White-Box Security Notions for Symmetric Encryption Schemes[⋆]

Cécile Delerablée[1], Tancrède Lepoint[1,2], Pascal Paillier[1], and Matthieu Rivain[1]

[1] CryptoExperts, 41 boulevard des Capucines, 75002 Paris, France
{firstname.lastname}@cryptoexperts.com

[2] École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France

Abstract. White-box cryptography has attracted a growing interest from researchers in the last decade. Several white-box implementations of standard block-ciphers (DES, AES) have been proposed but they have all been broken. On the other hand, neither evidence of existence nor proofs of impossibility have been provided for this particular setting. This might be in part because it is still quite unclear what white-box cryptography really aims to achieve and which security properties are expected from white-box programs in applications. This paper builds a first step towards a practical answer to this question by translating folklore intuitions behind white-box cryptography into concrete security notions. Specifically, we introduce the notion of white-box compiler that turns a symmetric encryption scheme into randomized white-box programs, and we capture several desired security properties such as one-wayness, incompressibility and traceability for white-box programs. We also give concrete examples of white-box compilers that already achieve some of these notions. Overall, our results open new perspectives on the design of white-box programs that securely implement symmetric encryption.

Keywords: White-Box Cryptography, Security Notions, Attack Models, Security Games, Traitor tracing.

1 Introduction

Traditionally, to prove the security of a cryptosystem, cryptographers consider attack scenarios where an adversary is only given a *black-box* access to the cryptographic system, namely to the inputs and outputs of its underlying algorithms. Security notions are built on the standard paradigm that the algorithms are known and that computing platforms can be trusted to effectively protect the secrecy of the private key.

However attacks on *implementations* of cryptographic primitives have become a major threat due to side-channel information leakage (see for example [17,27]) such as execution time, power consumption or electromagnetic emanations. More generally, the increasing penetration of cryptographic applications onto untrusted platform (the end points being possibly controlled by a malicious party) makes the black-box model too restrictive to guaranty the security of *programs* implementing cryptographic primitives.

White-box cryptography was introduced in 2002 by Chow, Eisen, Johnson and van Oorschot [10,11] as the ultimate, *worst-case* attack model. This model considers an attacker far more powerful than in the classical black-box model (and thus more representative of real-world attackers); namely the attacker is given full knowledge and full control on both the algorithm and its execution environment. However, even such powerful capabilities should

[⋆] A short version of this paper appears in the proceedings of SAC 2013.

not allow her to *e.g.* extract the embedded key[3]. White-box cryptography can hence be seen as a restriction of general obfuscation where the function to protect belongs to some narrower class of cryptographic functions indexed by a secret key. From that angle, the ultimate goal of a white-box implementation is to leak nothing more than what a black-box access to the function would reveal. An implementation achieving this strong property would be as secure as in the black-box model, in particular it would resist *all existing and future* side-channel and fault-based attacks. Although we know that general obfuscation of any function is impossible to achieve [1], there is no known impossibility result for white-box cryptography and positive examples have even been discovered [14,7]. On the other hand, the work of Chow *et al.* gave rise to several proposals for white-box implementations of symmetric ciphers, specifically DES [10,20,32] and AES [11,6,33,18], even though all these proposals have been broken [15,3,12,31,21,23,22,19].

Our belief is that the dearth of promising white-box implementations is also a consequence of the absence of well-understood security goals to achieve. A first step towards a theoretical model was proposed by Saxena, Wyseur and Preneel [28], and subsequently extended by Wyseur in his PhD thesis [30]. These results show how to translate any security notion in the black-box model into a security notion in the white-box model. They introduce the *white-box property* for an obfuscator as the ability to turn a program (modeled as a polynomial Turing machine) which is secure with respect to some black-box notion into a program secure with respect to the corresponding white-box notion. The authors then give an example of obfuscator for a symmetric encryption scheme achieving the white-box equivalent of semantic security. In other words, the symmetric encryption scheme is turned into a secure asymmetric encryption scheme. While these advances describe a generic model to translate a given notion from the black-box to the white-box setting, our aim in this paper is to define explicit security notions that white-box cryptography should realize in practice. As a matter of fact, some of our security notions are not black-box notions that one would wish to preserve in the white-box setting, but arise from new features potentially introduced by the white-box compilation. Note that although we use a different formalism and pursue different goals, our work and those in [28,30] are not in contradiction but rather co-exist in a wider framework.

Our Contributions. We formalize the notion of *white-box compilers* for a symmetric encryption scheme and introduce several security notions for such compilers. As traditionally done in provable security (*e.g.* [2]), we consider separately various adversarial goals (*e.g.* decrypt some ciphertext) and attack models (*e.g.* chosen ciphertext attack), and then obtain distinct security definitions by pairing a particular goal with a particular attack model. We consider four different attack models in the white-box context: the chosen plaintext attack, the chosen ciphertext attack, the recompilation attack and the chosen ciphertext and recompilation attack. We formalize the main security objective of white-box cryptography which is to protect the secret key as a notion of *unbreakability.* We show that additional security notions should be considered in applications and translate folklore intuitions behind white-box

[3] Quoting [10], the "choice of the implementation is the sole remaining line of defense and is precisely what is pursued in white-box cryptography".

cryptography into concrete security notions; namely the *one-wayness*, *incompressibility* and *traceability* of white-box programs. For the first two notions, we show an example of a simple symmetric encryption scheme over an RSA group for which an efficient white-box compiler exists that provably achieves both notions. We finally show that white-box programs are efficiently traceable by simple means assuming that functional perturbations can be hidden in them. Overall, our positive results shed more light on the different aspects of white-box security and provide concrete constructions that achieve them in a provable fashion.

2 Preliminaries

Symmetric Encryption. A symmetric encryption scheme is a tuple $\mathcal{E} = (\mathsf{K}, \mathsf{M}, \mathsf{C}, K, E, D)$ where

- K is the key space,
- M is the plaintext (or message) space,
- C is the ciphertext space,
- K is a probabilistic algorithm that returns a key $k \in \mathsf{K} = \mathsf{range}\,(K())$,
- E is a deterministic encryption function mapping elements of $\mathsf{K} \times \mathsf{M}$ to elements of C,
- D is a deterministic decryption function mapping elements of $\mathsf{K} \times \mathsf{C}$ to elements of M.

We require that for any $k \in \mathsf{K}$ and any $m \in \mathsf{M}$, $D(k, E(k, m)) = m$. Most typically, $\mathcal{E}$ refers to a block-cipher in which case all sets are made of binary strings of determined length and $\mathsf{C} = \mathsf{M}$.

Programs. A program is a word in the language-theoretic sense and is interpreted in the explicit context of a programming model and an execution model, the details of which we want to keep as abstracted away as possible. Programs differ from remote oracles in the sense that their code can be executed locally, read, copied and modified at will. Successive executions are inherently stateless and all the “system calls” that a program makes to external resources such as a random source or a system clock can be captured and responded arbitrarily. Execution can be interrupted at any moment and all the internal variables identified by the program's instructions can be read and modified arbitrarily by the party that executes the program.

For some function f mapping some set A to some set B, we denote by $\mathsf{prog}\,(f)$ the set of all programs implementing f. A program $P \in \mathsf{prog}\,(f)$ is said to be fully functional with respect to f when for any $a \in \mathsf{A}$, $P(a)$ returns $f(a)$ with probability 1. P is said to be δ-functional (with respect to f) when P is at distance at most $\delta \in [0, 1]$ from f, *i.e.*

$$\varDelta(P, f) \overset{\text{def}}{=} \Pr[a \overset{\$}{\leftarrow} \mathsf{A}\,;\ b \leftarrow P(a) : b \neq f(a)] \leqslant \delta\ .$$

The set of δ-functional programs implementing f is noted δ-$\mathsf{prog}\,(f)$. Obviously 0-$\mathsf{prog}\,(f) = \mathsf{prog}\,(f)$.

Random Experiments. A random experiment is an interactive protocol played by a group of probabilistic algorithms interacting together. Random experiments are also referred to as (probabilistic) games and are expressed as just a list of actions involving the players. We denote by

$$\Pr\left[\mathsf{action}_1\,;\ \mathsf{action}_2\,;\ \ldots\,;\ \mathsf{action}_n : \mathsf{event}\right]$$

the probability that event occurs after executing $\mathsf{action}_1, \ldots, \mathsf{action}_n$ in sequential order, the probability being taken over the probability spaces of all the random variables involved in these actions. One often refers to those as the random coins of the game $(\mathsf{action}_1, \ldots, \mathsf{action}_n)$.

We denote by $a \overset{\$}{\leftarrow} S$ the action of picking a independently and uniformly at random from some set S, and by $a \leftarrow \mathcal{R}(\cdots)$ the action of running algorithm $\mathcal{R}$ on some inputs and naming a the value returned by $\mathcal{R}$.

Other Notations. If A is some set, $|\mathsf{A}|$ denotes its cardinality. If $\mathbb{A}$ is some generator *i.e.* a random source with some prescribed output range A, $H(\mathbb{A})$ denotes the output entropy of $\mathbb{A}$ as a source. Abusing notations, we may also denote it by $H(a)$ for $a \leftarrow \mathbb{A}(\cdots)$. Finally, when we write $\mathcal{O}(\cdot) = \epsilon$, we mean that $\mathcal{O}$ is the oracle which, on any input, returns the empty string ϵ.

3 White-Box Compilers

In this paper, we consider that a *white-box implementation* of the scheme $\mathcal{E}$ is a program produced by a publicly known compiling function $\mathbf{C}_{\mathcal{E}}$ which takes as arguments a key $k \in \mathsf{K}$ and possibly a diversifying nonce $r \in \mathsf{R}$ drawn from some randomness space R. We will denote the compiled program by $[E_k^r]$ (or $[E_k]$ when the random nonce r is implicit or does not exist), namely $[E_k^r] = \mathbf{C}_{\mathcal{E}}(k, r)$.

A compiler $\mathbf{C}_{\mathcal{E}}$ for $\mathcal{E}$ is *sound* when for any $(k, r) \in \mathsf{K} \times \mathsf{R}$, $[E_k^r]$ exactly implements the function $E(k, \cdot)$ (*i.e.* it is fully functional). Therefore $[E_k^r]$ accepts as input any $m \in \mathsf{M}$ and always returns the correct encryption $c = E(k, m)$. At this stage, we only care about sound compilers.

Remark 1. In the above definition, we consider white-box compilers for the encryption function. However, since we focus on deterministic encryption – $E(k, \cdot)$ and $D(k, \cdot)$ being inverse of one another, we can swap roles without loss of generality and get compilers for the decryption procedure. We will precisely do this in Section 7.

Note again that $[E_k]$ differs in nature from $E(k, \cdot)$. $E(k, \cdot)$ is a mapping from M to C, whereas $[E_k]$ is a word in some programming language (the details of which we want to keep away from) and has to fulfill some semantic consistency rules. Viewed as a binary string, it has a certain bitsize $\mathsf{size}\left([E_k]\right) \in \mathbb{N}$. Even though $E(k, \cdot)$ is deterministic, nothing forbids $[E_k]$ to collect extra bits from a random tape and behave probabilistically. For an input $m \in \mathsf{M}$ and random tape $\rho \in \{0, 1\}^*$, $[E_k](m, \rho)$ takes a certain time $\mathsf{time}\left([E_k](m, \rho)\right) \in \mathbb{N}$ to complete execution.

3.1 Attack Models

The first step in specifying new security notions for white-box cryptography is to classify the threats. This section introduces four distinct attack models for an adversary $\mathcal{A}$ in the white-box model: the *chosen plaintext attack* (CPA), the *chosen ciphertext attack* (CCA), the *recompilation attack* (RCA) and the *chosen ciphertext and recompilation attack* (CCA+RCA). In all of these, we assume that the compiler $\mathbf{C}_{\mathcal{E}}$ is public, *i.e.* at any point in time, the adversary $\mathcal{A}$ can select any key $k \in \mathsf{K}$ and nonce $r \in \mathsf{R}$ of her choosing and generate a white-box implementation $[E_k^r] = \mathbf{C}_{\mathcal{E}}(k, r)$ by herself.

In a *chosen plaintext attack* (CPA) the adversary can encrypt plaintexts of her choice under $E(k, \cdot)$. Indeed, even though the encryption scheme $\mathcal{E}$ is a symmetric primitive, the attacks are defined with respect to the compiler that generates white-box programs implementing $E(k, \cdot)$: given any one of these programs, the adversary can always evaluate it on arbitrary plaintexts at will. So clearly, chosen plaintexts attacks cannot be avoided, very much like in the public-key encryption setting.

In a *chosen ciphertext attack* (CCA), in addition to the challenge white-box implementation $[E_k^r]$, we give $\mathcal{A}$ access to a decryption oracle $D(k, \cdot)$, *i.e.* she can send decryption queries $c_1, \ldots, c_q \in \mathsf{C}$ adaptively to the oracle and be returned the corresponding plaintexts $m_1, \ldots, m_q \in \mathsf{M}$ where $m_i = D(k, c_i)$. Notice that this attack includes the CPA attack when $q = 0$.

In a *recompilation attack* (RCA), in addition to the challenge white-box implementation $[E_k^r]$, we give $\mathcal{A}$ access to a recompiling oracle $\mathbf{C}_{\mathcal{E}}(k, \mathsf{R})$ that generates other programs $[E_k^{r'}]$ with key k for adversarially unknown random nonces $r' \xleftarrow{\$} \mathsf{R}$. In other words, we give $\mathcal{A}$ the ability to observe other programs compiled with the same key and different nonces.

In a *chosen ciphertext and recompilation attack* (CCA+RCA) we give $\mathcal{A}$ (the challenge white-box implementation $[E_k^r]$ and) simultaneous access to a decryption oracle $D(k, \cdot)$ and a recompiling oracle $\mathbf{C}_{\mathcal{E}}(k, \mathsf{R})$, both parametrized with the same key k.

Remark 2. We emphasize that the recompilation attack model is *not* artificial when dealing with white-box cryptography. Indeed, it seems reasonable to assume that user-related values can be embedded in the random nonce $r \in \mathsf{R}$ used to compile a (user-specific) white-box implementation. Thus a coalition of malicious users can be modeled as a single adversary with (possibly limited) access to a recompiling oracle producing white-box implementations under fresh random nonces $r' \in \mathsf{R}$.

Remark 3. Notice that the recompilation attack may come in other flavors: the random nonce $r' \in \mathsf{R}$ could be adversarially known or even chosen. Typically, in a *chosen recompilation attack* (CRCA), $\mathcal{A}$ is given access to a recompiling oracle $\mathbf{C}_{\mathcal{E}}(k, \cdot)$ that generates other programs $[E_k^{r'}]$ with key k for nonces $r' \in \mathsf{R}$ of her choice. In the following, we will not focus on this (stronger) attack model, as it seems much harder to achieve: having access to the randomness of the compiler could prove fatal for the security of the compiler. We mention, however, that it would be of great interest to design a compiler that achieves resistance even in this extreme adversarial model.

3.2 The Prime Goal: Unbreakability

Chow *et al.* stated in [10,11] that the first security objective of white-box cryptography is, given a program $[E_k]$, to preserve the privacy of the key k embedded in the program (see also [16, Q1] and [30, Definition 2]). We define the following game, illustrated on Figure 1, to capture that intuition:

1. randomly generate a key $k \leftarrow K()$ and a nonce $r \overset{\$}{\leftarrow} \mathsf{R}$,
2. the adversary $\mathcal{A}$ is run on input $[E_k^r] = \mathbf{C}_{\mathcal{E}}(k, r)$,
3. $\mathcal{A}$ returns a guess $\hat{k} \in \mathsf{K}$,
4. $\mathcal{A}$ succeeds if $\hat{k} = k$.

Notice that at Step 2, the adversary may have access to the decryption oracle $D(k, \cdot)$ or to the recompiling oracle $\mathbf{C}_{\mathcal{E}}(k, \mathsf{R})$, or both, depending on the attack model.

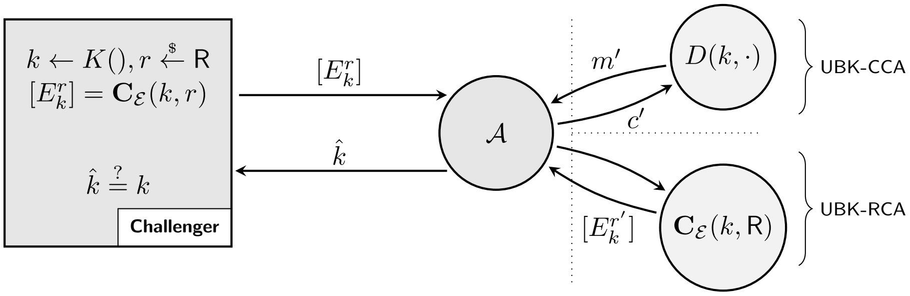

$k \leftarrow K(), r \overset{\$}{\leftarrow} \mathsf{R}$
$[E_k^r] = \mathbf{C}_{\mathcal{E}}(k, r)$
$\hat{k} \overset{?}{=} k$
Challenger
$[E_k^r]$
$\hat{k}$
$\mathcal{A}$
m'
c'
$D(k, \cdot)$
UBK-CCA
$[E_k^{r'}]$
$\mathbf{C}_{\mathcal{E}}(k, \mathsf{R})$
UBK-RCA

Fig. 1. Illustration of the security game UBK-ATK

Let us define more concisely and precisely the notion of unbreakability with respect to the attack model ATK (CPA, CCA, RCA or CCA+RCA).

Definition 1 (Unbreakability). *Let $\mathcal{E}$ be a symmetric encryption scheme as above, $\mathbf{C}_{\mathcal{E}}$ a white-box compiler for $\mathcal{E}$ and let $\mathcal{A}$ be an adversary. For $\mathsf{ATK} \in \{\mathsf{CPA}, \mathsf{CCA}, \mathsf{RCA}, \mathsf{CCA} + \mathsf{RCA}\}$, we define*

$$\mathsf{Succ}_{\mathcal{A},\mathbf{C}_{\mathcal{E}}}^{\mathsf{UBK\text{-}ATK}} \overset{\text{def}}{=} \Pr\left[k \leftarrow K()\,;\ r \overset{\$}{\leftarrow} \mathsf{R}\,;\ [E_k^r] = \mathbf{C}_{\mathcal{E}}(k, r)\,;\ \hat{k} \leftarrow A^{\mathcal{O}}([E_k^r]) : \hat{k} = k\right]$$

where

$$\begin{array}{ll} \mathcal{O}(\cdot) = \epsilon & \textit{if } \mathsf{ATK} = \mathsf{CPA} \\ \mathcal{O}(\cdot) = D(k, \cdot) & \textit{if } \mathsf{ATK} = \mathsf{CCA} \\ \mathcal{O}(\cdot) = \mathbf{C}_{\mathcal{E}}(k, \mathsf{R}) & \textit{if } \mathsf{ATK} = \mathsf{RCA} \\ \mathcal{O}(\cdot) = \{D(k, \cdot), \mathbf{C}_{\mathcal{E}}(k, \mathsf{R})\} & \textit{if } \mathsf{ATK} = \mathsf{CCA} + \mathsf{RCA}\ . \end{array}$$

We say that $\mathbf{C}_{\mathcal{E}}$ is (τ, ε)-secure in the sense of UBK-ATK *if for any adversary $\mathcal{A}$ running in time at most τ, $\mathsf{Succ}_{\mathcal{A},\mathbf{C}_{\mathcal{E}}}^{\mathsf{UBK\text{-}ATK}} \leqslant \varepsilon$.*

Note that in our setting, a total break requires the adversary to output the whole key k embedded into $[E_k^r]$. Basing UBK on the semantic security of k makes no sense here since it is straightforward to ascertain, for some guess $\hat{k}$, that $\hat{k} = k$ by just checking whether the value returned by $[E_k^r](m)$ is equal to $E(\hat{k}, m)$ for sufficiently many plaintext(s) $m \in \mathsf{M}$. In other words, the distributions $\{k, [E_k^r]\}_{k \in \mathsf{K}, r \in \mathsf{R}}$ and $\{k', [E_k^r]\}_{(k,k') \in \mathsf{K}^2, r \in \mathsf{R}}$ are computationally distinguishable. As a result, one cannot prevent some information leakage about k from $[E_k^r]$, whatever the specification of the compiler $\mathbf{C}_{\mathcal{E}}$.

Remark 4. Although not required in the above definition, for a white-box compiler to be cryptographically sound, one would require that there exist some security parameter λ such that ε/τ be exponentially small in λ and $\mathsf{size}\,([E_k])$ and $\mathsf{time}\,([E_k](\cdot))$ be polynomial in λ. Otherwise said, one aims to get a negligible ε/τ while keeping fair $\mathsf{size}\,([E_k])$ and $\mathsf{time}\,([E_k](\cdot))$.

3.3 Security Notions Really Needed in Applications

When satisfied, unbreakability ensures that an adversary cannot extract the secret key of a randomly generated white-box implementation. Therefore any party should have to execute the program rather than simulating it with the secret key. While this property is the very least that can be expected from white-box cryptography, it is rather useless on its own. Indeed, knowing the white-box program amounts to knowing the key in some sense since it allows one to process the encryption without restriction. As discussed in [30, Sect. 3.1.3], an attacker only needs to isolate the cryptographic code in the implementation. This is a common threat in DRM applications, which is known as *code lifting.* Although some countermeasures can make code lifting a tedious task[4] it is reasonable to assume that sooner or later a motivated attacker would eventually recover the cryptographic code. That is why, in order to make the white-box compilation useful, the availability of the white-box program should restrict the adversary capabilities compared to the availability of the secret key.

One-Wayness. A natural restriction is that although the white-box implementation allows one to encrypt at will, it should not enable decryption. In other words, it should be difficult to invert the program computations. In that case, the program is said to be *one-way*, to keep consistency with the notion of one-wayness (for a function or a cryptosystem) traditionally used in cryptography. As already noted in [16], a white-box compiler achieving one-wayness is of great interest as it turns a symmetric encryption scheme into a public-key encryption scheme. This is also one of the many motivations to design methods for general obfuscation [1,13].

Incompressibility of Programs. Another argument often heard in favor of white-box cryptography is that a white-box program is less convenient to store and exchange than a

[4] A work around to circumvent code lifting was proposed in [10,11] and consists in working with encoded variants (see also discussion in [30, Sect. 3.2.3]). Namely, instead of implementing $E(k, \cdot)$, one produces a white-box implementation that is functionally equivalent to the encoded primitive $E'(k, \cdot) = G \circ E(k, \cdot) \circ F^{-1}$, where F and G are randomly selected bijections. The annihilating encodings F and G^{-1} are then embedded in other parts of the application such that they are hard to isolate and identify.

mere secret key due to its bigger size. As formulated in [30, Sect. 3.1.3], white-box cryptography allows to "hide a key in an even bigger key". For instance, Chow *et al.* implementation of AES [11] makes use of 800 KB of look-up tables, which represents a significant overhead compared to a 128-bit key. Suppose this implementation was unbreakable in the sense of Definition 1 (which we know to be false [3]), the question that would arise would be: what is the computationally achievable minimum size of a program functionally equivalent to this implementation? When a program is hard to compress beyond a certain prescribed size, we shall say that this program is *incompressible.* Section 6 shows an example of computationally incompressible programs for symmetric encryption.

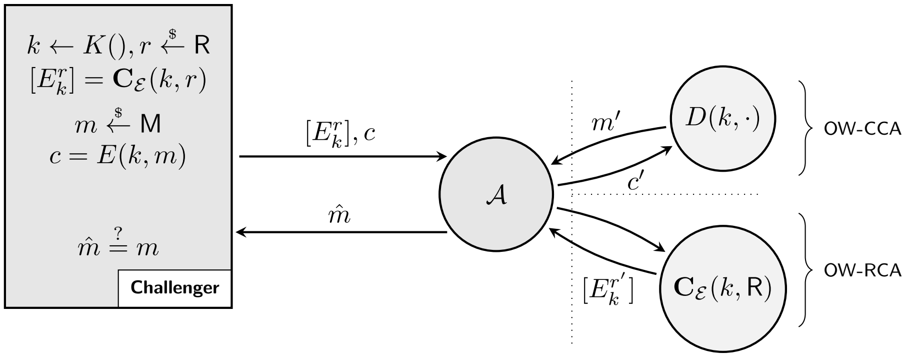
Traceability of Programs. It is often heard that white-box compilation can provide traceability (see for instance [30, Sect. 5.5.1]). Specifically, white-box compilation should enable one to derive several functionally equivalent versions of the same encryption (or decryption) program. A typical use case for such a system is the distribution of protected digital content where every legitimate user gets a different version of some decryption software. If a malicious user shares its own program (*e.g.* over the Internet), then one can trace the so-called *traitor* by identifying its unique copy of the program. However, in a white-box context, a user can easily transform its version of the program while keeping the same functionality. Therefore to be effective, the tracing should be robust to such transformations, even in the case where several malicious users collude to produce an untraceable software. We show in Section 7 how to achieve such a robust tracing from a compiler that can *hide* functional perturbations in a white-box program. Accordingly, we define new security notions for such a white-box compiler. Combined with our tracing scheme, a compiler achieving these security notions is shown to provide traceable white-box programs.

4 One-Wayness

An adversarial goal of interest in white-box cryptography consists, given a white-box implementation $[E_k^r]$, in recovering the plaintext of a given ciphertext with respect to the embedded key k. This security notion is even essential when white-box implementations are deployed as an asymmetric primitive [16, Q4]. We define the following security game, illustrated on Figure 2, to capture that intuition:

1. randomly select a key $k \leftarrow K()$ and a nonce $r \overset{\$}{\leftarrow} \mathsf{R}$,
2. generate the white box program $[E_k^r] = \mathbf{C}_{\mathcal{E}}(k, r)$,
3. randomly select a plaintext $m \overset{\$}{\leftarrow} \mathsf{M}$
4. compute its encryption $c = E(k, m)$,
5. the adversary $\mathcal{A}$ is run on inputs $[E_k^r]$ and c,
6. $\mathcal{A}$ returns a guess $\hat{m}$,
7. $\mathcal{A}$ succeeds if $\hat{m} = m$.

Notice that at Step 5, the adversary may have access to the decryption oracle $D(k, \cdot)$ or to the recompiling oracle $\mathbf{C}_{\mathcal{E}}(k, \mathsf{R})$ (or both) depending on the attack model. When $\mathcal{A}$ is



$k \leftarrow K(), r \xleftarrow{\$} \mathsf{R}$
$[E_k^r] = \mathbf{C}_{\mathcal{E}}(k,r)$
$m \xleftarrow{\$} \mathsf{M}$
$c = E(k,m)$
$\hat{m} \overset{?}{=} m$
Challenger
$[E_k^r], c$
$\hat{m}$
$\mathcal{A}$
m'
c'
$D(k,\cdot)$
OW-CCA
$[E_k^{r'}]$
$\mathbf{C}_{\mathcal{E}}(k,\mathsf{R})$
OW-RCA

Fig. 2. Illustration of the security game OW-ATK

given access to the decryption oracle, the challenge ciphertext c itself shall be rejected by the oracle.

Let us define more precisely the notion of one-wayness with respect to the attack model ATK.

Definition 2 (One-Wayness). *Let $\mathcal{E}$ be a symmetric encryption scheme as above, $\mathbf{C}_{\mathcal{E}}$ a white-box compiler for $\mathcal{E}$ and $\mathcal{A}$ an adversary. For* $\mathsf{ATK} \in \{\mathsf{CPA}, \mathsf{CCA}, \mathsf{RCA}, \mathsf{CCA}+\mathsf{RCA}\}$, *let*

$$\mathsf{Succ}^{\mathsf{OW\text{-}ATK}}_{\mathcal{A},\mathbf{C}_{\mathcal{E}}} \overset{\text{def}}{=} \Pr\left[\begin{array}{c} k \leftarrow K()\,;\ r \xleftarrow{\$} \mathsf{R}\,;\ [E_k^r] = \mathbf{C}_{\mathcal{E}}(k,r)\,; \\ m \xleftarrow{\$} \mathsf{M}\,;\ c = E(k,m)\,;\ \hat{m} \leftarrow \mathcal{A}^{\mathcal{O}}([E_k^r], c) \end{array} : \hat{m} = m\right]$$

where

$$\begin{array}{ll} \mathcal{O}(\cdot) = \epsilon & \textit{if}\ \mathsf{ATK} = \mathsf{CPA} \\ \mathcal{O}(\cdot) = D(k,\cdot) & \textit{if}\ \mathsf{ATK} = \mathsf{CCA} \\ \mathcal{O}(\cdot) = \mathbf{C}_{\mathcal{E}}(k,\mathsf{R}) & \textit{if}\ \mathsf{ATK} = \mathsf{RCA} \\ \mathcal{O}(\cdot) = \{D(k,\cdot), \mathbf{C}_{\mathcal{E}}(k,\mathsf{R})\} & \textit{if}\ \mathsf{ATK} = \mathsf{CCA}+\mathsf{RCA}\,. \end{array}$$

We say that $\mathbf{C}_{\mathcal{E}}$ is (τ,ε)-secure in the sense of OW-ATK *if $\mathcal{A}$ running in time at most τ implies $\mathsf{Succ}^{\mathsf{OW\text{-}ATK}}_{\mathcal{A},\mathbf{C}_{\mathcal{E}}} \leqslant \varepsilon$.*

Similarly to the unbreakability notion, it is obvious that any incorrect guess $\hat{m}$ on m can be rejected by comparing the value returned by $[E_k^r](\hat{m})$ with c. In other words, the two distributions

$$\{[E_k^r], E(k,m), m\}_{k\in\mathsf{K}, r\in\mathsf{R}, m\in\mathsf{M}} \quad \text{and} \quad \{[E_k^r], E(k,m), m'\}_{k\in\mathsf{K}, r\in\mathsf{R}, m,m'\in\mathsf{M}}$$

are easily distinguishable. Moreover, there is an easy reduction from OW-ATK to UBK-ATK. Clearly, extracting k from $[E_k]$ enables one to use it and the challenge as inputs to the (publicly available) decryption function $D(\cdot,\cdot)$ and thus to recover m.

5 Incompressibility of White-Box Programs

In this section, we formalize the notion of incompressibility for a white-box compiler. What we mean by incompressibility here is the hardness, given a (large) compiled program $[E_k]$, of coming up with a significantly smaller program functionally close to $E(k,\cdot)$. A typical example is when a content provider distributes a large encryption program (*e.g.* 100 GB or more) and wants to make sure that no smaller yet equivalent program can be redistributed by subscribers to illegitimate third parties. The content provider cannot prevent the original program from being shared *e.g.* over the Internet; however, if compiled programs are provably incompressible then redistribution may be somewhat discouraged by the size of transmissions.

We define (λ,δ)-INC as the adversarial goal that consists, given a compiled program $[E_k]$ with $\mathsf{size}([E_k]) \gg \lambda$, in building a smaller program P that remains satisfactorily functional, *i.e.* such that

$$\mathsf{size}(P) < \lambda \qquad \text{and} \qquad P \in \delta\text{-}\mathsf{prog}(E(k,\cdot)) \ .$$

This is formalized by the following game, also illustrated on Figure 3:

1. randomly select $k \leftarrow K()$ and $r \xleftarrow{\$} \mathsf{R}$,
2. compile $[E_k^r] = \mathbf{C}_{\mathcal{E}}(k,r)$,
3. run $\mathcal{A}$ on input $[E_k^r]$,
4. $\mathcal{A}$ returns some program P,
5. $\mathcal{A}$ succeeds if $\Delta(P, E(k,\cdot)) \leqslant \delta$ and $\mathsf{size}(P) < \lambda$.

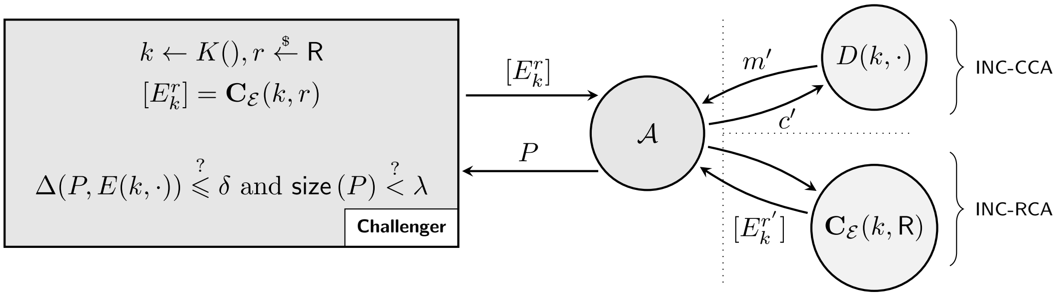

$k \leftarrow K(), r \xleftarrow{\$} \mathsf{R}$
$[E_k^r] = \mathbf{C}_{\mathcal{E}}(k,r)$
$\Delta(P, E(k,\cdot)) \overset{?}{\leqslant} \delta$ and $\mathsf{size}(P) \overset{?}{<} \lambda$
Challenger
$[E_k^r]$
P
$\mathcal{A}$
m'
c'
$D(k,\cdot)$
INC-CCA
$[E_k^{r'}]$
$\mathbf{C}_{\mathcal{E}}(k,\mathsf{R})$
INC-RCA

Fig. 3. Illustration of the security game (λ,δ)-INC

Definition 3 ((λ,δ)-Incompressibility). *Let $\mathcal{E}$ be a symmetric encryption scheme, $\mathbf{C}_{\mathcal{E}}$ a white-box compiler for $\mathcal{E}$ and $\mathcal{A}$ an adversary. For $\mathsf{ATK} \in \{\mathsf{CPA}, \mathsf{CCA}, \mathsf{RCA}, \mathsf{CCA}+\mathsf{RCA}\}$, let*

$$\mathsf{Adv}_{\mathcal{A},\mathbf{C}_{\mathcal{E}}}^{(\lambda,\delta)\text{-}\mathsf{INC}\text{-}\mathsf{ATK}} \overset{\mathrm{def}}{=} \Pr\left[\begin{array}{c} k \leftarrow K()\,;\ r \xleftarrow{\$} \mathsf{R}\,; \\ [E_k^r] = \mathbf{C}_{\mathcal{E}}(k,r)\,; \\ P \leftarrow \mathcal{A}^{\mathcal{O}}([E_k^r]) \end{array} : (\Delta(P, E(k,\cdot) \leqslant \delta) \wedge (\mathsf{size}(P) < \lambda)\right]$$

where

$$\begin{array}{ll} \mathcal{O}(\cdot) = \epsilon & \textit{if } \mathsf{ATK} = \mathsf{CPA} \\ \mathcal{O}(\cdot) = D(k,\cdot) & \textit{if } \mathsf{ATK} = \mathsf{CCA} \\ \mathcal{O}(\cdot) = \mathbf{C}_{\mathcal{E}}(k,\mathsf{R}) & \textit{if } \mathsf{ATK} = \mathsf{RCA} \\ \mathcal{O}(\cdot) = \{D(k,\cdot), \mathbf{C}_{\mathcal{E}}(k,\mathsf{R})\} & \textit{if } \mathsf{ATK} = \mathsf{CCA}+\mathsf{RCA}\ . \end{array}$$

We say that $\mathbf{C}_{\mathcal{E}}$ is (τ,ε)-secure in the sense of (λ,δ)-INC-ATK if having $\mathcal{A}$ running in time at most τ implies that $\mathsf{Adv}_{\mathcal{A},\mathbf{C}_{\mathcal{E}}}^{(\lambda,\delta)\text{-}\mathsf{INC}\text{-}\mathsf{ATK}} \leqslant \varepsilon$.

Notice that for some values of λ and δ, the (λ, δ)-incompressibility may be trivially broken. For example, the problem is trivial for $\delta = 1$ as the user can always construct any program smaller than λ bits with outputs unrelated to $E(k, \cdot)$. Even though the definition allows any $\delta \in [0, 1]$, the notion makes more sense (and surely is harder to break) when δ is taken small enough. In that case, the adversary has to output a program which correctly encrypts nearly all plaintexts (or at least a significant fraction).

It seems natural to hope that a reduction exists from $\mathsf{INC\text{-}ATK}$ to $\mathsf{UBK\text{-}ATK}$: intuitively, extracting k from $[E_k]$ enables one to build a small program that implements $E(k, \cdot)$. Let $\lambda(k)$ be the size of that program; it is easily seen that $\lambda(k)$ is lower-bounded by

$$\lambda_0 = H(k) + \mathsf{size}\,(P_E)$$

where $H(k)$ is the average number of bits needed to represent the key k and P_E the smallest known program that implements the generic encryption function $E(\cdot, \cdot)$ that takes k, m as inputs and returns $E(k, m)$. When $\lambda_0 \leqslant \lambda$, a total break (*i.e.* recovering the key k) will allow to break $(\lambda, 0)$-incompressibility by outputting a program P composed of P_E and a string representing k, which will be of size at most λ_0 ($\leqslant \lambda$).

On the other hand, denoting

$$\lambda^+ = \sup_{k \in \mathsf{K}, r \in \mathsf{R}} \mathsf{size}\,([E_k^r]) \quad \text{and} \quad \lambda^- = \inf_{k \in \mathsf{K}, r \in \mathsf{R}} \mathsf{size}\,([E_k^r]) \ ,$$

we also see that when $\lambda \geqslant \lambda^+$, the challenge program $[E_k^r]$ given to $\mathcal{A}$ already satisfies the conditions of a satisfactorily compressed program and $\mathcal{A}$ may then return $P = [E_k^r]$ as a solution. (λ, δ)-INC is therefore trivial to break in that case. However, (λ, δ)-incompressibility for $\lambda \leqslant \lambda^-$ may not be trivial to break. To conclude, the (λ, δ)-incompressibility notion makes sense in practice for parameters $\lambda \in (\lambda_0, \lambda^-)$ and δ close to 0.

6 A Provably One-Way and Incompressible White-Box Compiler

In this section, we give an example of a symmetric encryption scheme for which there exists a efficient one-way and incompressible white-box compiler. This example is a symmetric-key variant of the RSA cryptosystem [26]. The one-wayness and incompressibility properties of the compiler are provably achieved based on standard hardness assumptions related to the integer factoring problem.

One-way Compilers from Public-Key Encryption. It is worthwhile noticing that any *one-way public-key* encryption scheme straightforwardly gives rise to a symmetric encryption scheme for which a one-way compiler exists. The symmetric key is defined as the secret key of the asymmetric encryption scheme and encryption is defined as the function deriving the public key from the secret key composed with the encryption procedure. The white-box compiler then simply produces a program evaluating the encryption algorithm with the public key embedded in it. The one-wayness of the compiler comes directly from the one-wayness of the asymmetric scheme. Such an example of a one-way compiler is given in [28, Theorem 3],[30, Sect. 4.8.2].

We present hereafter another compiler obtained from the RSA cryptosystem and whose one-wayness straightforwardly holds by construction. The main interest of our example is to further satisfy $(\lambda, 0)$-incompressibility for any arbitrary λ. We first recall some background on RSA groups.

6.1 RSA Groups

We consider a (multiplicative) group $\mathcal{G}$ of unknown order ω, also called an *RSA group*. A typical construction for $\mathcal{G}$ is to take the group of invertible integers modulo a composite number or a carefully chosen elliptic curve over a ring. Practical RSA groups are known to be efficiently samplable in the sense that there exists a group generation algorithm $\mathbb{G}$ which, given a security parameter $n \in \mathbb{N}$, outputs the public description $\mathsf{desc}(\mathcal{G})$ of a random group $\mathcal{G}$ together with its order ω. Efficient means that the random selection

$$(\mathsf{desc}(\mathcal{G}), \omega) \leftarrow \mathbb{G}(1^n)$$

takes time polynomial in n. The parameter n determine the size of the returned order (*i.e.* $|\omega| = n$) and hence tunes the hardness of breaking the group. For security reasons, we require the returned order ω to have a low smoothness. More specifically, we require that it satisfy $\varphi(\omega) \geqslant \frac{1}{3}\omega$, where φ denotes the Euler's totient function.[5] The group descriptor $\mathsf{desc}(\mathcal{G})$ intends to contain all the necessary parameters for performing group operations. Obviously ω is excluded from the group description.

In the following, we shall make the usual hardness assumptions for RSA group generators. Namely, we assume that the groups sampled by $\mathbb{G}$ have the following properties (formal definitions for these security notions are provided in Appendix A):

Unbreakability – $\mathsf{UBK}[\mathbb{G}]$:
It is hard to compute the secret order ω of $\mathcal{G}$ from $\mathsf{desc}(\mathcal{G})$.

Hardness of Extracting Orders – $\mathsf{ORD}[\mathbb{G}]$:
It is hard to compute the order of a random group element $x \xleftarrow{\$} \mathcal{G}$ (or a multiple thereof) from $\mathsf{desc}(\mathcal{G})$.

Hardness of Extracting Roots – $\mathsf{RSA}[\mathbb{G}]$:
For a random integer $e \in [0, \omega)$ such that $\gcd(e, \omega) = 1$, it is hard to compute the e-th root of a random group element $x \in \mathcal{G}$ from e and $\mathsf{desc}(\mathcal{G})$.

Intuition tells that breaking a random group may be significantly easier when one can make calls to an oracle performing an operation that seems to require the knowledge of the hidden group order ω such that the extraction of e-th roots or computing the order of group elements. It appears however that these two oracles are not equivalently powerful since for practical RSA group generators, well-known results state that (see Appendix A):

Fact 1. Extracting orders in a random group is equivalent to breaking that group, *i.e.* $\mathsf{ORD}[\mathbb{G}]$ is hard iff $\mathsf{UBK}[\mathbb{G}]$ is hard.

[5] In practice, it is well known how to generate such groups. For instance, the multiplicative group $\mathbb{Z}_{pq}^*$ with p and q being *safe primes* has order $\omega = (p-1)(q-1)$ with $\varphi(\omega) \approx \frac{1}{2}\omega$.

Fact 2. Extracting roots in a random group does not seem to make that group any easier to break.

6.2 The White-Box Compiler

We consider the symmetric encryption scheme $\mathcal{E} = (\mathsf{K}, \mathsf{M}, \mathsf{C}, K, E, D)$ where:

1. $\mathcal{E}$ makes use of a security parameter $n \in \mathbb{N}$,
2. $K()$ randomly selects a group $(\mathsf{desc}(\mathcal{G}), \omega) \leftarrow \mathbb{G}(1^n)$ and a public exponent $e \in [0, \omega)$ such that $\gcd(e, \omega) = 1$, and returns $k = (\mathsf{desc}(\mathcal{G}), \omega, e)$,
3. plaintexts and ciphertexts are group elements *i.e.* $\mathsf{M} = \mathsf{C} = \mathcal{G}$,
4. given a key $k = (\mathsf{desc}(\mathcal{G}), \omega, e)$ and a plaintext $m \in \mathcal{G}$, $E(k, m)$ computes $m^{e \bmod \omega}$ in the group and returns that value,
5. given a key $k = (\mathsf{desc}(\mathcal{G}), \omega, e)$ and a ciphertext $c \in \mathcal{G}$, $D(k, c)$ computes $c^{\frac{1}{e} \bmod \omega}$ in the group and returns that value.

It is clear that $D(k, E(k, m)) = m$ for any $k \in \mathsf{K}$ and $m \in \mathsf{M}$. Our white-box compiler $\mathbf{C}_{\mathcal{E}}$ is then defined as follows:

1. $\mathbf{C}_{\mathcal{E}}$ makes use of an additional security parameter $h \in \mathbb{N}$,
2. the randomness space R is the integer set $[0, 2^h/\omega)$,
3. we define the *blinded exponent* f with respect to the public exponent e and a random nonce $r \in \mathsf{R}$ as the integer $f = e + r \cdot \omega$,
4. given a key $k = (\mathsf{desc}(\mathcal{G}), \omega, e) \in \mathsf{K}$, and a random nonce $r \in \mathsf{R}$, our white-box compiler $\mathbf{C}_{\mathcal{E}}$ generates a program $[E_k]$ which simply embeds $\mathsf{desc}(\mathcal{G})$ and f and computes m^f for any input $m \in \mathcal{G}$.

According to the above definition, we clearly have that the white-box program $[E_k]$ is a functional program with respect to the encryption function $E(k, \cdot)$. Moreover, we state:

Theorem 1. *The white-box compiler* $\mathbf{C}_{\mathcal{E}}$ *is* $\mathsf{UBK\text{-}CPA}$ *secure under the assumption that* $\mathsf{UBK}[\mathbb{G}]$ *is hard, and* $\mathsf{OW\text{-}CPA}$ *secure under the assumption that* $\mathsf{RSA}[\mathbb{G}]$ *is hard.*

Proof. Given $\mathsf{desc}(\mathcal{G})$, the reduction selects a random integer $f \in [0, 2^h)$ and generates the white-box program P computing m^f for any $m \in \mathcal{G}$. Assuming that f is co-prime to ω (which occurs with probability $\frac{\varphi(\omega)}{\omega} \geqslant \frac{1}{3}$), P is identical to the white-box program $[E_k]$ generated by $\mathbf{C}_{\mathcal{E}}$ on input $r = \lfloor f/\omega \rfloor$ and $k = (\mathsf{desc}(\mathcal{G}), \omega, e)$ with $e = f \bmod \omega$. Any adversary able to extract $k = (\mathsf{desc}(\mathcal{G}), \omega, e)$ from $[E_k]$ then recovers the order ω of $\mathcal{G}$, and can thus be used to solve $\mathsf{UBK}[\mathbb{G}]$. Therefore $\mathbf{C}_{\mathcal{E}}$ is unbreakable if $\mathsf{UBK}[\mathbb{G}]$ is hard. Similarly, it is easily seen that any adversary able to break the one-wayness game *i.e.* given $[E_k]$ and a challenge c, can recover m such that $c = m^e \in \mathcal{G}$, can be used to solve $\mathsf{RSA}[\mathbb{G}]$ in a straightforward fashion. □

6.3 Proving Incompressibility under Chosen Plaintext Attacks

We now show that $\mathbf{C}_{\mathcal{E}}$ is $(\lambda, 0)$-$\mathsf{INC\text{-}CPA}$ secure under $\mathsf{UBK}[\mathbb{G}]$ as long as the security parameter h is slightly greater than λ. We actually show a slightly weaker result: our reduction assumes that the program P output by the adversary is *algebraic*. An algebraic program P (see [5,25]) with respect to group $\mathcal{G}$ has the property that each and every group element $y \in \mathcal{G}$ output by P is computed as a linear combination of all the group elements $x_1, \ldots, x_t$ that were given to P as input in the same execution. Relying on the definition of [25], P must then admit an efficient extractor $\mathsf{Extract}$ (running in time τ_{Ex}) which, given the code of P as well as all its inputs and random tape for some execution, returns the coefficients α_i such that $y = x_1^{\alpha_1} \cdots x_t^{\alpha_t}$.

Theorem 2. *For every $h > \lambda + \log_2(3)$, the compiler $\mathbf{C}_{\mathcal{E}}$ is $(\tau_{\mathcal{A}}, \varepsilon_{\mathcal{A}})$-secure in the sense of* $(\lambda, 0)$-$\mathsf{INC\text{-}CPA}$ *under the assumption that* $\mathsf{ORD}[\mathbb{G}]$ *is (τ, ε)-hard, with*

$$\tau_{\mathcal{A}} = \tau - \tau_{\mathsf{Ex}} \quad \textit{and} \quad \varepsilon_{\mathcal{A}} < \frac{3}{1 - 3 \cdot 2^{\lambda - h}}\, \varepsilon \ .$$

Proof. We build a reduction $\mathcal{R}$ which, given an adversary $\mathcal{A}$ running in time $\tau_{\mathcal{A}}$ with a non-negligible success

$$\varepsilon_{\mathcal{A}} = \mathsf{Succ}_{\mathcal{A}, \mathbf{C}_{\mathcal{E}}}^{(\lambda,0)\text{-}\mathsf{INC\text{-}CPA}}$$

breaks the security game $\mathsf{ORD}[\mathbb{G}]$ in time $\tau = \tau_{\mathcal{A}} + \tau_{\mathsf{Ex}}$ with probability $\varepsilon > \frac{1}{3}(1 - 3 \cdot 2^{\lambda - h})\varepsilon_{\mathcal{A}}$.

From $(\mathsf{desc}\,(\mathcal{G})\,, \omega) \leftarrow \mathbb{G}(1^n)$ and $m_* \xleftarrow{\$} \mathcal{G}$, $\mathcal{R}$ is given $\mathsf{desc}\,(\mathcal{G})$, m_* and returns $\mathsf{ord}\,(m_*)$. The reduction starts by simulating the random selection $k \leftarrow K()\,;\ r \xleftarrow{\$} \mathsf{R}\,;\ [E_k] = \mathbf{C}_{\mathcal{E}}(k, r)$. This is simply done by randomly choosing an integer $f \in [0, 2^h)$ and building the white-box program P that computes m^f for any $m \in \mathcal{G}$. If f is co-prime to ω (which occurs with probability $\frac{\varphi(\omega)}{\omega} \geqslant \frac{1}{3}$), then P is identical to the white-box program $[E_k]$ generated by $\mathbf{C}_{\mathcal{E}}$ on input $r = \lfloor f/\omega \rfloor$ and $k = (\mathsf{desc}\,(\mathcal{G})\,, \omega, e)$ with $e = f \bmod \omega$. Otherwise, if f is not co-prime to ω, then the reduction fails.

Now $\mathcal{R}$ runs $\mathcal{A}([E_k])$ to get some new program P which must satisfy the success conditions of the $(\lambda, 0)$-$\mathsf{INC\text{-}CPA}$ game with probability $\varepsilon_{\mathcal{A}}$, in which case P is of size $\mathsf{size}\,(P) < \lambda$ and for any $m \in \mathcal{G}$, $P(m) = E(k, m) = m^f$. By assumption P is algebraic; therefore running $\mathsf{Extract}$ on P, m_* and any random tape $\rho \in \{0,1\}^*$ yields some α such that $P(m_*, \rho)$ outputs m_*^{α}. Since for every $m \in \mathcal{G}$, we have $P(m) = m^f$, we deduce $m_*^{\alpha} = m_*^f$, and hence $f - \alpha$ is a multiple of $\mathsf{ord}\,(m_*)$. Then $\mathcal{R}$ simply returns $f - \alpha$. The probability ε that $\mathcal{R}$ succeeds is hence the joint probability that f is co-prime to ω, that $\mathcal{A}$ succeeds and that α is different from f, that is:

$$\varepsilon = \frac{1}{3} \cdot \varepsilon_{\mathcal{A}} \cdot \Pr[\alpha \neq f \mid \gcd(f, \omega) = 1 \wedge \mathcal{A} \text{ succeeds}] \ .$$

To complete the proof, we show hereafter that for any given random tape value ρ_0, the recovered α is different from f with probability greater than $(1 - 3 \cdot 2^{\lambda - h})$. Let us denote by $\mathsf{F} \subseteq [0, 2^h)$ the set of integers lower than 2^h and co-prime to ω. By definition we have

$|\mathsf{F}| \geqslant 2^h/3$. Let us further denote by $\mathsf{B} \subseteq \mathsf{F}$, the set of integers $\beta \in \mathsf{F}$ for which there exists a program P with $\mathsf{size}\,(P) < \lambda$ such that $\mathsf{Extract}(P, m_*, \rho) = \beta$. Let us eventually denote:

$$P_\beta = \underset{\substack{P \\ \mathsf{Extract}(P,m_*,\rho)=\beta}}{\arg\min} \; \mathsf{size}\,(P) \quad \text{and} \quad s_\beta = \mathsf{size}\,(P_\beta) \ .$$

The program P_β can be seen as a binary word of length s_β coding the integer β (with efficient decoding function $\mathsf{Extract}(\cdot, m_*, \rho)$). Therefore, Shannon's source coding Theorem [29] states that the expected value of s_β is at least the entropy of β. Applying this result to the uniform distribution over B, we deduce

$$\frac{1}{|\mathsf{B}|} \sum_{\beta \in \mathsf{B}} s_\beta \geqslant \log(|\mathsf{B}|) \ .$$

Since by assumption $s_\beta < \lambda$ for any $\beta \in \mathsf{B}$, the above inequality implies $|\mathsf{B}| < 2^\lambda$. We deduce that the probability that f lies in B satisfies $\Pr[f \in \mathsf{B}] = |\mathsf{B}|/|\mathsf{F}| < 3 \cdot 2^{\lambda - h}$. Now since the attacker succeeds, we must have $\mathsf{size}\,(P) < \lambda$ implying $\alpha \in \mathsf{B}$. Then the probability to have $\alpha \neq f$ is at least the probability to have $f \notin \mathsf{B}$ which is greater than $(1 - 3 \cdot 2^{\lambda - h})$. □

Remark 5. The white-box compiler can also be shown to be $(\lambda, 0)$-INC-CCA secure under the (gap) assumption that $\mathsf{ORD}[\mathbb{G}]$ remains hard when $\mathsf{RSA}[\mathbb{G}]$ is easy. The reduction would work similarly but with an oracle solving $\mathsf{RSA}[\mathbb{G}]$ that it would use to simulate decryption queries.

7 Traceability of White-Box Programs

One of the main applications of white-box cryptography is the secure distribution of valuable content through applications enforcing digital rights management (DRM). Namely, some digital content is distributed in encrypted form to legitimate users. A service user may then recover the content in clear using her own private white-box-secure decryption software.

However, by sharing their decryption software, users may collude and try to produce a pirate decryption software *i.e.* a non-registered utility capable of decrypting premium content. Traitor tracing schemes [8,9,24,4] were specifically designed to fight copyright infringement, by enabling a designated authority to recover the identity of at least one of the traitors in the malicious coalition who constructed the rogue decryption software. In this section, we show how to apply some of these techniques to ensure the full traceability of programs assuming that slight perturbations of the programs functionality by the white-box compiler can remain *hidden* to an adversary.

As opposed to previous sections, we interchange the roles of encryption and decryption, considering that for our purpose, user programs would implement decryption rather than encryption.

7.1 Programs with Hidden Perturbations

A program can be made traceable by unnoticeably modifying its functionality. The basic idea is to *perturbate* the program such that it returns an incorrect output for a small set of unknown inputs (which remains a negligible fraction of the input domain). The set of so-called *tracing inputs* varies according to the identity of end users so that running the decryption program over inputs from different sets and checking the returned outputs efficiently reveals the identity of a traitor. We consider tracing schemes that follow this approach to make programs traceable in the presence of pirate coalitions. Of course, one must consider collusions of several users aiming to produce an untraceable program from their own legitimate programs. A tracing scheme that resists such collusions is said to be *collusion-resistant.*

In the context of deterministic symmetric encryption schemes, one can generically describe functional perturbations with the following formalism. Consider a symmetric encryption scheme $\mathcal{E} = (\mathsf{K}, \mathsf{M}, \mathsf{C}, K, E, D)$ under the definition of Section 2. A white-box compiler $\mathbf{C}_\mathcal{E}$ with respect to $\mathcal{E}$ that *supports perturbation* takes as additional input an ordered list of dysfunctional ciphertexts $\boldsymbol{c} = \langle c_1, \ldots, c_u \rangle \in \mathsf{C}^u$ and returns a program

$$[D^r_{k,\boldsymbol{c}}] = \mathbf{C}_\mathcal{E}(k, r; \boldsymbol{c})$$

such that $[D^r_{k,\boldsymbol{c}}](c) = D(k, c)$ for any $c \in \mathsf{C} \setminus \boldsymbol{c}$ and for $i \in [1, u]$, $[D^r_{k,\boldsymbol{c}}](c_i)$ returns some incorrect plaintext randomly chosen at compilation. We will say that $\mathbf{C}_\mathcal{E}$ *hides* functional perturbations when, given a program instance $P = [D^r_{k,\boldsymbol{c}}]$, an adversary cannot extract enough information about the dysfunctional input-output pairs to be able to correct P back to its original functionality. It is shown later that perturbated programs can be made traceable assuming that it is hard to recover the correct output of dysfunctional inputs. This is formalized by the following game:

1. randomly select $k \leftarrow K()$, $m \overset{\$}{\leftarrow} \mathsf{M}$ and $r \overset{\$}{\leftarrow} \mathsf{R}$,
2. compile $[D^r_{k,\langle c \rangle}] = \mathbf{C}_\mathcal{E}(k, r; \langle c \rangle)$ with $c = E(k, m)$,
3. run $\mathcal{A}$ on input $(c, [D^r_{k,\langle c \rangle}])$,
4. $\mathcal{A}$ return some message $\hat{m}$,
5. $\mathcal{A}$ succeeds if $\hat{m} = m$.

Definition 4 (Perturbation-Value Hiding). *Let $\mathcal{E}$ be a symmetric encryption scheme, $\mathbf{C}_\mathcal{E}$ a white-box compiler for $\mathcal{E}$ that supports perturbations, and let $\mathcal{A}$ be an adversary. Let*

$$\mathsf{Succ}^{\mathsf{PVH}}_{\mathcal{A},\mathbf{C}_\mathcal{E}} \overset{\text{def}}{=} \Pr \left[\begin{array}{c} k \leftarrow K()\,;\ m \overset{\$}{\leftarrow} \mathsf{M}\,;\ c = E(k, m)\,; \\ r \overset{\$}{\leftarrow} \mathsf{R}\,;\ [D^r_{k,\langle c \rangle}] = \mathbf{C}_\mathcal{E}(k, r; \langle c \rangle)\,; \\ \hat{m} \leftarrow \mathcal{A}^{\mathcal{O}}(c, [D^r_{k,\langle c \rangle}]) \end{array} : \hat{m} = m \right] .$$

where $\mathcal{O}$ is a recompiling oracle $\mathcal{O}(\cdot) \overset{\text{def}}{=} \mathbf{C}_\mathcal{E}(k, \mathsf{R}; \langle c, \cdot \rangle)$ that takes as input a list of dysfunctional inputs containing c and returns a perturbated program accordingly, under adversarially unknown randomness. The white-box compiler $\mathbf{C}_\mathcal{E}$ is said (τ, ε)-secure in the sense of PVH if $\mathcal{A}$ running in time at most τ implies $\mathsf{Succ}^{\mathsf{PVH}}_{\mathcal{A},\mathbf{C}_\mathcal{E}} \leqslant \varepsilon$.

A second security notion that we will make use of for our tracing construction relates to the intuition that all perturbations should be equally hidden by the white-box compiler. Namely, it should not matter in which order the dysfunctional inputs are given to the compiler: they should all appear equally hard to recover to an adversary. When this property is realized, we say that the compiler achieves *perturbation-index hiding.* We formalize this notion with the following game, where $n > 1$ and $v \in [1, n-1]$ are fixed parameters:

1. randomly select $k \leftarrow K()$,
2. for $i \in [1, n]$, randomly select $m_i \xleftarrow{\$} \mathsf{M}$ and set $c_i = E(k, m_i)$,
3. for $i \in [1, n]$ with $i \neq v$, randomly select $r_i \xleftarrow{\$} \mathsf{R}$ and generate $P_i = \mathbf{C}_{\mathcal{E}}(k, r_i; \langle c_1, \dots, c_i \rangle)$,
4. randomly pick $b \xleftarrow{\$} \{0, 1\}$,
5. run $\mathcal{A}$ on inputs $P_1, \dots, P_{v-1}, P_{v+1}, \dots, P_n$ and (m_{v+b}, c_{v+b}),
6. $\mathcal{A}$ returns a guess $\hat{b}$ and succeeds if $\hat{b} = b$.

Definition 5 (Perturbation-Index Hiding). *Let $\mathcal{E}$ be a symmetric encryption scheme, $\mathbf{C}_{\mathcal{E}}$ a white-box compiler for $\mathcal{E}$ that supports perturbations, and let $\mathcal{A}$ be an adversary. Let*

$$\mathsf{Adv}^{\mathsf{PIH}}_{\mathcal{A}, \mathbf{C}_{\mathcal{E}}} \overset{\text{def}}{=} \left| \Pr \left[\begin{array}{c} k \leftarrow K()\,;\ m_i \xleftarrow{\$} \mathsf{M}\,;\ c_i = E(k, m_i) \text{ for } i \in [1, n] \\ r_i \xleftarrow{\$} \mathsf{R}\,;\ P_i = \mathbf{C}_{\mathcal{E}}(k, r_i; \langle c_1, \dots, c_i \rangle) \text{ for } i \in [1, n], i \neq v\ :\ \hat{b} = b \\ b \xleftarrow{\$} \{0, 1\}\,;\ \hat{b} \leftarrow \mathcal{A}(\{P_i\}_{i \neq v}, m_{v+b}, c_{v+b}) \end{array} \right] - \frac{1}{2} \right| .$$

The white-box compiler $\mathbf{C}_{\mathcal{E}}$ is said to be (τ, ε)-secure in the sense of PIH *if $\mathcal{A}$ running in time at most τ implies $\mathsf{Adv}^{\mathsf{PIH}}_{\mathcal{A}, \mathbf{C}_{\mathcal{E}}} \leqslant \varepsilon$.*

Note that in a PIH-secure white-box compiler, all entries in the list of its dysfunctional inputs can be permuted with no (non-negligible) impact on the security of the compiler.

7.2 A Generic Tracing Scheme

We now give an example of a tracing scheme $\mathcal{T}$ for programs generated by a white-box compiler $\mathbf{C}_{\mathcal{E}}$ that supports hidden perturbations. We formally prove that the identification of at least one traitor is computationally enforced assuming that $\mathbf{C}_{\mathcal{E}}$ is secure in the sense of PVH and PIH, independently of the total number n of issued programs. Under these assumptions, $\mathcal{T}$ therefore resists collusions of up to n users *i.e.* is maximally secure. As usual in traitor-tracing schemes, $\mathcal{T}$ is composed of a setup algorithm $\mathcal{T}$.setup and a tracing algorithm $\mathcal{T}$.trace. These algorithms are defined as follows.

Setup algorithm. A random key $k \xleftarrow{\$} K()$ is generated as well as n random input-output pairs (m_i, c_i) where $m_i \xleftarrow{\$} \mathsf{M}$ and $c_i = E(k, m_i)$ for $i \in [1, n]$. $\mathcal{T}$ keeps $\mathsf{perturbations} = ((m_1, c_1), \dots, (m_n, c_n))$ as private information for later tracing. For $i \in [1, n]$, user i is (securely) given the i-perturbated program $P_i = \mathbf{C}_{\mathcal{E}}(k, r_i; \langle c_1, \dots, c_i \rangle)$ where $r_i \xleftarrow{\$} \mathsf{R}$. It is easily seen that all P_i's correctly decrypt any $c \notin \{c_i, i \in [1, n]\}$. However when $c = c_i$, user programs $P_i, \dots, P_n$ return junk while $P_1, \dots, P_{i-1}$ remain functional. Therefore $\mathcal{T}$ implements a private linear broadcast encryption (PLBE) scheme in the sense of [4].

```
1.  evaluate $\widehat{p_0}$ and $\widehat{p_n}$
2.  set $a = 0$ and $b = n$
3.  while $a \neq b - 1$
    3.1.  set $v = \lceil (a + b)/2 \rceil$
    3.2.  evaluate $\widehat{p_v}$
    3.3.  if $|\widehat{p_v} - \widehat{p_a}| > |\widehat{p_v} - \widehat{p_b}|$ then set $b = v$ else set $a = v$
4.  return $b$ as the identified traitor.
```

Fig. 4. Dichotomic search implemented by $\mathcal{T}$.trace

Tracing algorithm. Given a rogue decryption program Q constructed from a set of user programs $\{P_j \mid j \in T \subseteq [1, n]\}$, $\mathcal{T}$.trace uses its knowledge of k and perturbations to identify a traitor $j \in T$ in $O(\log n)$ evaluations of Q as follows. Since Q is just a program and is therefore stateless, the general tracing techniques of [24,4] are applicable. $\mathcal{T}$.trace makes use of two probability estimators as subroutines:

1. a probability estimator $\widehat{p_0}$ which intends to measure the actual probability

$$p_0 = \Pr \left[m \xleftarrow{\$} \mathsf{M} \, ; \; c = E(k, m) : Q(c) = m \right]$$

when all calls Q makes to an external random source are fed with a perfect source. Since the pirate decryption program is assumed to be fully or almost fully functional, p_0 must be significantly close to 1. It is classical to require from Q that $p_0 \geqslant 1/2$.

2. a probability estimator $\widehat{p_v}$ which, given $v \in [1, n]$, estimates the actual probability

$$p_v = \Pr \left[Q(c_v) = m_v \right]$$

where Q is run over a perfect random source again.

To estimate p_v for $v \in [0, n]$, Q is executed θ times (on fresh random tapes), where θ is an accuracy parameter. Then, one counts how many times, say ν, the returned output is as expected and $\widehat{p_v}$ is set to ν/θ. Finally, $\mathcal{T}$.trace implements a dichotomic search as shown on Fig. 4.

We state (see proof in Appendix B):

Theorem 3. *Assume* $\mathbf{C}_{\mathcal{E}}$ *is secure in the sense of both* PVH *and* PIH*. Then for any subset of traitors* $T \subseteq [1, n]$*,* $\mathcal{T}$.trace *correctly returns a traitor* $j \in T$ *with overwhelming probability after* $O(\log n)$ *executions of the pirate decryption program* Q*.*

This result validates the folklore intuition according to which cryptographic programs can be made efficiently traceable when properly obfuscated and assuming that slight alterations can be securely inserted in them. It also identifies clearly which sufficient security properties must be fulfilled by the white-box compiler to achieve traceability even when all users collude *i.e.*, in the context of total piracy.

Acknowledgements

This work has been financially supported by the French national FUI12 project MARSHAL+. The authors would like to thank Jean-Sébastien Coron and Louis Goubin for interesting discussions and suggestions.

References

1. Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2001.
2. Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer Berlin Heidelberg, 1998.
3. Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *SAC 2004*, pages 227–240. Springer-Verlag, 2005.
4. Dan Boneh, Amit Sahai, and Brent Waters. Fully Collusion Resistant Traitor Tracing with Short Ciphertexts and Private Keys. In *EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 573–592. Springer, 2006.
5. Dan Boneh and Ramarathnam Venkatesan. Breaking RSA May Not Be Equivalent to Factoring. In *EUROCRYPT 1998*, volume 1403 of *Lecture Notes in Computer Science*, pages 59–71. Springer, 1998.
6. Julien Bringer, Hervé Chabanne, and Emmanuelle Dottax. White Box Cryptography: Another Attempt. Cryptology ePrint Archive, Report 2006/468, 2006. http://eprint.iacr.org/.
7. Nishanth Chandran, Melissa Chase, and Vinod Vaikuntanathan. Functional Re-encryption and Collusion-Resistant Obfuscation. In *TCC 2012*, volume 7194 of *Lecture Notes in Computer Science*, pages 404–421. Springer, 2012.
8. Benny Chor, Amos Fiat, and Moni Naor. Tracing Traitors. In Y. Desmedt, editor, *CRYPTO 1994*, volume 839 of *Lecture Notes in Computer Science*, pages 257–270. Springer-Verlag, 1994.
9. Benny Chor, Amos Fiat, Moni Naor, and Benny Pinkas. Tracing Traitors. *IEEE Transactions on Information Theory*, 46(3):893–910, 2000.
10. Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. van Oorschot. A White-Box DES Implementation for DRM Applications. In Joan Feigenbaum, editor, *DRM 2002*, volume 2696 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002.
11. Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. van Oorschot. White-Box Cryptography and an AES Implementation. In *SAC 2002*, pages 250–270. Springer-Verlag, 2003.
12. Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of White Box DES Implementations. In Carlisle Adams, Ali Miri, and Michael Wiener, editors, *SAC 2007*, volume 4876, pages 278–295. Springer Berlin Heidelberg, 2007.
13. Dennis Hofheinz, John Malone-Lee, and Martijn Stam. Obfuscation for Cryptographic Purposes. *Journal of Cryptology*, 23(1):121–168, 2010.
14. Susan Hohenberger, Guy N. Rothblum, Abhi Shelat, and Vinod Vaikuntanathan. Securely Obfuscating Re-encryption. In *TCC 2007*, volume 4392 of *Lecture Notes in Computer Science*, pages 233–252. Springer, 2007.
15. Matthias Jacob, Dan Boneh, and Edward Felten. Attacking an Obfuscated Cipher by Injecting Faults. In Joan Feigenbaum, editor, *DRM 2002*, volume 2696, pages 16–31. Springer Berlin Heidelberg, 2002.
16. Marc Joye. On white-box cryptography. In Bart Preneel Atilla Elçi, S. Berna Ors, editor, *Security of Information and Networks*, pages 7–12. Trafford Publishing, 2008.
17. Marc Joye. Basics of Side-Channel Analysis. In *Cryptographic Engineering*, pages 365–380. Springer, 2009.
18. Mohamed Karroumi. Protecting white-box AES with dual ciphers. In *Proceedings of the 13th international conference on Information security and cryptology*, ICISC'10, pages 278–291. Springer-Verlag, 2010.
19. Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart Preneel. Two Attacks on a White-Box AES Implementation. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, Lecture Notes in Computer Science. Springer, 2013.
20. Hamilton E. Link and William D. Neumann. Clarifying obfuscation: improving the security of white-box DES. In *ITCC 2005*, volume 1, pages 679–684, 2005.

21. Wil Michiels, Paul Gorissen, and Henk D. L. Hollmann. Cryptanalysis of a Generic Class of White-Box Implementations. In *SAC 2008*, volume 5381 of *Lecture Notes in Computer Science*, pages 414–428. Springer, 2009.
22. Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao - Lai White-Box AES Implementation. In Lars R. Knudsen and Huapeng Wu, editors, *SAC 2012*, volume 7707, pages 34–49. Springer Berlin Heidelberg, 2013.
23. Yoni De Mulder, Brecht Wyseur, and Bart Preneel. Cryptanalysis of a Perturbated White-Box AES Implementation. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT 2010*, volume 6498, pages 292–310. Springer Berlin Heidelberg, 2010.
24. Dalit Naor, Moni Naor, and Jeffery Lotspiech. Revocation and Tracing Schemes for Stateless Receivers. In *CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2001.
25. Pascal Paillier and Damien Vergnaud. Discrete-Log-Based Signatures May Not Be Equivalent to Discrete Log. In Bimal Roy, editor, *ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2005.
26. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
27. Pankaj Rohatgi. Improved Techniques for Side-Channel Analysis. In *Cryptographic Engineering*, pages 381–406. Springer, 2009.
28. Amitabh Saxena, Brecht Wyseur, and Bart Preneel. Towards Security Notions for White-Box Cryptography. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio A. Ardagna, editors, *Information Security*, volume 5735 of *Lecture Notes in Computer Science*, pages 49–58. Springer Berlin Heidelberg, 2009.
29. Claude Elwood Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.
30. Brecht Wyseur. *White-Box Cryptography.* PhD thesis, Katholieke Universiteit Leuven, 2009.
31. Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In Carlisle Adams, Ali Miri, and Michael Wiener, editors, *SAC 2007*, volume 4876, pages 264–277. Springer Berlin Heidelberg, 2007.
32. Brecht Wyseur and Bart Preneel. Condensed white-box implementations. *Proceedings of the 26th Symposium on Information Theory in the Benelux*, pages 296–301, 2005.
33. Yaying Xiao and Xuejia Lai. A Secure Implementation of White-Box AES. In *CSA 2009*, pages 1–6, 2009.

A Security Notions for RSA Group Generators

We now recall the different security notions associated to a group generator $\mathbb{G}$ as above.

Definition 6 (Unbreakability – $\mathsf{UBK}[\mathbb{G}]$). *A probabilistic algorithm $\mathcal{A}$ is said to break the RSA group generator $\mathbb{G}$ when it succeeds in recovering ω from $\mathsf{desc}(\mathcal{G})$ for a random group $\mathcal{G}$ output by $\mathbb{G}$. The security game is as follows:*

1. *randomly select $(\mathsf{desc}(\mathcal{G}), \omega) \leftarrow \mathbb{G}(1^n)$,*
2. *run $\mathcal{A}$ over $\mathsf{desc}(\mathcal{G})$,*
3. *$\mathcal{A}$ returns some value $\hat{\omega}$,*
4. *$\mathcal{A}$ succeeds if $\hat{\omega} = \omega$.*

The success of $\mathcal{A}$ is defined as

$$\mathsf{Succ}_{\mathcal{A},\mathbb{G}}^{\mathsf{UBK}[\mathbb{G}]} = \Pr\left[\begin{array}{c}(\mathsf{desc}(\mathcal{G}), \omega) \leftarrow \mathbb{G}(1^n)\,; \\ \hat{\omega} \leftarrow \mathcal{A}(\mathsf{desc}(\mathcal{G}))\end{array} : \hat{\omega} = \omega\right] .$$

$\mathcal{A}$ is said to (τ, ε)-break $\mathbb{G}$ when it runs in time at most τ and $\mathsf{Succ}_{\mathcal{A},\mathbb{G}}^{\mathsf{UBK}[\mathbb{G}]} \geqslant \varepsilon$.

Definition 7 (Hardness of Extracting Orders – $\mathsf{ORD}[\mathbb{G}]$). *$\mathcal{A}$ is said to extract (multiplicative) orders on a random group $\mathcal{G}$ when given a random $x \in \mathcal{G}$, $\mathcal{A}$ returns the order $y = \mathsf{ord}(x)$ of x, or a multiple thereof. The security game is defined as follows:*

1. *randomly select* $(\mathsf{desc}(\mathcal{G}), \omega) \leftarrow \mathbb{G}(1^n)$,
2. *randomly select* $x \overset{\$}{\leftarrow} \mathcal{G}$,
3. *run* $\mathcal{A}$ *over* $(\mathsf{desc}(\mathcal{G}), x)$,
4. $\mathcal{A}$ *returns some value* $\hat{y}$,
5. $\mathcal{A}$ *succeeds if* $x^{\hat{y}} = 1_{\mathcal{G}}$.

The success of $\mathcal{A}$ is defined as

$$\mathsf{Succ}^{\mathsf{ORD}[\mathbb{G}]}_{\mathcal{A},\mathbb{G}} = \Pr\left[\begin{array}{c}(\mathsf{desc}(\mathcal{G}), \omega) \leftarrow \mathbb{G}(1^n)\,;\; x \overset{\$}{\leftarrow} \mathcal{G}\,; \\ \hat{y} \leftarrow \mathcal{A}(\mathsf{desc}(\mathcal{G}), x)\end{array} : x^{\hat{y}} = 1_{\mathcal{G}}\right].$$

$\mathcal{A}$ is said to (τ, ε)-break $\mathsf{ORD}[\mathbb{G}]$ when it runs in time at most τ and $\mathsf{Succ}^{\mathsf{ORD}[\mathbb{G}]}_{\mathcal{A},\mathbb{G}} \geqslant \varepsilon$.

Definition 8 (Hardness of Extracting Roots – $\mathsf{RSA}[\mathbb{G}]$). *$\mathcal{A}$ is said to extract roots on a random group $\mathcal{G}$ when given a random integer $e \in [0, \omega)$ such that $\gcd(e, \omega) = 1$, a random $x \in \mathcal{G}$, $\mathcal{A}$ returns the e-th root $y = x^{\frac{1}{e} \bmod \omega}$ of x. The game is as follows:*

1. *randomly select* $(\mathsf{desc}(\mathcal{G}), \omega) \leftarrow \mathbb{G}(1^n)$,
2. *randomly select* $e \overset{\$}{\leftarrow} \mathbb{Z}^*_\omega$,
3. *randomly select* $x \overset{\$}{\leftarrow} \mathcal{G}$,
4. *run* $\mathcal{A}$ *over* $(\mathsf{desc}(\mathcal{G}), e, x)$,
5. $\mathcal{A}$ *returns some value* $\hat{y}$,
6. $\mathcal{A}$ *succeeds if* $\hat{y} = x^{\frac{1}{e} \bmod \omega}$.

The success of $\mathcal{A}$ is defined as

$$\mathsf{Succ}^{\mathsf{RSA}[\mathbb{G}]}_{\mathcal{A},\mathbb{G}} = \Pr\left[\begin{array}{c}(\mathsf{desc}(\mathcal{G}), \omega) \leftarrow \mathbb{G}(1^n)\,;\; e \overset{\$}{\leftarrow} \mathbb{Z}^*_\omega\,; \\ x \overset{\$}{\leftarrow} \mathcal{G}\,;\; \hat{y} \leftarrow \mathcal{A}(\mathsf{desc}(\mathcal{G}), e, x)\end{array} : \hat{y} = x^{\frac{1}{e} \bmod \omega}\right].$$

$\mathcal{A}$ is said to (τ, ε)-break $\mathsf{RSA}[\mathbb{G}]$ when it runs in time at most τ and $\mathsf{Succ}^{\mathsf{RSA}[\mathbb{G}]}_{\mathcal{A},\mathbb{G}} \geqslant \varepsilon$.

Security Results for Known Constructions.

Claim ($\mathsf{UBK}[\mathbb{G}] \Leftarrow_{\mathcal{R}_1} \mathsf{ORD}[\mathbb{G}]$). There is a known reduction $\mathcal{R}_1$ polynomial in n, h that reduces $\mathsf{UBK}[\mathbb{G}]$ to extracting orders within the group. More precisely, if $\mathcal{R}_1$ is given an adversary that breaks the $\mathsf{ORD}[\mathbb{G}]$ game with success probability $\varepsilon_{\mathcal{A}}$, $\mathcal{R}_1$ can make use of $\mathcal{A}$ to recover ω with probability $\varepsilon_{\mathcal{R}_1} \geqslant \varepsilon_{\mathcal{A}}$ in time $\tau_{\mathcal{R}_1} \approx \tau_{\mathcal{A}}$.

Claim ($\mathsf{UBK}[\mathbb{G}] \not\Leftarrow_{\mathcal{R}_2} \mathsf{RSA}[\mathbb{G}]$). There is no known reduction $\mathcal{R}_2$ polynomial in n, h that reduces $\mathsf{UBK}[\mathbb{G}]$ to extracting roots within the group. One may therefore assume that it is hard to reduce $\mathsf{UBK}[\mathbb{G}]$ to $\mathsf{RSA}[\mathbb{G}]$. This is sometimes referred to as the gap factoring assumption.

B Proof of Theorem 3

To prove Theorem 3, it is enough (see [4]) to prove the following:

- **Property 0** : $p(0) \geqslant 1/2$.
- **Property 1** : $p(n)$ is negligibly close to 0.
- **Property 2** : $p(v)$ is negligibly close to $p(v+1)$ unless user v is a traitor.

These properties imply that there must be a substantial gap on the curve of $v \mapsto p(v)$ for some $v_{gap} \in [0, n-1]$ (*i.e.* $|p(v_{gap}) - p(v_{gap}+1)| \gg 0$) and that the user identity v_{gap} for which that gap occurs is a traitor with overwhelming probability. Our tracing procedure precisely searches for such a v_{gap} using dichotomy.

Property 0 is true by assumption on the pirate decryption program Q. We show that the other properties are also fulfilled under appropriate assumptions.

Proof (Property 1). We show that if PVH is (τ, ε)-hard, then $p(n) \leqslant \varepsilon$. We make use of the following adversarial formalization. An adversary $\mathcal{A}$ is assumed to corrupt all users $i \in [1, n]$ by having access to the user decryption programs $P_1, \ldots, P_n$. $\mathcal{A}$ then outputs a rogue decryption program Q. $\mathcal{A}$ wins the game if $Q(c_1) = m_1$, that is, if Q succeeds in decrypting c_1 although none of the P_i's could. We see that $\mathcal{A}$'s success probability is precisely $\varepsilon_{\mathcal{A}} = \Pr[Q(c_1) = m_1] = p(n)$. The total time $\tau_{\mathcal{A}}$ taken by the adversary is defined to be the running time of $\mathcal{A}$ to issue Q added to the execution time of Q itself.

We now build a reduction $\mathcal{R}$ which, given a pirate adversary $\mathcal{A}$ as above that runs in time $\tau_{\mathcal{A}}$ and outputs a rogue decryption program Q such that $p(n) = \varepsilon_{\mathcal{A}}$, wins the PVH game with probability $\varepsilon_{\mathcal{R}} = \varepsilon_{\mathcal{A}}$ in time $\tau_{\mathcal{R}} \approx \tau_{\mathcal{A}}$. $\mathcal{R}$ is given a PVH instance (c, P_c) where c is a random ciphertext and $P_c = \mathbf{C}_{\mathcal{E}}(k, r; \langle c \rangle)$. $\mathcal{R}$ wants to return the correct output $m = D_k(c)$ of the perturbated entry c. To use the pirate adversary against the scheme, $\mathcal{R}$ has to produce perturbated programs $P_1, \ldots, P_n$ in accordance with the distribution as per our construction. To do this, $\mathcal{R}$ sets $c_1 := c$, $P_1 := P_c$, picks random ciphertexts $(c_2, \ldots, c_n)$ and for $i = 2$ to n, queries the recompiling oracle to get

$$P_i = \mathcal{O}\left(\langle c, c_2, c_3, \ldots, c_i \rangle\right) = \mathbf{C}_{\mathcal{E}}(k, r_i; \langle c, c_2, c_3, \ldots, c_i \rangle) = [D^{r_i}_{k, \langle c, c_2, \ldots, c_i \rangle}]$$

for some (unknown) $r_i \xleftarrow{\$} \mathsf{R}$. It is easily seen that $P_1, \ldots, P_n$ perfectly comply with the specification of the decryption programs assigned to end users in our construction. Now $\mathcal{R}$ runs $\mathcal{A}$ on $(P_1, \ldots, P_n)$ and obtains a rogue decryption program Q. $\mathcal{R}$ runs Q once to set $m := Q(c)$ and returns m to its challenger. Now by assumption,

$$\varepsilon_{\mathcal{A}} = p(n) = \Pr[Q(c_1) = D_k(c_1)] = \Pr[Q(c) = D_k(c)] \ .$$

Finally, $\varepsilon_{\mathcal{R}} = \Pr[Q(c) = D_k(c)] = \varepsilon_{\mathcal{A}}$ and $\tau_{\mathcal{R}} = \tau_{\mathcal{A}} + O(n)$ as claimed. □

Proof (Property 2). We show that if the white-box compiler is (τ, ε)-secure in the sense of PIH for some $v \in [1, n-1]$, then $|p(v) - p(v+1)| \leqslant \varepsilon$. We prove this by defining an adversary $\mathcal{A}$ that outputs a rogue decryption program Q such that $|p(v) - p(v+1)| \geqslant \varepsilon_{\mathcal{A}}$. $\mathcal{A}$ is assumed to corrupt all users but one *i.e.* is only given programs $P_1, \dots, P_{v-1}, P_{v+1}, \dots, P_n$, thereby excluding P_v. The total time $\tau_{\mathcal{A}}$ taken by the adversary is defined to be the running time of $\mathcal{A}$ to return Q added to the execution time of Q itself.

We build a reduction algorithm $\mathcal{R}$ which, given an adversary $\mathcal{A}$ as above, wins the PIH game with advantage $\varepsilon_{\mathcal{R}} \geqslant \varepsilon_{\mathcal{A}}$ in time $\tau_{\mathcal{R}} \approx \tau_{\mathcal{A}}$. $\mathcal{R}$ is given a PIH instance

$$(P_1, \dots, P_{v-1}, P_{v+1}, \dots, P_n, m_{v+b}, c_{v+b})$$

where

$$\begin{aligned}
P_1 &= \mathbf{C}_{\mathcal{E}}(k, \mathsf{R}; \langle c_1 \rangle) \\
P_2 &= \mathbf{C}_{\mathcal{E}}(k, \mathsf{R}; \langle c_1, c_2 \rangle) \\
&\vdots \\
P_{v-1} &= \mathbf{C}_{\mathcal{E}}(k, \mathsf{R}; \langle c_1, c_2, \dots, c_{v-1} \rangle) \\
P_{v+1} &= \mathbf{C}_{\mathcal{E}}(k, \mathsf{R}; \langle c_1, c_2, \dots, c_{v-1}, c_v, c_{v+1} \rangle) \\
P_{v+2} &= \mathbf{C}_{\mathcal{E}}(k, \mathsf{R}; \langle c_1, c_2, \dots, c_{v-1}, c_v, c_{v+1}, c_{v+2} \rangle) \\
&\vdots \\
P_n &= \mathbf{C}_{\mathcal{E}}(k, \mathsf{R}; \langle c_1, c_2, \dots, c_n \rangle) \ .
\end{aligned}$$

The reduction $\mathcal{R}$ shall eventually output a guess $\hat{b}$ for b. To produce the end-user decryption programs expected by $\mathcal{A}$, $\mathcal{R}$ just forward the $n-1$ programs $P_1, \dots, P_{v-1}, P_{v+1}, \dots, P_n$ and obtains a rogue decryption program $Q \leftarrow \mathcal{A}(P_1, \dots, P_{v-1}, P_{v+1}, \dots, P_n)$. Then $\mathcal{R}$ runs Q on c_{v+b} with a fresh random tape. If $Q(c_{v+b}) = m_{v+b}$ then $\hat{b} := 1$ is returned otherwise $\mathcal{R}$ returns $\hat{b} := 0$. This completes the description of our reduction algorithm $\mathcal{R}$. We see that

$$\begin{aligned}
\varepsilon_{\mathcal{R}} &= \left| \Pr[\mathcal{R} = 1 \mid b = 1] - \Pr[\mathcal{R} = 1 \mid b = 0] \right| \\
&= \left| \Pr[Q(c_{v+b}) = m_{v+b} \mid b = 1] - \Pr[Q(c_{v+b}) = m_{v+b} \mid b = 0] \right| \\
&= \left| \Pr[Q(c_{v+1}) = D_k(c_{v+1})] - \Pr[Q(c_v) = D_k(c_v)] \right| \\
&= \left| p(v+1) - p(v) \right| \geqslant \varepsilon_{\mathcal{A}} \ .
\end{aligned}$$

Concluding, this shows that $\varepsilon_{\mathcal{R}} \geqslant \varepsilon_{\mathcal{A}}$ with $\tau_{\mathcal{R}} \approx \tau_{\mathcal{A}}$ as claimed. □