu/campaigns/security-hub/pdf/acr-2018.pdf, 2018.

[21] S. Miller and P. Smith, "Rise of legitimate services for backdoor command and control," https://www.anomali.com/resources/anomali-labs-report/rise-of-legitimate-services-for-backdoor-command-and-control, Anomali, Tech. Rep., 2017, Tech. Rep., 2017.

[22] S. Gallagher, "Nearly half of malware now use tls to conceal communications," https://news.sophos.com/en-us/2021/04/21/nearly-half-of-malware-now-use-tls-to-conceal-communications/, 2021.

[23] T. Morrow and J. Pitts, "Genetic malware: Designing payloads for specific targets," *Infiltrate*, 2016.

[24] J. Comenetz, "Frequently occurring surnames in the 2010 census," *United States Census Bureau*, pp. 1–8, 2016.

[25] K. Tzioumis, "Demographic aspects of first names," *Scientific data*, vol. 5, no. 1, pp. 1–9, 2018.

[26] L. Lovász, "Computation complexity," *Lecture Notes. http://ftp. cs. yale. edu/pub/lovasz. pub*, 1994.

[27] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press, 2016.

[28] E. Filiol, "Malicious cryptography techniques for unreversable (malicious or not) binaries," *arXiv preprint arXiv:1009.4000*, 2010.

[29] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky, "Deniable encryption," in *Annual International Cryptology Conference*. Springer, 1997, pp. 90–104.

[30] É. Filiol and F. Raynal, "Malicious cryptography... reloaded," in *CanSecWest Conference*, 2008.

[31] Z. Brakerski and G. N. Rothblum, "Virtual black-box obfuscation for all circuits via generic graded encoding," in *Theory of Cryptography Conference*. Springer, 2014, pp. 1–25.

[32] S. Garg, C. Gentry, and S. Halevi, "Candidate multilinear maps from ideal lattices," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 1–17.

[33] J.-S. Coron, T. Lepoint, and M. Tibouchi, "Practical multilinear maps over the integers," in *Annual Cryptology Conference*. Springer, 2013, pp. 476–493.

[34] C. Gentry, S. Gorbunov, and S. Halevi, "Graph-induced multilinear maps from lattices," in *Theory of Cryptography Conference*. Springer, 2015, pp. 498–527.

[35] M. R. Albrecht, P. Farshim, S. Han, D. Hofheinz, E. Larraia, and K. G. Paterson, "Multilinear maps from obfuscation," *Journal of Cryptology*, vol. 33, no. 3, pp. 1080–1113, 2020.

[36] A. Jain, H. Lin, and A. Sahai, "Indistinguishability obfuscation from well-founded assumptions," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, 2021, pp. 60–73.

[37] D. Apon, Y. Huang, J. Katz, and A. J. Malozemoff, "Implementing cryptographic program obfuscation," *Cryptology ePrint Archive*, 2014.

[38] H. Xu, Y. Zhou, J. Ming, and M. Lyu, "Layered obfuscation: a taxonomy of software obfuscation techniques for layered security," *Cybersecurity*, vol. 3, no. 1, pp. 1–18, 2020.

[39] S. Halevi, T. Halevi, V. Shoup, and N. Stephens-Davidowitz, "Implementing bp-obfuscation using graph-induced encoding," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[40] H. S. Anderson and P. Roth, "Ember: an open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.

[41] R. Harang and E. M. Rudd, "Sorel-20m: A large scale benchmark dataset for malicious pe detection," *arXiv preprint arXiv:2012.07634*, 2020.

[42] A. Futoransky, E. Kargieman, C. Sarraute, and A. Waissbein, "Foundations and applications for secure triggers," *ACM Transactions on Information and System Security (TISSEC)*, vol. 9, no. 1, pp. 94–112, 2006.

[43] D. A. Glynos, "Context-keyed payload encoding: Fighting the next generation of ids," in *Athens IT Security Conference (ATH. C0N)*. Citeseer, 2010.

[44] L. Loobeek, "Protect your payloads: Modern keying techniques," http://www.irongeek.com/i.php?page=videos/derbycon8/track-4-03-protect-your-payloads-modern-keying-techniques-leo-loobeek, 2018.

[45] B. Bencsáth, G. Pék, L. Buttyán, and M. Felegyhazi, "The cousins of stuxnet: Duqu, flame, and gauss," *Future Internet*, vol. 4, no. 4, pp. 971–1003, 2012.

[46] A. Dorais-Joncas and F. Munõz, "Jumping the air gap: 15 years of nation-state effort," 2021.

[47] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[48] J. Blackthorne, B. Kaiser, and B. Yener, "A formal framework for environmentally sensitive malware," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 211–229.

[49] H. Galteland and K. Gjøsteen, "Malware encryption schemes-rerandomizable ciphertexts encrypted using environmental keys." *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 1007, 2017.

[50] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 2010, pp. 297–300.

[51] A. Young and M. Yung, "Cryptovirology: Extortion-based security threats and countermeasures," in *Proceedings 1996 IEEE Symposium on Security and Privacy*. IEEE, 1996, pp. 129–140.