White-box Cryptography with Device Binding from Token-based Obfuscation and more

Shashank Agrawal[1], Estuardo Alpírez Bock[2]*, Yilei Chen[3], and Gaven Watson[4]

[1] Western Digital Research `shashank.agraval@gmail.com`
[2] Aalto University `estuardo.alpirezbock@aalto.fi`
[3] Tsinghua University `chenyilei.ra@gmail.com`
[4] Visa Research `gawatson@visa.com`

Abstract. White-box cryptography has been proposed as a software countermeasure technique for applications where limited or no hardware-based security is available. In recent years it has been crucial for enabling the security of mobile payment applications. In this paper we continue a recent line of research on device binding for white-box cryptography [2,3]. Device binding ensures that a white-box program is only executable on one specific device and is unusable elsewhere. Building on this, we ask the following question: is it possible to design a *global* white-box program which is compiled once, but can be securely shared with multiple users and bound to each of their devices? Acknowledging this question, we provide two new types of provably-secure constructions for white-box programs.

First, we consider the use of Token-Based Obfuscation (TBO) [28] and show that TBO can provide us a direct way to construct white-box programs with device-binding, as long as we can securely share a token generation key between the compiling entity and the device running the white-box program. This new feasibility result provides more general and efficient results than previously presented for white-box cryptography and demonstrates a new application of TBO not previously considered.

We then consider a stronger family of global white-boxes, where secrets don't need to be shared between users and providers. We show how to extend approaches used in practice based on message recoverable signatures and validate our proposed approach, by providing a construction based on puncturable PRFs and indistinguishability obfuscation.

Keywords: White-box cryptography · Obfuscation · Device-binding · Mobile payments

1 Introduction

We use cryptography to address a multitude of use cases and as a result it is deployed in all types of environments, from ultra-secure government facilities to our mobile phones. For the most sensitive environments, we can use special hardware to secure cryptographic implementations. However, in many settings where such specialized hardware is not available, we must ensure security against attackers which may have full control of the execution environment. Here, white-box cryptography aims to implement cryptographic algorithms that remain secure in the presence of such adversaries. White-box cryptography was introduced in 2002 by Chow, Eisen, Johnson, and van Oorschot [19,20].

White-box cryptography finds its main use cases in Digital-Rights Management (DRM) and mobile-payment applications [45,23] (see Section A for an extended discussion on the role of white-box cryptography in mobile payments). In both cases, standard symmetric encryption ciphers are implemented in a white-box fashion. However, it is not publicly known how

*Research conducted while interning at VISA Research.

exactly the ciphers are implemented and which level of security they actually achieve. The scientific community has proposed white-box designs for DES [20,37] and AES [19,15,48,33,5], but all of these approaches have been subject to key-extraction attacks (cf. [31,29,47] and [10,41,40,35,22]).

It remains an open question, whether one can construct a white-box implementation of DES or AES which remains secure against key-extraction attacks. In practice, white-box designs remain robust for a certain period of time, and key extraction is mitigated by periodically rotating the embedded keys and updating the white-box obfuscation. However, we should note that in the application scenarios discussed above, white-box programs also implement countermeasures against so called *code-lifting* attacks.

1.1 Code-lifting Attacks and Device Binding

White-box programs may be susceptible to code-lifting attacks, where, an adversary simply copies the execution code of the program with its embedded secret key [46]. The adversary can then run the code for his own purposes on any device or environment of its choice, without needing to perform a key-extraction or reverse-engineering attack. In real-life applications, such attacks take place when a user re-distributes his DRM decryption software, or when an invasive adversary (in the form of malware) copies the payment application and credentials of a user.

One approach to mitigate code-lifting attacks is to implement *device binding* operations [6,44,23]. That is, the white-box programs are configured such that they can only be executed on one specific device. In practice, device binding plays an important role in combination with white-box cryptography for protecting mobile payment applications, as recommended by EMVCo [23]. Device binding can be implemented by having the white-box program verify a specific signature provided by some trusted component or party. Alternatively, one can design a white-box program which does not compute directly on raw inputs, but rather on *encoded* inputs. The corresponding encoding algorithm can be extracted from the secure hardware of a device. This encoding forces us thus to run our white-box program in combination with that device.

It may seem counterintuitive to combine white-box cryptography with dependency on some trusted hardware. However, in many environments the availability of specific secure hardware and what algorithms they support may be somewhat limited. For example, payment processes rely heavily on 3DES which has only been supported in Android Keystore recently (since v9) and is not supported by many secure elements. This is clearly an example where a device may have trusted hardware available but it cannot provide the required cryptographic functionality. One can view our approach of using both a trusted component and a white-box obfuscation scheme as a means to bootstrap the limited functionality of the trusted component to facilitate the secure computation of a complex functionality. In addition it facilitates added *crypto-agility*, once secure hardware is deployed it can be relatively hard to update with newer algorithms. By bootstapping with a white-box obfuscation scheme we can enable a secure hardware deployment to support new and needed cryptographic schemes.

Security definitions for white-box cryptography with device binding have been presented in recent years. Alpirez Bock et al. [2] define security for white-box encryption programs and, in further work [3], the security of a white-box key derivation function and a payment application. The definitions in these works capture the property that the corresponding programs only execute correctly in combination with one specific device and are otherwise useless, while preserving their corresponding security properties. In the second work [3], the authors provide feasibility results for white-boxed payment applications. Their constructions are based on puncturable pseudorandom functions (PPRF) and indistinguishability obfuscation (iO) [7]. In a way, the result gives a general feasibility result for the white-box payment applications, since the construction can be augmented by any secure symmetric-encryption scheme. One may wonder whether white-box cryptography for arbitrary ciphers can only be achieved via strong obfuscation assumptions, such as indistinguishability obfuscation.

Other methods for mitigating code-lifting. In white-box related literature, the property of *incompressibility* has been widely studied as a means to mitigate code-lifting attacks. Incompressibility aims at designing a program of large size, which remains functional only in its complete form. If the program is compressed or fragments of the program are removed, the program should lose its functionality. Thus, incompressible programs (of very large size) should be difficult to share over the network. A line of definitions, feasibility results and concrete constructions have been presented for incompressibility [21,13,14,11,24,18,1,34]. However to the best of our knowledge, incompressibility is not really implemented in practice. Namely as discussed in [2], the large size of the programs also makes their *legal* distribution difficult. Moreover, the idea of having software programs of very large size stands in contrast with the folklore goal of designing small sized applications for mobile devices. We note also that incompressible ciphers are rather new, white-box friendly ciphers, while commercial applications usually make use of more standardized ciphers such as AES. For this reason as explained above, commercial applications aim to implement an obfuscated version of a standardized cipher and bind it to one specific device. In this paper we focus on device-binding as opposed to incompressibility since we wish to study white-box cryptography from a perspective of how it is implemented in real life.

1.2 Our Contributions

In this paper, we continue the study of white-box cryptography with device binding. We start by observing that the previous constructions [3,2] necessitate that a unique white-box program is created for each device. In this line, we ask the following question: is it possible to create a single *global white-box* which can be distributed to multiple users and securely bound to each user's device? With such a construction, we aim to take further steps towards more practical white-box approaches. Namely with a global white-box, our compiler needs to generate only one white-box application and place it directly in the cloud available for download (avoiding multiple expensive obfuscation operations). Such a setting stands in-line with traditional app stores and does not necessitate any additional side-loading when downloading the program.

To enable our work we define a new formal framework for global white-boxes and examine two different strengths of global white-box constructions. In one case, we assume that a secret is shared with the corresponding devices and this secret will be used for the binding. In the second case, we relax this assumption and consider thus a stronger class of global white-boxes. Further intuition on these is given in the next section of the introduction.

At a high-level we consider a global white-box compiler which will obfuscate some function F. Generally, the function F corresponds to the description of a program we wish to combine with a white-box decryption program. Our schemes must satisfy two important properties: ensure the confidentiality of inputs to the white-box (e.g. an encrypted stream in the case of DRM or a bank provided key in the case of payments) and ensure that the white-box can only be evaluated on inputs which have been *"bound"* to a device's trusted component. To capture these properties we introduce corresponding notions of security for both classes of global white-box programs. On this line, we give examples of how each class of global white-box can be constructed by providing corresponding provably secure constructions. For the first class, our constructions are based on Token-based obfuscation. This result provides us a new flavour of white-box constructions, showing that strong obfuscation assumptions, such as indistinguishability obfuscation, are *not* necessary for white-box obfuscation of arbitrary ciphers. At the same time, this construction allows us to demonstrate a new use case for Token-based obfuscation which had not been previously considered. Our second flavour of constructions, which meet the stronger version of global white-boxes, are inspired by message-recoverable signatures and augmented by puncturable pseudorandom functions and indistinguishability obfuscation. This type of constructions suggest thus a general approach for device-binding *if* full-fledged obfuscation schemes were secure and practical. In the rest of the introduction we dive deeper into the concepts of global white-boxes, their different flavors and how we construct them.

1.3 Global White-boxes

Ideally, we would like a software provider (or server) to compile a single white-box program and upload it to the cloud. Any user who wants to acquire the program can download it to their device. However upon download, the white-box program should not (yet) be functional. Instead, the program should first be properly enrolled and bound to a user's device. This way we ensure that the program is not subject to code-lifting attacks. Below we introduce two possible ways of constructing global white-boxes with their corresponding enrollment processes.

First we consider a white-box which is compiled to only work on inputs which are encoded using some secret key material. This key material is generated by the server such that the corresponding *decoding* key can be embedded when creating the white-box. A user can only run this white-box if he obtains the corresponding encoding key from the server. By securely storing this key in the hardware of a user's device, we ensure that use of the white-box is bound to that device. Below we show an example in pseudocode of how such a white-box program would work in collaboration with the secure hardware, denoted here as HW. The secure hardware takes as input a ciphertext that we wish to later decrypt with our white-box program. HW encodes the ciphertext with the key it obtained from the server, e_s, and obtains $\tilde{c}$. The white-box program has a decoding and a decryption key embedded (d_s and k) and it takes as input the encoded ciphertext $\tilde{c}$. The white-box first decodes a ciphertext and then decrypts it.

```
$\mathsf{HW}(c)$
    $\tilde{c} \leftarrow \mathsf{Encode}(e_s, c)$
    return $\tilde{c}$

$\mathsf{WB}(\tilde{c})$
    $c \leftarrow \mathsf{Decode}(d_s, \tilde{c})$
    $m \leftarrow \mathsf{Dec}(k, c)$
    return $m$
```

Note that the white-box program needs to be obfuscated such that an adversary is unable to separate the decoding and the decryption operations. This makes it impossible for an adversary to run the white-box without access to a hardware device with the corresponding encoding key. The scenario described above achieves our initial goal of a *global white-box* as long as the encoding key can be securely shared between the server and the devices.

A stronger approach is if each user makes use of their own (unique) key for encoding inputs to the white-box. Here we additionally ensure that even if a device's trusted hardware is compromised this only affects the binding of that device and no others, thus providing a strongly level of security. To implement this approach we need a white-box which takes two encoded inputs. One input is the ciphertext encoded via the secret key of the user's hardware. The other input is the corresponding decoding key of the user's hardware, but encoded via the server's secret key. The white-box should first recover the decoding key of the user and then use the recovered key to decode the encoded ciphertext, as shown in the pseudocode below. Here d_h is the decoding key of the user's hardware. The server encodes d_h via the server's secret key e_s. The hardware encodes a ciphertext via the hardware's secret key and returns $\hat{c}$.

```
$\mathsf{Server}(d_h)$
$\tilde{d_h} \leftarrow \mathsf{Encode}(e_s, d_h)$
return $\tilde{d_h}$
```

```
$\mathsf{HW}(c)$
$\tilde{c} \leftarrow \mathsf{Encode}(e_h, c)$
return $\hat{c}$
```

```
$\mathsf{WB}(\tilde{d_h}, \tilde{c})$
$d_h \leftarrow \mathsf{Decode}(d_s, \tilde{d_h})$
$c \leftarrow \mathsf{Decode}(d_h, \tilde{c})$
$m \leftarrow \mathsf{Dec}(k, c)$
return $m$
```

Here as before, the white-box program should be obfuscated such that an adversary is unable to separate the decode and decryption algorithms. Note that if an adversary copies the white-box, a ciphertext *and* the encoded key $\tilde{d_h}$ (or even gains knowledge of the decoding keys d_s and d_h), the adversary is still unable to correctly perform a decryption, unless he can access the corresponding hardware in order to correctly encode the ciphertext.

We refer to the scenario described above as a *strong global white-box*. The idea is that the white-box program will only output correct values if both its inputs are provided consistently, i.e. if the encoded key $\tilde{d_h}$ is used in combination with inputs encoded via the device holding d_h. In what follows we explain how we can construct white-boxes used in both, the simple and in the stronger global white-box setting.

1.4 Token-based Obfuscation for Global White-boxes

An obfuscator is a compiler that takes a plaintext circuit C as input and outputs a circuit $\widetilde{C}$ that is functionally equivalent to C but is "unintelligible" [30,7]. General-purpose code obfuscation is an amazingly powerful technique, making it possible to hide arbitrary secrets in any software. Tremendous progress has been made in the area of obfuscation in the last two decades. Strong security definitions of obfuscation were formalized in the work of Hada [30] and Barak et al. [7] in early 2000s. The first candidate construction of iO, proposed by Garg et al. [25], opened up a new direction of research that transformed our thinking about what can and cannot be done in cryptography. The goal of white-box cryptography can be viewed as providing practical obfuscation of special programs, namely symmetric ciphers.

However, constructing a full-fledged obfuscator that is both efficient and secure based on the hardness of established mathematical problems is quite difficult. Since the proposal of the first full-fledged obfuscation candidate in 2013 [25], there were several attacks [17,16] and alternative proposals [12,4,36]. So far, only one of them [32] is based on relatively well-established assumptions.

On the other hand, if we would like to produce obfuscated programs that are only executable on inputs for which some special token has been issued, then such a restricted type of obfuscation is easier to construct, and was indeed formalized as Token-Based Obfuscation

(TBO) by Goldwasser et al. [28]. Readers who are familiar with garbled circuits can think of token-based obfuscation as reusable garbled circuits, where the obfuscated circuit is the reusable garbled circuit and the tokens are the garbled input. Here, the obfuscated circuit leaks no information other than the size of the circuit and the outputs obtained from evaluating the reusable garbled circuit on the specific garbled inputs. TBO can be constructed based on the hardness of learning-with-errors problems [42], as opposed to full-fledged obfuscation for which the security of none of the candidate constructions is well-established.

A convenient restriction. As mentioned above, the security of TBO is achieved under the restriction that the obfuscated circuit (or program) can only be executed for specific inputs: the inputs for which a user obtains a token. A token-based obfuscation scheme is therefore defined in combination with a token-generation algorithm, where the token generation key is created together with the obfuscated program. We recall now that for achieving device-binding, we want to generate a white-box program which can only be correctly executed in combination with a specific trusted component. Here, token-based obfuscation directly gives us the functionality and security we desire for our complete white-box scheme when we place our token input generator directly on the trusted component.

For context, consider the following simplified example, where we already introduce part of our notation. Let F be a pseudorandom permutation with an embedded secret key, such that $F(c) = x$. Furthermore, assume that we want to use this pseudorandom permutation as a decryption function. That is, the input c corresponds to a ciphertext which was generated for some message x via the inverse of F. We now obfuscate F via token-based obfuscation: $(O[F], \mathsf{MSK}) \leftarrow_{\$} \mathsf{TBO.Obf}(F)$. We obtain thus a program $O[F]$ which alone reveals nothing about F. $O[F]$ can be used for recovering x only when we can obtain a valid token for the corresponding c value: $\tilde{c} \leftarrow_{\$} \mathsf{TBO.InpGen}(\mathsf{MSK}, c)$. Upon receiving $\tilde{c}$, a user can recover x via $O[F](\tilde{c})$. Thus, if we implement the token-generation algorithm within the trusted component of a device, the obfuscated program becomes useful only if it has access to that device, achieving thus the property of device binding.

In this paper we formalize the construction described above and show how we can use TBO to construct global white-boxes. This leads us to new interesting feasibility results for white-box cryptography which are based on LWE [42] and we thus show that *strong notions of obfuscation such as iO are not always necessary* for building white-box applications instantiated with any encryption scheme, such as AES.

1.5 Message-recoverable Signatures for Strong Global White-boxes

Message-recoverable signatures (MRS) were introduced by Bellare and Rogaway [8]. Unlike traditional signature schemes, whose signing algorithm generates a signature for a particular message, an MRS signing algorithm *embeds* the signature within the message. Additionally, an MRS scheme consists of a *recover* algorithm which verifies the signature and returns the original message. Security of MRS holds as long as an adversary is unable to forge a valid signed message. The original benefit of using MRS as opposed to a traditional signature is that it reduces the amount of data that must be sent between the signing and the verifying entity. In [8], the authors show how signature schemes such as RSA can be used to construct MRS schemes.

As of today, MRS-inspired approaches are used in practice for implementing device binding for white-box programs running on mobile phones with trusted components [38]. In such approaches, the trusted component on a user's phone generates an RSA key pair and securely stores the secret key. The public key is shared with the entity compiling the white-box program and the white-box is compiled such that it has the public key embedded in it, together with a symmetric decryption key. Whenever we want to decrypt a ciphertext, we first sign it via the secret key stored in the trusted component. We then give this signed ciphertext as input to the white-box, which first uses the public key to recover the ciphertext before the final decryption with the symmetric key. Note that given such a white-box program, it should be difficult for an adversary to separate the message recovery from the decryption algorithm. Thus, the white-box program (with the embedded public key) can only be used in the presence of the trusted component which generates the signed ciphertexts.

By extending this approach, we can build a compiler which creates a single white-box program that may be used by all legitimate users. This approach would use two layers of message-recoverable signatures and the white-box program would have the public key of the server and a symmetric decryption key embedded. The first layer will use the embedded public key to recover the unique public key of the user. The second layer will use that user's public key to decode the ciphertext. Thus ensuring the stricter requirement of per-device binding of a strong global white-box. The use of MRS makes the enrollment process described in the previous section easier, since no secrets need to be shared between the server and the user.

MRS vs Traditional signatures. We note that device binding for white-box programs could also be implemented using a traditional signature scheme. In this case instead of encoding the message we wish to compute via an MRS scheme, the trusted component generates a separate signature for that specific message. The white-box program would verify the signature and only proceed if the signature was valid. In this paper we choose to focus on the use of MRS in order to validate and extend approaches used in practice [38]. Why such MRS-inspired approaches are used in practice instead of traditional signature schemes is not completely clear. In early white-box related works, it was proposed to make use of *external encodings* for protecting white-box AES designs against code-lifting attacks [19,39]. Here, external encodings refer to encodings provided by an external source (e.g. a secure hardware), and the white-box program is designed such that it computes correctly only on values encoded via that external source. The approach of using MRS correlates with this proposal since we are using the secret signing key as our encoding function and the public key embedded in the white-box as a decoding function. Note that for such a design, the recover algorithm can be simplified and does not really need to check for the validity of the signature. Instead, it can directly perform the decoding using the embedded public key. If the input was not encoded or encoded using the wrong secret key, then the output of the white-box will be anyway faulty (see Section 1.3). It is an interesting question whether it is easier to obfuscate (in practice) a program which checks for the validity of a signature (and then decrypts) or a program which decrypts directly on encoded inputs.

2 Preliminaries and Notation

$a \leftarrow b$ denotes the assignment of a value b to a variable a. We denote by $a \leftarrow A(b)$ the execution of a deterministic algorithm A on input b to produce an output a. $a \leftarrow_{\$} A(b)$ denotes the execution of a probabilistic algorithm A. We use square brackets to denote a fixed value hard-coded into an algorithm, e.g. $A[k]$ denotes that the value k is hard-coded in the algorithm

A. $a \| b$ denotes the concatenation of two values a and b, while $|a|$ denotes the length of a value a. $q \leftarrow\!\!\$\, Q$ denotes the process of randomly sampling an element q from a set (or distribution) Q.

By 1^λ we denote (the unary representation of) the security parameter, which all algorithms receive as input. We write it explicitly for the algorithms which only take the security parameter as input and leave it implicit for the rest. The subscript to an adversary $\mathcal{A}$ denotes the class of oracles the adversary gets access to in our security definitions. With $\approx_c$ we denote computationally indistinguishability.

2.1 Theoretical Background on Obfuscation

Let us recall the definitions of strong VBB obfuscation and indistinguishability obfuscation.

Definition 1 (Obfuscation [30,7]). *A probabilistic algorithm O is an obfuscator for a class of circuit $\mathcal{C}$ if the following conditions hold:*

- *(Preservation of the function) For all inputs x, $\Pr[C(x) = O(C(x))] > 1 - \mathsf{negl}(\lambda)$.*
- *(Polynomially slowdown) There is a polynomial p s.t. $|O(C)| < p(|C|)$.*
- *(Strong virtual black-box obfuscation) For any PPT adversary $\mathcal{A}$, there is a PPT simulator Sim s.t. for all C, $\left\{\mathcal{A}(1^\lambda, O(C))\right\} \approx_c \left\{\mathsf{Sim}^{C(\cdot)}(1^\lambda, |C|)\right\}$.*
- *(Indistinguishability obfuscation) For functionally equivalent circuits C_0, C_1, $O(C_0) \approx_c O(C_1)$.*

Let us recall the definition of token-based obfuscation [28].

Definition 2 (Token-based obfuscation [28]). *A token-based obfuscation scheme for a class of functions $\mathcal{F}_n = \{F : \{0,1\}^n \to \{0,1\}\}$ is a tuple of PPT algorithms ($\mathsf{TBO.Obf}$, $\mathsf{TBO.InpGen}$) such that:*

- *$\mathsf{TBO.Obf}(1^\lambda, F)$ takes as input the security parameter 1^λ and a function F, and outputs the master secret key MSK and an obfuscated program $O[F]$.*
- *$\mathsf{TBO.InpGen}(\mathsf{MSK}, m)$ takes MSK and a message $m \in \{0,1\}^n$, and outputs a token $\tilde{m}$.*

We require that for every message $m \in \{0,1\}^n$ and function $F \in \mathcal{F}_n$, we have correctness*:*

$$\Pr[O[F](\mathsf{TBO.InpGen}(\mathsf{MSK}, m)) = F(m)] > 1 - \mathsf{negl}(\lambda),$$

where the probability is taken over the randomness of the algorithms $\mathsf{TBO.Obf}$ and $\mathsf{TBO.InpGen}$.

Efficiency. *The running time of $\mathsf{TBO.InpGen}$ is independent of the size of F.*

Security. *We say that a token-based obfuscation scheme is secure if for all PPT stateful algorithms $\mathcal{A}$, there is a PPT stateful algorithm Sim such that:*

$$\left\{\textit{Experiment } \mathit{REAL}_{\mathcal{A}}(1^\lambda)\right\}_{\lambda \in \mathbb{N}} \approx_c \left\{\textit{Experiment } \mathit{IDEAL}_{\mathcal{A},\mathsf{Sim}}(1^\lambda)\right\}_{\lambda \in \mathbb{N}},$$

where the real and ideal experiments are defined as follows:

```
Experiment $REAL_{\mathcal{A}}(1^\lambda)$
$(F, \mathsf{state}_A) \leftarrow\$ \mathcal{A}(1^\lambda)$;
$(\mathsf{MSK}, O[F]) \leftarrow\$ \mathsf{TBO.Obf}(1^\lambda, F)$;
$b \leftarrow\$ \mathcal{A}^{\mathsf{TBO.InpGen}(\mathsf{MSK}, m)}(O[F], \mathsf{state}_A)$;
Output $b$
```

```
Experiment $IDEAL_{\mathcal{A},\mathsf{Sim}}(1^\lambda)$
$(F, \mathsf{state}_A) \leftarrow\$ \mathcal{A}(1^\lambda)$;
$(\mathsf{state}_S, O_S) \leftarrow\$ \mathsf{Sim}_P(1^\lambda, 1^{|F|})$;
$b \leftarrow\$ \mathcal{A}^{\mathsf{Sim}_I(\mathsf{state}_S, 1^{|m|}, F(m))}(O_S, \mathsf{state}_A)$;
Output $b$
```

In the experiments, $\mathcal{A}$ can ask for a single obfuscated program and polynomially many input queries, denoted as m afterwards. Once $\mathcal{A}$ picks a function $F \in \mathcal{F}_\lambda$, in the real experiment, $\mathcal{A}$ obtains the obfuscated code $O[F]$; in the ideal experiment, it obtains a program generated by Sim_P, where Sim_P is given only the size of F. For the input-token queries made by $\mathcal{A}$ after the function query, in the real experiment, $\mathcal{A}$ obtains the token of m. In the ideal experiment, $\mathcal{A}$ obtains a value generated by Sim_I, where Sim_I is given only $1^{|m|}$ and $F(m)$. The output of the experiment is the final output bit of $\mathcal{A}$.

3 Global White-boxes with Device Binding

Device binding ensures that a white-box program can only be executed on one specific device and is useless without access to that device. As mentioned earlier, device binding is a countermeasure technique for preventing code-lifting attacks. Previous notions for white-box device binding [3,2] relate to schemes which require a specific white-box application for each individual user. That is, every time a user enrolls to use a service, a unique white-box is compiled and sent to that user. Here we introduce our new definitions which consider a more general approach for distributing white-boxes, where a single global white-box program is created for all users.

3.1 Definitions

Our Setting. We consider a setting where a *provider* creates an obfuscated program (for some functionality F) which will be distributed to multiple devices. We assume devices each have a trusted component which is used for the purposes of device binding. In practice, this trusted component has limited functionality and supports only standard cryptographic primitives. We assume trusted components have been securely initialized based on some enrollment process with the provider.

The provider *encrypts* inputs for the white-box program which will be provided to the device. Before executing the white-box program, the device submits these encrypted inputs to its trusted component to bind them to the device (by performing an additional *encoding* step). An obfuscated white-box program execution proceeds by verifying that the input is properly bound to the device and then executing the functionality F.

Formally achieving these goals. We now define the formal syntax for our global white-box scheme, which we refer to as a GW-scheme. In the definition below, we consider a program (compiler) Comp which based on a function F, generates an obfuscated program and the necessary key material for the scheme. The obfuscated program WB should work on encoded data and be functionally equivalent to F. This is further depicted in Figure 1.

Definition 3. *A Global White-box GW-Scheme consists of the following algorithms:*

- *A randomized compiler Comp which takes as input a function F with syntax $z \leftarrow F(x, y)$. The compiler returns a program WB and randomly generated keys ek and bk, i.e. $(\mathsf{ek}, \mathsf{bk}, \mathsf{WB}) \leftarrow_\$ \mathsf{Comp}(F)$.*
- *A probabilistic algorithm $\mathsf{Encrypt}$ which takes as input a secret (encryption) key ek, a value x, and an identifier ID, and outputs an encrypted message c, i.e. $c \leftarrow_\$ \mathsf{Encrypt}(\mathsf{ek}, x, \mathsf{ID})$.*
- *A probabilistic algorithm Encode, which takes as input a secret (binding) key bk, a value c, and an additional value y, and returns an encoded value $\tilde{c}$, i.e. $\tilde{c} \leftarrow_\$ \mathsf{Encode}(\mathsf{bk}, c, y)$.*

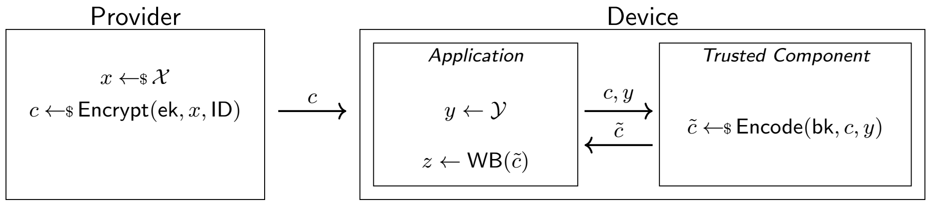
Provider
$x \leftarrow\!\$\, \mathcal{X}$
$c \leftarrow\!\$\, \mathsf{Encrypt}(\mathsf{ek}, x, \mathsf{ID})$
c
Device
Application
$y \leftarrow \mathcal{Y}$
$z \leftarrow \mathsf{WB}(\tilde{c})$
c, y
$\tilde{c}$
Trusted Component
$\tilde{c} \leftarrow\!\$\, \mathsf{Encode}(\mathsf{bk}, c, y)$

Fig. 1: *Global White-box (*GW*) Scheme.* For initial setup the provider runs the compiler program to generate the whitebox and some key material, $(\mathsf{ek}, \mathsf{bk}, \mathsf{WB}) \leftarrow\!\$\, \mathsf{Comp}(F)$. WB is distributed to all devices via a central appstore. The binding key bk is distributed to all devices through some PKI-based enrollment phase.

Correctness states that for every function $F : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$, for all $x \in \mathcal{X}$, all $y \in \mathcal{Y}$, we have

$$\Pr[\mathsf{WB}(\mathsf{Encode}(\mathsf{bk}, \mathsf{Encrypt}(\mathsf{ek}, x, \mathsf{ID}), y)) = F(x, y)] = 1,$$

where the probability is over the randomness of all algorithms and the corresponding secret keys bk, ek*.*

For context, note that the WB program has the following syntax: $\mathsf{WB}(\tilde{c}) = F(x, y)$. Note that $\tilde{c}$ is generated based on an encryption of a value x *and* a value y. WB recovers the values x and y and then computes $F(x, y)$. Note that the values x and y should not be revealed during computation. Below we define the security notions for our scheme. We consider two variants of security. For all games, the distribution of the output of the adversaries should be indistinguishable (to achieve security).

Definition 4 (Security of GW-schemes). *Let* Comp *be a* GW*-scheme for some functionality F. For PPT adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and simulators* $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ *consider the following experiments:*

Experiment $REAL_{\mathsf{GW},\mathcal{A}}(1^\lambda)$	*Experiment* $IDEAL_{\mathsf{GW},\mathcal{A},\mathsf{Sim}}(1^\lambda)$
$(F, \mathsf{state}) \leftarrow\!\$\, \mathcal{A}_1(1^\lambda)$;	$(F, \mathsf{state}_\mathcal{A}) \leftarrow\!\$\, \mathcal{A}_1(1^\lambda)$;
$(\mathsf{ek}, \mathsf{bk}, \mathsf{WB}) \leftarrow\!\$\, \mathsf{Comp}(F)$;	$(\widetilde{\mathsf{WB}}, \mathsf{state}_s) \leftarrow\!\$\, \mathsf{Sim}_1(1^{\lvert F \rvert})$;
$\alpha \leftarrow\!\$\, \mathcal{A}_2^{\mathsf{TC}}(\mathsf{WB}, F, \mathsf{state})$;	$\alpha \leftarrow\!\$\, \mathcal{A}_2^{\mathcal{O}}(\widetilde{\mathsf{WB}}, F, \mathsf{state}_\mathcal{A})$;
Return α	*Return* α

Here TC *means the adversary can query the following algorithms:*

– $\mathsf{Encrypt}()$*, by providing a chosen value x and obtaining a ciphertext c.*
– $\mathsf{Encode}()$*, by providing a ciphertext and a value y: (c, y).*

$\mathcal{O}$ means the adversary can query the following oracles, which keep track of state_s*. Note that* $\tilde{\mathsf{ek}}, \tilde{\mathsf{bk}} \in \mathsf{state}_s$*:*

– $\mathcal{O}_{\mathsf{Encrypt}}(x)$*, the oracle runs* Sim_2 *on the length of x, $1^{\lvert x \rvert}$, and returns c_s and* state_s*.*
The oracle sets $X[x] \leftarrow c_s$ and updates the state. The oracle then returns c_s to the adversary.
– $\mathcal{O}_{\mathsf{Encode}}(c_s, y)$*: the oracle retrieves $x \leftarrow X[c_s]$ and runs* Sim_2 *with input $F(x, y), y$ and length of x, and returns $\tilde{c}_s$ and* state_s*.*
The oracle sets $C[c] \leftarrow \tilde{c}_s$ and updates the state. The oracle then returns $\tilde{c}_s$ to the adversary.

Here the simulators only see the size of the function F and the size of the input x. So if the simulators are able to produce outputs that are indistinguishable from the real world, it means the function and the input are hidden in the global white-box scheme.

We say that the global white-box GW*-scheme is secure if there exists a pair of ppt simulators* $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$*, such that for all pairs of ppt adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*, the following two distributions are computationally indistinguishable:*

$$\left\{ \textit{Experiment } REAL_{\mathsf{GW},\mathcal{A}}(1^\lambda) \right\}_{\lambda\in\mathbb{N}} \approx \left\{ \textit{Experiment } IDEAL_{\mathsf{GW},\mathcal{A},\mathsf{Sim}}(1^\lambda) \right\}_{\lambda\in\mathbb{N}}$$

3.2 Security of GW-schemes with user-specific encryption

We now present a variant of the security definition for GW-schemes. The definition below differs in that it ensures that the provider chosen inputs x are now encrypted for a specific user/device. Here the adversary is able to choose which ID values are used for generating the ciphertexts. Additionally, the adversary can choose different Encode() algorithms for encoding the ciphertexts.

Definition 5 (Security of GW-schemes with user-specific encryption). *Let* Comp *be a* GW*-scheme for some functionality F. For PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *and simlulators* $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ *consider the following experiments:*

Experiment $REAL_{\mathcal{A}}(1^\lambda)$	Experiment $IDEAL_{\mathcal{A},\mathsf{Sim}}(1^\lambda)$
$(F, \mathsf{state}) \leftarrow_\$ \mathcal{A}_1(1^\lambda)$;	$(F, \mathsf{state}_{\mathcal{A}}) \leftarrow_\$ \mathcal{A}_1(1^\lambda)$;
$(\mathsf{ek}, \mathsf{bk}, \mathsf{WB}) \leftarrow_\$ \mathsf{Comp}(F)$;	$(\widetilde{\mathsf{WB}}, \mathsf{state}_s) \leftarrow_\$ \mathsf{Sim}_1(1^{\lvert F\rvert})$;
$\alpha \leftarrow_\$ \mathcal{A}_2^{\mathsf{TC}}(\mathsf{WB}, F, \mathsf{state})$;	$\alpha \leftarrow_\$ \mathcal{A}_2^{\mathcal{O}}(\widetilde{\mathsf{WB}}, F, \mathsf{state}_{\mathcal{A}})$;
Return α	*Return* α

Here TC *means the adversary can query the following algorithms:*

- Encrypt()*, with a randomly chosen value x and an* ID*, returning c and* ID*.*
- Encode()*, with a selected pair (c, y) and an* ID *indicating which evaluator to use. The algorithm returns $\tilde{c}$.*

$\mathcal{O}$ *means the adversary can query the following oracles, which keep track of* state_s*. Note that* $\tilde{\mathsf{ek}}, \tilde{\mathsf{bk}} \in \mathsf{state}_s$*. Note that every time the oracle runs the simulator, it provides it with the updated state:*

- $\mathcal{O}_{\mathsf{Encrypt}}(x, \mathsf{ID}_s)$*, the oracle runs* Sim_2 *on* $(1^{\lvert x\rvert}, \mathsf{ID}_s)$ *and returns c_s and* state_s*.*
  The oracle sets $X[x] \leftarrow (c_s, \mathsf{ID}_s)$, updates the state, and returns (c_s, ID_s) to the adversary.
- $\mathcal{O}_{\mathsf{Encode}}(c, y)$*: The oracle retrieves $(c_s, \mathsf{ID}_s) \leftarrow C[c_s]$. It runs* Sim_2 *on $F(x, y)$ and the lengths of x, y, and returns $\tilde{c}$ and* state_s*.*
  The oracle sets $C[\tilde{c}] \leftarrow c$, updates the state, and returns $\tilde{c}$ to the adversary.

We say that the global white-box GW*-scheme is secure if there exists a pair of PPT simulators* $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$*, such that for all pairs of PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*, the following two distributions are computationally indistinguishable:*

$$\left\{ \textit{Experiment } REAL_{\mathsf{GW},\mathcal{A}}(1^\lambda) \right\}_{\lambda\in\mathbb{N}} \approx \left\{ \textit{Experiment } IDEAL_{\mathsf{GW},\mathcal{A},\mathsf{Sim}}(1^\lambda) \right\}_{\lambda\in\mathbb{N}}$$

4 Constructions based on Token-Based Obfuscation

We recall that in token-based obfuscation, the master secret key used for generating the tokens is generated by the same entity which obfuscates the programs. In Definition 3, the compiler Comp is responsible for obfuscating the program and for generating the key material ek, bk. The binding key (bk) is used as the master secret key for the token-generation algorithm. Therefore for our construction, we assume that the provider generating the obfuscated program can securely communicate bk to the trusted components running the Encode algorithm. As an example, we explain below how this could be achieved for mobile-phone applications used in combination with trusted components on phones.

First the provider of a service generates the program WB and the corresponding key material. Next the provider can make the program WB available for download. At this point, a user can download WB, but note that the user still needs to obtain bk in order to encode inputs to WB properly. One way to achieve this is by each user generating a key pair within his trusted component. The user's public key is then given to the service provider who will use this to encrypt bk. Upon receiving the encrypted bk, this can be decrypted by and stored in a user's trusted component, and be used by the token-generation algorithm. In Section 5, we present a strengthened variant of global white-boxes, where we extend our syntax to include this enrollment procedure as part of our white-box programs.

Construction 1. *Let F be any function of the syntax $F(x, y) = z$. Let* SE = (SE.KGen, SE.Enc, SE.Dec) *be a symmetric encryption scheme and let* TBO = (TBO.Obf, TBO.InpGen) *be an obfuscation scheme.*

```
C[K](w)
    c||y ← w
    x ← SE.Dec(K, c)
    z ← F(x, y)
    return z
```

```
Comp[C](F)
    ek ←$ SE.KGen(1^n)
    (MSK, O) ←$ TBO.Obf(C[ek])
    WB ← O
    bk ← MSK
    return ek, bk, WB
```

```
Encrypt(ek, x, ID)
    c ← SE.Enc(ek, x)
    return c
```

```
Encode(bk, c, y)
    MSK ← bk
    w ← c||y
    c̃ ←$ TBO.InpGen(MSK, w)
    return c̃
```

Additionally, we consider an execution algorithm for WB, *which takes as input $\tilde{c}$, runs* WB *on $\tilde{c}$ and outputs the value z, i.e.* WB$(\tilde{c}) = z$.

Theorem 1. *If* SE *a secure symmetric encryption scheme and* TBO *is a secure token-based obfuscation scheme, then Construction 1 is a secure* GW*-scheme.*

The proof of Theorem 1 is a reduction from any TBO scheme since our program WB is the direct output of the obfuscator and it is not possible to compute any value on WB without the token generation algorithm (namely Encode). We first argue the correctness of the program WB.

Proof of correctness. First note that WB is the obfuscated program generated via TBO.Obf, i.e. WB ← O. WB gets as input $\tilde{c}$, which is the output of the TBO.InpGen algorithm ran on the key MSK. Both MSK and WB (i.e. O) are generated via the TBO.Obf algorithm and it follows from the correctness of the TBO scheme that $\tilde{c}$ is a valid input for WB. This means that WB will compute $C[\mathsf{ek}](w)$, i.e. will split w in two and decrypt one half to recover x. From the correctness of the symmetric encryption scheme, it follows that $\mathsf{SE.Dec}(\mathsf{ek}, \mathsf{SE.Enc}(\mathsf{ek}, x)) = x$. Thus, WB correctly retrieves x and computes $F(x, y) = z$. $\square$

Proof of security. To prove the security of Construction 1, it suffices to build a simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ in Definition 4. We build the simulators $\mathsf{Sim}_1, \mathsf{Sim}_2$ from the simulators for the TBO and semantic secure encryption as follows.

- Sim_1 **from the simulators of TBO:** We build Sim_1 by calling the simulator Sim_P used for token-based obfuscation (see Definition 2). In short, Sim_1 runs the Sim_P on the size of the function $|F|$, gets back O_s, and treats O_s as $\tilde{\mathsf{WB}}$, recorded the (possibly) simulated MSK in the state state_s.
- Sim_2 **in collaboration with** $\mathcal{O}_{\mathsf{Encrypt}}$**:** We build this simulator by directly calling the simulator of the semantic secure probabilistic symmetric encryption scheme. Recall the simulator of a semantic secure encryption scheme is supposed to output a string that is indistinguishable from the real ciphertext given the length of the message. For example, for an encryption scheme with pseudorandom ciphertext, the simulator simply draws a random value once given the length of the message.
- Sim_2 **in collaboration with** $\mathcal{O}_{\mathsf{Encode}}$**:** We build this simulator from Sim_I used for token-based obfuscation (see Definition 2). Here Sim_2 runs the Sim_I on values $F(x, y)$ and the lengths of x and y. Sim_I returns a value $\tilde{c}$. Sim_2 then outputs $\tilde{c}$.

We now prove indistinguishability of the real and ideal world experiments. Here, we define a sequence of hybrid experiments. We invoke the security of the underlying SE and TBO schemes to prove that the outcome of the hybrid experiments are indistinguishable.

Hybrid 0. The first hybrid is the output of the ideal experiment in Definition 4.

Hybrid 1. This hybrid is the same as the previous one with the following exception. Instead of generating a ciphertext by running Sim_2 in collaboration with $\mathcal{O}_{\mathsf{Encrypt}}$, we replace it by running the semantic secure encryption algorithm on x. Following the semantic security of SE, Hybrid 1 is indistinguishable from Hybrid 0, since the secret key of the semantic secure encryption scheme is not used in the rest of the experiment in both Hybrid 1 and Hybrid 0.

Hybrid 2. The next hybrid is the same as the previous one with the following exception: the simulated encoded ciphertext (denoted as $\tilde{c}$) is replaced by the encoded ciphertext obtained from running c on the algorithm $\mathcal{O}_{\mathsf{Encode}}$, and the simulated program $\tilde{\mathsf{WB}}$ is replaced by the real token-based obfuscated program. Hybrid 2 is indistinguishable from Hybrid 1 following the security of the TBO.

Hybrid 2 is then the real experiment. $\square$

In Section C we present a TBO-based construction for user specific encodings and prove its security with regard to Definition 5.

Remark on the *key-loading* assumption. As already pointed out, for Construction 1 and Construction 3 we assume that a user is able to load external key material (bk) in its secure element. We note however that this assumption is only necessary when constructing global white-boxes. If we were to construct white-boxes with device binding compiled individually for each user (following the style of [3]), we would not need to make such a key-loading assumption and we would obtain a simpler TBO-based feasibility result. We explain this briefly below and refer the reader to Section 4.1 of [28] for more details on how TBO is constructed.

First we explain why the key-loading assumption is necessary for constructing global white-boxes via TBO. In short, the obfuscated program in a TBO scheme is a garbled circuit which consists of an *encryption* of the original circuit C. Thus whenever running the garbled circuit, we need to provide some key as input so that the original circuit can be recovered and run. If we are considering global white-boxes compiled only once, then all such white-boxes consist of the same circuit encrypted via the same secret key. Thus, this key has to be generated on the server side and shared with all legitimate users who wish to run the white-box on their device.

Now if we were to construct white-boxes compiled individually for each user, such a key could be generated on the user's secure element. The key can be securely shared with the server, who can then use that key for encrypting the circuit when compiling the white-box. The white-box is then shared with the user who can run it via the secret key already stored in his secure element. Below we explain more formally how such a construction and flow would look. We will partly use the notation from Section 4.1 of [28].

1. First the user generates some secret key SK in its secure element and securely sends it to the server. Note that this does not need to be some unique fixed key, but it can be some sub key derived from a unique key, as done in the constructions of [3].
2. The server will now generate the white-box program according to the steps in Section 4.1 of [28]. That is, the server first generates some functional encryption key material. Then in Step 2, the server encrypts circuit C via the SK it obtained from the user. Steps 3, 4 and 5 are performed in the same way as described in the original paper. The server can now share the compiled program with the user.
   We note that to run the generated program (i.e. our white-box), we need (1) the FE public key generated by the server in Step 1 [28, Section 4.1] and (2) the symmetric key SK located on the user's secure element.
3. **Content generation and distribution.** The server generates some content (LUK or DRM), encrypts it, and appends the FE public key, i.e. all ciphertexts are of the form $c = (c, \mathsf{fmpk})$. The server sends the obfuscated program to the user along with all ciphertexts.
4. Now, the user can decrypt c by first sending c to the secure element. The secure element parses $(c, \mathsf{fmpk}) \leftarrow c$, and runs $\mathsf{FE.Enc}(\mathsf{fmpk}, (\mathsf{SK}, c))$ to generate a token. Now the user has a correctly generated token and can run his white-box program on it.

```
$\mathsf{Encrypt}((\mathsf{ek}, \mathsf{fmpk}), x)$
$c \leftarrow \mathsf{SE.Enc}(\mathsf{ek}, x)$
$c \leftarrow (c, \mathsf{fmpk})$
return $c$
```

```
$\mathsf{Encode}(\mathsf{SK}, c)$
$(c, \mathsf{fmpk}) \leftarrow c$
$\tilde{c} \leftarrow\!\$\ \mathsf{FE.Enc}(\mathsf{fmpk}, (\mathsf{SK}, c))$
return $\tilde{c}$
```

It is easy to see that such a construction achieves correctness and the desired security for a white-box program with device binding.

5 Strong Global White-boxes

In the previous sections, we introduced the concept of global white-boxes and presented corresponding definitions. We note that, in those previous definitions, we necessitate a "global" binding key as well. That is, the binding key bk is generated by the provider and is distributed to each device's trusted component through some secure provisioning process. A single compromise of a trusted component would reveal this global binding key to the adversary and thus requires our full confidence in the trusted components. By relaxing this assumption and instead enforcing per-device binding keys we formalize a stronger definition which still allows us to achieve global white-boxes. We refer to this notion as strong global white-box.

In addition to introducing a per-device binding key we make a further change to our syntax. In the previous setting the device-chosen input y is bound to the device together with the provider-chosen input x. In our new setting we only require that the provider-chosen input needs to be device bound. This permits the device greater flexibility by decoupling when binding happens and the additional input y is chosen.

This second change does however impact our definitional style. When we consider functions that take two inputs coming from potentially different parties, this makes it hard to achieve security in a simulation-based setting. When the input of the function is split into two pieces (i.e. the provider-chosen input x and device-chosen input y), a natural simulation-based definition for global white-boxes is impossible to achieve. This is due to similar reasons as for multi-input functional encryption [27], where a natural simulation-based definition for 2-input functional encryption implies VBB obfuscation. As result we switch our definitional style to now provide game (or, indistinguishability)-based definitions to capture the security for this alternate setting.

Our Extended Setting. As in Section 3.1, we consider a setting where a provider creates an obfuscated program (for some functionality F) which will be distributed to a collection of devices. Unlike before, we additionally define the enrollment process for trusted components. The following explanation is a more detailed extension to what was discussed in the second part of Section 1.3. Also see Section 6 for more context. A user *enrolls* his device by calling the trusted component and submitting the result to the provider for certification. More precisely, the trusted component generates a secret-public key pair and some ID value. The user then shares the public key and the ID value with the server. The server will generate a certificate based on this public key and the certificate will be used as an input for the white-box program. More precisely, the server *certifies* the public key via an MRS-like scheme: the server uses its secret key for signing the user's public key. Thus, we obtain a *signed* (or encoded) version of the user's public key, denoted cert, which is sent back to the user.

The provider generates and encrypts the inputs of the white-box program (e.g. the DRM content or the LUKs) and sends the corresponding ciphertexts to the user. For decrypting the ciphertexts, the user first encodes them via its trusted component. The trusted component encodes the ciphertexts via an MRS-like scheme: it signs them using its secret key. The user now gives the encoded ciphertext and the cert as inputs to the white-box program. The white-box program will first *recover* (or decode) the user's public key from the cert and then use that public key to recover the original ciphertext, which afterwards will be decrypted.

Note that ciphertexts will only be recovered correctly if the public key obtained from cert (for the purpose of decoding) is the same public key that was generated during the initial enrollment process. Thus, a user is only able to correctly run the white-box program if he (1) obtains a legitimate cert and (2) encodes the inputs via the secret key of its trusted component. Moreover the cert used must be an encoding, under the secret key of the server, of the public key corresponding to the secret key used to encode inputs to the white-box. Below we define the syntax of each algorithm. Figure 2 gives further clarity on which party runs each algorithm.

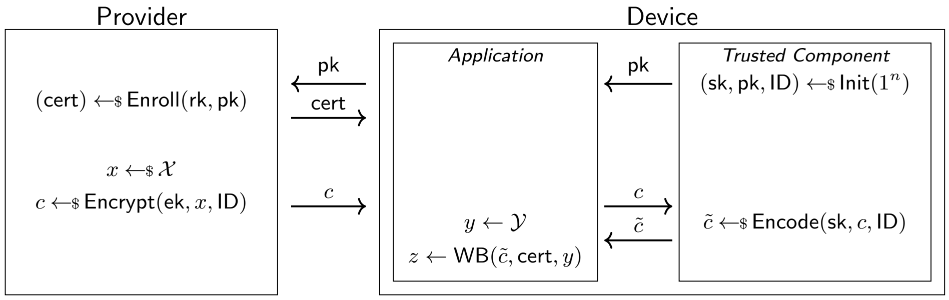
Provider
Device
Application
Trusted Component
$(\mathsf{cert}) \leftarrow_\$ \mathsf{Enroll}(\mathsf{rk}, \mathsf{pk})$
$x \leftarrow_\$ \mathcal{X}$
$c \leftarrow_\$ \mathsf{Encrypt}(\mathsf{ek}, x, \mathsf{ID})$
pk
cert
c
$y \leftarrow \mathcal{Y}$
$z \leftarrow \mathsf{WB}(\tilde{c}, \mathsf{cert}, y)$
pk
c
$\tilde{c}$
$(\mathsf{sk}, \mathsf{pk}, \mathsf{ID}) \leftarrow_\$ \mathsf{Init}(1^n)$
$\tilde{c} \leftarrow_\$ \mathsf{Encode}(\mathsf{sk}, c, \mathsf{ID})$

Fig. 2: *Strong Global White-box (sGW) Scheme.* For initial setup the provider runs the compiler program to generate the whitebox and some key material, $(\mathsf{ek}, \mathsf{rk}, \mathsf{WB}) \leftarrow_\$ \mathsf{Comp}(F)$. WB is distributed to all devices via a central appstore.

Definition 6. *A Strong Global White-box scheme* sGW-Scheme *consists of the following algorithms*

- *A randomized compiler* Comp *which takes as input a function F, with syntax $z \leftarrow F(x, y)$. The compiler returns a program* WB *and two randomly generated keys* rk *and* ek*, i.e.* $(\mathsf{rk}, \mathsf{ek}, \mathsf{WB}) \leftarrow_\$ \mathsf{Comp}(F)$.
- *A probabilistic algorithm* Init*, which on input the security parameter, outputs the following values: a secret key* sk*, an associated public key* pk *and an identifier* ID*, i.e.* $(\mathsf{sk}, \mathsf{pk}, \mathsf{ID}) \leftarrow_\$ \mathsf{Init}(1^n)$.
- *A probabilistic algorithm* Enroll *which takes as input a secret (registration) key* rk *and a request message* pk*, and outputs an authenticated badge* cert*, i.e.* $(\mathsf{cert}) \leftarrow_\$ \mathsf{Enroll}(\mathsf{rk}, \mathsf{pk})$.
- *A probabilistic algorithm* Encrypt *which takes as input a secret (encryption) key* ek*, a value x, and an identifier* ID*, and outputs an encrypted message c, i.e.* $c \leftarrow_\$ \mathsf{Encrypt}(\mathsf{ek}, x, \mathsf{ID})$.
- *A probabilistic algorithm* Encode *which on input of a secret (binding/encoding) key* sk *and a value c, and an identifier* ID *returns a value $\tilde{c}$, i.e.* $\tilde{c} \leftarrow_\$ \mathsf{Encode}(\mathsf{sk}, c, \mathsf{ID})$.

Correctness states that for every genuine rk *and* pk*, such that* $\mathsf{cert} \leftarrow_\$ \mathsf{Enroll}(\mathsf{rk}, \mathsf{pk})$*, for every function $F : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$, for all $x \in \mathcal{X}$, all $y \in \mathcal{Y}$, we have*

$$\Pr[\mathsf{WB}(\mathsf{Encode}(\mathsf{sk}, \mathsf{Encrypt}(\mathsf{ek}, x, \mathsf{ID})), \mathsf{cert}, y) = F(x, y)] = 1,$$

where the probability is over the randomness of all algorithms and the corresponding secret keys sk, ek, rk.

Security Definitions We now introduce our security notions for an sGW-scheme. We provide both a privacy notion, which captures the security of the private inputs x, and a forgery notion with respect to some underlying cryptographic functionality F. The more general properties captured by these security notions are the following. Firstly, they capture the property that a device (trusted component) should be properly enrolled with the provider such that it can run the white-box program correctly. Secondly, the notions capture the property that inputs to the white-box should be properly bound to the device, i.e. properly encoded by the device.

We first formalize the privacy property of an sGW-scheme via the game depicted in Figure 3. We recall that we could have two different types of global white-boxes depending on the application we are considering. If the global white-box is meant to decrypt broadcasted data, e.g. as in DRM applications, then *all* global white-boxes should be able to decrypt all values. In turn if the white-boxes are used for payment applications, the inputs to the white-boxes are user specific. That is, the ciphertexts sent should only be decrypted by one specific user. For our privacy game, we consider the case where information is broadcast to all owners of white-boxes. Thus, we remark that one white-box allows an adversary to decrypt any broadcasted value as long as he has access to a registered hardware. This also implies that the encryptions are not user specific. For this reason, we do not consider the ID values in this game (they will become relevant for a later model). We now describe the security experiment.

The adversary is given access to several oracles which permit him to enroll devices, receive encrypted inputs from the provider, and bind these to a legitimately enrolled device. For the purposes of the experiment, we assume that the trusted component cannot be impersonated (i.e. some secure attestation process ensures the validity of pk submitted to the Enroll oracle). Note that the adversary can run the white-box on any value c encoded via any device which has been properly enrolled, and can enroll as many devices as he wants.[5] The adversary encodes values via the Encode oracle, which he queries with c and a public key value pk which indicates which device he will use for encoding. Additionally, the adversary is given access to a challenge oracle, to which he submits an input x. As output, this oracle returns either the encryption of x or the encryption of a random value r. The adversary wins the security experiment if he can distinguish between these values. Note that to prevent trivial attacks the adversary cannot query the Encode oracle for values which were output by the challenge oracle unless the encoding is meant to be done with a device which has not yet been enrolled. This captures the property that the white-box program should only work properly on inputs encoded via enrolled devices.

Definition 7 (Privacy). *We say that a* sGW*-scheme is private if for all PPT adversaries* $\mathcal{A}$ *playing the privacy game described in Figure 3, the distinguishing advantage* $\Pr[\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{A}}(1^n) = 1] - \frac{1}{2}$ *is negligible.*

An obfuscation scheme which satisfies the privacy notion above ensures that an adversary cannot deduce the private input x shared by the provider. When obfuscating a specific function F, what we actually wish to guarantee is that the adversary cannot evaluate the function F on secret input x without using a legitimate device enrolled with the provider. For instance when considering a mobile payment application, the function F we obfuscate is a message authentication code (MAC). Specifically, $F(x, y) = \mathrm{MAC}(x, y)$, where x is a limited-use key (LUK) for the MAC, and y is the transaction data to be authenticated. Just like for DRM, the Enroll process ensures that only legitimate users who have registered their device are able to run WB correctly. Therefore, for use case such as payments what we ultimately wish to ensure is that an adversary in unable to *forge* the output of an obfuscated program.

An additional difference for mobile payments use case is that the generated ciphertexts should only be decrypted by one specific user. Otherwise, an adversary could download the global white-box, enroll it, and then steal the ciphertexts corresponding to some other user. Thus, the ciphertexts given to a user need to somehow be linked with the user's device, in such way that they can only be decrypted if the white-box program is run on that one

[5]This capability is somewhat similar to the capability an adversary might have to obtain re-compiled versions of a white-box program, introduced by Delerablée et al. in [21] with respect to notions such as security against key extraction, one-wayness, incompressibility and traceability. Each new white-box program is compiled based on different randomness, but on the same secret key, allowing thus to decrypt or encrypt the same values.

```
$\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{A}}(1^n)$
$b \leftarrow\!\$\ \{0,1\}$
$F \leftarrow\!\$\ \mathcal{A}(1^n)$
$\mathcal{C}, \mathcal{P} \leftarrow \emptyset$
$(\mathsf{rk}, \mathsf{ek}, \mathsf{WB}) \leftarrow\!\$\ \mathsf{Comp}(F)$
$b^* \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}}(F, \mathsf{WB})$
return $(b^* = b)$
```

```
$\mathcal{O}_{\mathsf{Init}}()$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\ \mathsf{Init}(1^n)$
$\mathsf{SK}[\mathsf{pk}] \leftarrow \mathsf{sk}$
return $\mathsf{pk}$
```

```
$\mathcal{O}_{\mathsf{Enroll}}(\mathsf{pk})$
if $\mathsf{SK}[\mathsf{pk}] \neq \perp$ and $\mathsf{pk} \notin \mathcal{P}$
    $\mathsf{cert} \leftarrow\!\$\ \mathsf{Enroll}(\mathsf{rk}, \mathsf{pk})$
    $\mathsf{E}[\mathsf{pk}] \leftarrow 1$
    return $\mathsf{cert}$
```

```
$\mathcal{O}_{\mathsf{Encrypt}}(x)$
$c \leftarrow\!\$\ \mathsf{Encrypt}(\mathsf{ek}, x)$
return $c$
```

```
$\mathcal{O}_{\mathsf{Encode}}(c, \mathsf{pk})$
if $c \notin \mathcal{C}$ or $\perp \leftarrow \mathsf{E}[\mathsf{pk}]$
    $\mathsf{sk} \leftarrow \mathsf{SK}[\mathsf{pk}]$
    $\tilde{c} \leftarrow\!\$\ \mathsf{Encode}(\mathsf{sk}, c)$
    $\mathcal{P} := \mathcal{P} \cup \mathsf{pk}$
    return $\tilde{c}$
```

```
$\mathcal{O}_{\mathsf{Chall}}(x)$
$r \leftarrow\!\$\ \mathcal{X}$
if $b = 1$
    $c \leftarrow\!\$\ \mathsf{Encrypt}(\mathsf{ek}, x)$
else
    $c \leftarrow\!\$\ \mathsf{Encrypt}(\mathsf{ek}, r)$
$\mathcal{C} := \mathcal{C} \cup c$
return $c$
```

Fig. 3: Privacy $\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{A}}(1^n)$ security game.

device. Below we introduce our forgery security definition for strong global white-boxes. This definition captures the property that an adversary should not be able to forge valid outputs of the function F for ciphertexts which correspond to some other user's device.

The forgery security experiment and corresponding oracles are depicted in Figure 4. Unlike the privacy experiment, this notion is now parameterized by a specific F. We need this restriction since otherwise, the adversary could choose an F which computes values which are trivial to forge. An additional difference is that we do not let the adversary choose the values to be encrypted by the encryption oracle. This is because knowing a value x would trivially allow the adversary to compute a valid output $F(x, y) = z$. Instead when the adversary queries an encryption oracle, a value is generated at random, encrypted and then the ciphertext is given to the adversary. We note that this restriction seems fair in this model. Namely in this use case, the encrypted values correspond to key material which will later be used for authentication. Therefore, it is normal to assume that these values will be generated at random and will initially be unknown to a user or an adversary. Below we explain how the oracles are defined.

The Init oracle generates key material for a specific device. It also generates an ID value which identifies the device (or the owner of that device). The Enroll oracle is queried via a public key and its corresponding ID. This corresponds to the device for which we wish to obtain a valid certificate in order to run the white-box program on that device. The oracle returns a certificate specific for that device. The Encrypt oracle is queried via a public key and its corresponding ID. The oracle encrypts a randomly generated value according to the specific ID provided by the adversary and keeps track of a list for the plaintext-ciphertext pairs. Note that this ciphertext should only be decrypted correctly via the device corresponding to the given ID.

The Encode oracle is queried by providing a ciphertext (together with its corresponding ID) and a public key (together with its corresponding ID). That is, when querying this oracle, we indicate which device we wish to use for encoding the given ciphertext. If the device we

```
$\mathsf{Exp}^{\mathsf{unf}}_{F,\mathcal{A}}(1^n)$
$\mathcal{C} \leftarrow \emptyset$
$(\mathsf{rk}, \mathsf{ek}, \mathsf{WB}) \leftarrow\!\!\$\ \mathsf{Comp}(F)$
$(c, y, z^*, \overline{\mathsf{pk}}), (x^*, c^*) \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O}}(F, \mathsf{WB})$
$\mathsf{pk}||\mathsf{ID} \leftarrow \overline{\mathsf{pk}}$
$x \leftarrow \mathsf{Dec}(\mathsf{ek}, c, \mathsf{ID})$
if $((c, \overline{\mathsf{pk}}) \notin \mathcal{C}$ or $\perp \leftarrow \mathsf{E}[\overline{\mathsf{pk}}])$
 and $F(x, y) = z^*$
    return 1
or if $\mathsf{X}[c^*] \rightarrow x^*$
    return 1
else return 0

$\mathcal{O}_{\mathsf{Init}}()$
$(\mathsf{sk}, \mathsf{pk}, \mathsf{ID}) \leftarrow\!\!\$\ \mathsf{Init}(1^n)$
$\overline{\mathsf{pk}} \leftarrow \mathsf{pk}||\mathsf{ID}$
$\mathsf{SK}[\overline{\mathsf{pk}}] \leftarrow \mathsf{sk}$
return $\overline{\mathsf{pk}}$

$\mathcal{O}_{\mathsf{Enroll}}(\overline{\mathsf{pk}})$
if $\mathsf{SK}[\mathsf{pk}] \neq \perp$
    $\mathsf{pk}||\mathsf{ID} \leftarrow \overline{\mathsf{pk}}$
    $\mathsf{cert} \leftarrow\!\!\$\ \mathsf{Enroll}(\mathsf{rk}, \mathsf{pk})$
    $E[\overline{\mathsf{pk}}] \leftarrow 1$
    return $\mathsf{cert}$

$\mathcal{O}_{\mathsf{Encrypt}}(\overline{\mathsf{pk}})$
$x \leftarrow\!\!\$\ \mathcal{X}$
$\mathsf{pk}||\mathsf{ID} \leftarrow \overline{\mathsf{pk}}$
$c \leftarrow\!\!\$\ \mathsf{Encrypt}(\mathsf{ek}, x, \mathsf{ID})$
$\mathsf{X}[c] \leftarrow x$
$\bar{c} \leftarrow c||\mathsf{ID}$
return $\bar{c}$

$\mathcal{O}_{\mathsf{Encode}}(\bar{c}, \overline{\mathsf{pk}})$
$c||\mathsf{ID}^* \leftarrow \bar{c}$
$\mathsf{pk}||\mathsf{ID} \leftarrow \overline{\mathsf{pk}}$
if $\mathsf{ID} = \mathsf{ID}^*$
    $\mathcal{C} := \mathcal{C} \cup (c, \overline{\mathsf{pk}})$
$\mathsf{sk} \leftarrow \mathsf{SK}[\overline{\mathsf{pk}}]$
$\tilde{c} \leftarrow\!\!\$\ \mathsf{Encode}(\mathsf{sk}, c, \mathsf{ID})$
return $\tilde{c}$

$\mathcal{O}_{\mathsf{Ver}}(c, y, z, \overline{\mathsf{pk}})$
$\mathsf{pk}||\mathsf{ID} \leftarrow \overline{\mathsf{pk}}$
$x \leftarrow \mathsf{Dec}(\mathsf{ek}, c, \mathsf{ID})$
if $F(x, y) = z$
    return 1
else return $\perp$
```

Fig. 4: Forgery security game.

mean to use has the same ID as the ciphertext, then we store this ciphertext-public key pair in a set. Namely for this pair, we know that the resulting encoding should be properly *decoded* by the global white-box. Thus having such an encoded value would let the adversary trivially generate a valid value z.

We now explain the winning conditions. The adversary outputs two tuples, each relevant to one different winning condition. For the first tuple $(z^*, c, y, \overline{\mathsf{pk}})$, the adversary wins if his given values compute to $F(\mathsf{Dec}(\mathsf{ek}, c, \mathsf{ID}), y) = z^*$, as long as one of the following holds: (1) The public key ID provided for the decryption of c, and c, were not used together for querying the Encode oracle; or, (2) the device public key ID provided for the decryption of c has not been enrolled yet. This captures the general property of device enrollment mentioned above. For the second tuple (c^*, x^*), the adversary wins if the provided value c^* corresponds to a ciphertext encrypting x^*.

Alternative winning condition. With the second winning condition we wish to capture that a value x is never leaked during its decryption and computation with the white-box program. In practice, we usually obfuscate the complete white-box program including function F and we wish to ensure the privacy of such values x. Namely, if such values x are leaked, an adversary would have an easier way of attacking a user's application by simply observing how they are once computed by the white-box, bypassing an adversary's need to perform a code-lifting attack. Note that without this second winning condition, a white-box program which does not obfuscate F would be secure in the model.

Definition 8 (Forgery). *We say that an* sGW*-scheme in combination with some function F is unforgeable if all PPT adversaries $\mathcal{A}$ have a negligible probability of winning the* $\mathsf{Exp}^{\mathsf{forgery}}_{\mathcal{A},F}(1^n)$ *game in Figure 4.*

6 Message-Recoverable Signatures for sGW-Schemes

We recall that a message-recoverable signature scheme consists of a signing and a *recover* algorithm. Here, the signing algorithm uses a secret key to sign and encode a message, while the recover algorithm uses a public key to recover the message (or an error if the signature is invalid).

We now recall how MRSs are used in practice to implement device binding. First the device's trusted component generates a key pair $(\mathsf{sk}, \mathsf{pk})$. The public key is given to the provider who embeds this within the white-box application. That is, the white-box application has the user's public key hard-coded in it, plus its decryption key. Now to use the white-box application, the device first calls its trusted component to sign/encode the input using sk. During the white-box application's execution, it uses the embedded public key pk to recover the input message. Following successful recovery, the white-box proceeds to perform the other functionality embedded, e.g. a decryption.

However, this approach requires a different white-box application to be created for every device (note that we encode a different public key for each device). The next question is how can we enhance the discussed approach to provide a secure obfuscator that is used to construct a single, strong global white-box. To address this, we use two layers of recover algorithms within our white-box program. The first layer will recover the user's public key, which has been signed by the provider as a method of enrolling this device within the system. The second recover algorithm uses the recovered public key to recover the input which was signed by the trusted component. Below we provide a pseudocode description of the algorithms for the case that WB is a white-box decryption program. Here, WB is bound to the Encode component and can only be used correctly if the corresponding pk_1 has been signed by the Enroll algorithm.

```
$\mathsf{Enroll}(\mathsf{sk}_1, \mathsf{pk}_2)$
    $\mathsf{cert} \leftarrow\$ \mathsf{Sig}(\mathsf{sk}_1, \mathsf{pk}_2)$
    return $\mathsf{cert}$

$\mathsf{Encode}[\mathsf{sk}_2](c)$
    $\tilde{c} \leftarrow\$ \mathsf{Sig}(\mathsf{sk}_2, c)$
    return $\tilde{c}$

$\mathsf{WB}(\mathsf{cert}, \tilde{c})$
    $\mathsf{pk}_2 \leftarrow \mathsf{Rcvr}(\mathsf{pk}_1, \mathsf{cert})$
    $c \leftarrow \mathsf{Rcvr}(\mathsf{pk}_2, \tilde{c})$
    $m \leftarrow \mathsf{Dec}(k, c)$
    return $m$
```

In practice, we would usually instantiate such a construction with efficient and standard primitives. For example, we would use RSA for the public-key operations and AES-based schemes for symmetric-key decryption. On top of that, we would apply some efficient form of obfuscation to the circuit describing WB to hide the symmetric key k and to stop an adversary from separating the decryption program from the recover algorithms.

For our formal constructions, we use indistinguishability obfuscation together with iO-friendly, puncturable pseudorandom functions. Let us remark that we can also view the construction as using general-purpose obfuscation together with normal signature with message recovery (such as RSA), since all the existing iO candidates can be viewed as candidate VBB obfuscators except for the "self-referring" programs used as the counterexamples of VBB [7]. Such a view is also used in other works [26]. The use of puncturable signature and other iO-friendly primitives is for achieving a feasibility result with provable security guarantee.

Instantiating our construction. We apply indistinguishability obfuscation to our circuit in order to achieve the desired security. Recall that for iO, we need two circuits which are functional equivalent but differ in their description. Specifically for a white-box design, our functional equivalent circuits should differ on their sensitive information, ensuring that an adversary is not able to extract that sensitive information from the obfuscated program. Thus, we construct our circuit using puncturable primitives.

Puncturable signature schemes have been presented by both Bellare et al. [9], and by Sahai and Waters [43] for short signatures. Both the schemes are based on PPRFs and provide a public verification algorithm which lets a user verify the validity of a signature. Recall however that in our scheme, we need a verification algorithm which actually recovers a signed message, instead of only verifying the validity of the signature. Thus, we need to extend the punctured schemes mentioned above. One possibility would be to use the CCA-secure public-key encryption scheme presented by Sahai and Waters [43, Section 5.3] as we explain below. Here, the authors build a CCA-PKE scheme by using two puncturable PRF keys K_1 and K_2 as a secret key for decrypting messages. Then they create an obfuscation of a program which uses the same keys K_1 and K_2 to perform the inverse operation (encryption). In this context, the obfuscated program is seen as a public key, since anybody can use it for encrypting messages but only the owner of the keys K_1 and K_2 can decrypt those messages.

We could construct a puncturable message-recoverable signature scheme using the same method as above, but by swapping the encryption and decryption functions. That is, the secret keys would be used for *encrypting* messages (i.e., signing in an MRS) and the obfuscated program would be used for recovering the message from a signature. Then, if we can show that the recover algorithm is puncturable, then we could construct a program of the form $\mathsf{Dec}(k^*, \mathsf{Rcvr}_2(\mathsf{Rcvr}_1(\mathsf{pk}^*, \cdot)))$.

In this paper however, we choose a more direct approach, where we define one single circuit Γ, which corresponds to all three operations. That is, in the circuit we define groups of operations corresponding to the first recover, the second recover and the decryption algorithm. Below we give more details to our construction and to how it maps a construction using actual signatures with message recovery.

6.1 Construction via PPRFs

$\Gamma[K_1, K_2, \mathsf{ek}, F](\mathsf{cert}, \tilde{c}, y)$

```
1 :  $\psi, c_1, c_2 \leftarrow \mathsf{cert}$
2 :  if $\mathtt{PRG}(c_2) \neq \mathtt{PRG}(\mathtt{PPRF}(K_2, \psi || c_1))$
3 :    return $\perp$
4 :  else
5 :    $w \leftarrow \mathtt{PPRF}(K_1, \psi)$
6 :    $\mathsf{pk}' \leftarrow c_1 \oplus w$
7 :    $K_1' || K_2' \leftarrow \mathsf{pk}'$
8 :    $\psi', c_1', c_2' \leftarrow \tilde{c}$
9 :    if $\mathtt{PRG}(c_2') \neq \mathtt{PRG}(\mathtt{PPRF}(K_2', \psi' || c_1'))$
10 :      return $\perp$
11 :    else
12 :      $w' \leftarrow \mathtt{PPRF}(K_1', \psi')$
13 :      $c \leftarrow c_1' \oplus w'$
14 :      $t, r \leftarrow c$
15 :      $v \leftarrow \mathtt{PPRF}(\mathsf{ek}, r)$
16 :      $x \leftarrow t \oplus v$
17 :      return $F(x, y)$
```

In this section we present our construction for a strong global white-box. Note that in the construction in this section, the key generation algorithms generate a key pair K_1, K_2 (and K_1', K_2'), where each pair is used as a secret key within the construction. We now provide some explanations with regard to how our construction maps the approach using MRS.

First, we focus on the circuit Γ which we will obfuscate, shown on the right. Lines 1 to 7 recover the user's key material. That is, these lines correspond to the recover algorithm which, ran on the server's public key, recovers the user's public key. Lines 8 to 13 use the user's key material to recover the ciphertext c. These lines correspond thus to the recover algorithm ran on the user's public key and on the encoded value $\tilde{c}$. Finally, lines 14 to 17 correspond to the decryption of c. Note that in this case we care only about achieving privacy for our global white-box, the function F will just output the value x.

Below we show the rest of our construction. Note that the ID variable introduced in the syntax of Definition 6 is not used in this construction and thus we omit it in the following description. We explain shortly what each algorithm does. The compiler (ran by the server or trusted entity) will generate the keys K_1, K_2, ek and embed them in the circuit Γ. Then it runs an indistinguishability obfuscator on the circuit, generating the white-box program. The Init (trusted hardware component on the user's device) generates the key material $K' = K'_1||K'_2$ and let the obfuscated programs $\mathtt{iO}(\mathtt{PPRF}(K'_1, \cdot))$, $\mathtt{iO}(\mathtt{PPRF}(K'_2, \cdot))$ be the public key of K', denoted as pk'. Enroll is the process ran by the trusted entity for certifying a user's public key, i.e. it corresponds to the process of signing the public key of the user by means of the secret key of the server. We are interested in the algorithm providing a notion of integrity for the generated value cert. $\mathsf{Encrypt}$ is ran by the trusted entity and simply encrypts some value x via the symmetric encryption key ek. Finally Encode corresponds to the process of encoding the inputs of the white-box program via a signing key located on the device's trusted component. Here, we are also interested in achieving a notion of integrity for the output values $\tilde{c}$.

```
Comp(F)
1 : $(K_1, K_2) \leftarrow\$ \{0,1\}^\lambda$
2 : $\mathsf{rk} \leftarrow K_1||K_2$
3 : $\mathsf{ek} \leftarrow\$ \{0,1\}^\lambda$
4 : $\mathsf{WB} \leftarrow\$ \mathtt{iO}(\Gamma[K_1, K_2, \mathsf{ek}, F])$
5 : return $\mathsf{rk}, \mathsf{ek}, \mathsf{WB}$
```

```
Encrypt(ek, (x))
1 : $r \leftarrow\$ \{0,1\}^n$
2 : $v \leftarrow \mathtt{PPRF}(\mathsf{ek}, r)$
3 : $t \leftarrow x \oplus v$
4 : $c \leftarrow (t, r)$
5 : return $c$
```

```
Enroll(rk, (pk'))
1 : $K_1||K_2 \leftarrow \mathsf{rk}$
2 : $\psi \leftarrow\$ \{0,1\}^\lambda$
3 : $w \leftarrow \mathtt{PPRF}(K_1, \psi)$
4 : $c_1 \leftarrow \mathsf{pk}' \oplus w$
5 : $c_2 \leftarrow \mathtt{PPRF}(K_2, \psi||c_1)$
6 : $\mathsf{cert} \leftarrow (\psi, c_1, c_2)$
7 : return $\mathsf{cert}$
```

```
Encode(K'_1, K'_2, (c))
1 : $\psi' \leftarrow\$ \{0,1\}^\lambda$
2 : $w' \leftarrow \mathtt{PPRF}(K'_1, \psi')$
3 : $c'_1 \leftarrow c \oplus w'$
4 : $c'_2 \leftarrow \mathtt{PPRF}(K'_2, \psi'||c'_1)$
5 : $\tilde{c} \leftarrow (\psi', c'_1, c'_2)$
6 : return $\tilde{c}$
```

```
Init(1^λ)
1 : $(K'_1, K'_2) \leftarrow\$ \{0,1\}^\lambda$
2 : $K' \leftarrow K'_1||K'_2$
3 : $\mathsf{pk}' \leftarrow \mathtt{iO}(\mathtt{PPRF}(K'_1, \cdot)), \mathtt{iO}(\mathtt{PPRF}(K'_2, \cdot))$
4 : return $(K', \mathsf{pk}')$
```

Construction 2. *Let the circuit Γ be as described above. Let F be any function of the syntax $F(x, y) = z$. Let* $\mathtt{iO}$ *be an indistinguishability obfuscator. Let PRG be a pseudorandom generator and* $\mathtt{PPRF}$ *be a secure puncturable PRF.*

Theorem 2. *Let* $\mathtt{iO}$ *be a an indistinguishability obfuscator. Let PRG be a pseudorandom generator, and* $\mathtt{PPRF}$ *a secure puncturable pseudorandom function. Then Construction 2 is a privacy-secure* sGW *scheme.*

Proof. To prove the privacy property of our sGW scheme, we place our construction on the privacy game presented in this section. Then we go through a series of hybrid experiments where we argue on the security properties of the primitives used within our construction. When we reach the final hybrid, we show that the adversary has no advantage of winning the game.

The first hybrid corresponds to our privacy game in Figure 3. We recall that in this game, we care that the adversary does not learn anything about the value x unless he has access to an enrolled hardware device.

- **Hybrid 0:** The keys $K_1, K_2, \mathsf{ek}, K'_1, K'_2$ are generated at random. Recall that the keys K_1, K_2, ek will be used for compiling the white-box program.
  The outputs of the Enroll, Init, Encrypt and Encode oracles are obtained in the same way as described for the algorithms in Construction 2 with the corresponding names. The challenge oracle generates the ciphertext z^* by encrypting either a value x or r (depending on the bit value of b). The encryption is performed in the same way as described for the Encrypt algorithm in Construction 2 and b is chosen at random.
  Note that in the program Γ, in Line 7, the decrypted pk' should be the iOed program of K'_1 and K'_2, but inside the program we denote them as K'_1 and K'_2 for simplicity.
- **Hybrid 1:** The same as Hybrid 0, except that we will puncture the keys on the points defined as follows and change the program Γ to a functionally equal one, which we denote Γ^*.
  1. Let $K'^* = K'^*_1 || K'^*_2$ denote the key K' of the protected user.
  2. Let ψ'^* denote the value of ψ' used by the encode oracle when our user calls it on c^*.
  3. Let $w'^* \leftarrow \mathtt{PPRF}(K'_1, \psi'^*)$.
  4. Let c'^*_1 be $c^* \oplus w'^*$.
  5. Let $c'^*_2 \leftarrow \mathtt{PPRF}(K'_2, \psi'^* || c'^*_1)$
  6. Puncture K'^*_1 on ψ'^* and get $K'^*_1\{\psi'^*\}$.
  7. Puncture K'^*_2 on $\psi'^* || c'^*_1$ and get $K'^*_2\{\psi'^* || c'^*_1\}$.
  8. Let ψ^* denote the value of ψ used by the enroll oracle when our user calls it on K'^*.
  9. Let w^* be $\mathtt{PPRF}(K_1, \psi^*)$.
  10. Let $c_1{}^*$ be $\big(\mathtt{iO}(\mathtt{PPRF}(K'^*_1\{\psi'^*\}, \cdot)) || \mathtt{iO}(\mathtt{PPRF}(K'^*_2\{\psi'^* || c'^*_1\}, \cdot))\big) \oplus w^*$.
  11. Let $c_2{}^*$ be $\mathtt{PPRF}(K_2, \psi^* || c_1{}^*)$.
  12. Puncture K_1 on ψ^* and get $K_1\{\psi^*\}$.
  13. Puncture K_2 on $\psi^* || c_1{}^*$ and get $K_2\{\psi^* || c_1{}^*\}$.
  14. Let r^* denote the value of r used when generating the challenge ct c^*.
  15. Let v^* be $\mathtt{PPRF}(\mathsf{ek}, r^*)$.
  16. Let c^* be $(x \oplus w^*, r^*)$.
  17. Puncture ek on r^* and get $\mathsf{ek}\{r^*\}$.
  18. Finally, generate
      $\Gamma[K_1\{\psi^*\}, K_2\{\psi^* || c_1{}^*\}, c_2{}^*, K'^*_1\{\psi'^*\}, K'^*_2\{\psi'^* || c'^*_1\}, c'^*_2, c^*, \mathsf{ek}\{r^*\}, v^*, F]$ as in Figure 5.
      Note that all the values in the keys have been defined.

  The circuits Γ and Γ^* in Hybrids 0 and 1 respectively are functional equivalent and this game hop reduces to $\mathtt{iO}$ security.
  Remark: note that on lines 17 and 20 of Γ^* the circuit uses non-punctured keys K'_1 and K'_2. This has the following reasoning: if K' is defined as in line 3, this means that the circuit is using a punctured key $K'^*_1\{\psi'^*\} || K'^*_2\{\psi'^* || c'^*_1\}$ as $K'_1 || K'_2$. In all other cases, the circuit uses a non-punctured key $K' = K'_1 || K'_2$.

```
$\Gamma^*[K_1\{\psi^*\}, K_2\{\psi^*||{c_1}^*\}, {c_2}^*, {K'_1}^*\{{\psi'}^*\}, {K'_2}^*\{{\psi'}^*||{c'_1}^*\}, {c'_2}^*, c^*, \mathsf{ek}\{r^*\}, v^*, F](\mathsf{cert}, \tilde{c}, y)$

1 :  $\psi, c_1, c_2 \leftarrow \mathsf{cert}$
2 :  if $\psi||c_1 = \psi^*||{c_1}^*$ and $\texttt{PRG}(c_2) = \texttt{PRG}({c_2}^*)$
3 :    $\mathsf{pk}' \leftarrow {K'_1}^*\{{\psi'}^*\}, {K'_2}^*\{{\psi'}^*||{c'_1}^*\}$
4 :  else if $\psi = \psi^*$ and $(c_1 \neq {c_1}^*$ or $\texttt{PRG}(c_2) \neq \texttt{PRG}({c_2}^*))$
5 :    return $\perp$
6 :  else if $\texttt{PRG}(c_2) \neq \texttt{PRG}(\texttt{PPRF}(K_2\{\psi^*||{c_1}^*\}, \psi||c_1))$
7 :    return $\perp$
8 :  else
9 :    $w \leftarrow \texttt{PPRF}(K_1\{\psi^*\}, \psi)$
10 :   $\mathsf{pk}' \leftarrow c_1 \oplus w$
11 :   $K'_1||K'_2 \leftarrow \mathsf{pk}'$
12 :   $\psi', c'_1, c'_2 \leftarrow \tilde{c}$
13 :   if $\psi'||c'_1 = {\psi'}^*||{c'_1}^*$ and $\texttt{PRG}(c'_2) = \texttt{PRG}({c'_2}^*)$
14 :     $c \leftarrow c^*$
15 :   else if $\psi' = {\psi'}^*$ and $(c'_1 \neq {c'_1}^*$ or $\texttt{PRG}(c'_2) \neq \texttt{PRG}({c'_2}^*))$
16 :     return $\perp$
17 :   else if $\texttt{PRG}(c'_2) \neq \texttt{PRG}(\texttt{PPRF}(K'_2, \psi'||c'_1))$
18 :     return $\perp$
19 :   else
20 :     $w' \leftarrow \texttt{PPRF}(K'_1, \psi')$
21 :     $c \leftarrow c'_1 \oplus w'$
22 :     $t, r \leftarrow c$
23 :     if $r = r^*$
24 :       $v \leftarrow v^*$
25 :     else
26 :       $v \leftarrow \texttt{PPRF}(\mathsf{ek}\{r^*\}, r)$
27 :       $x \leftarrow t \oplus v$
28 :       return $F(x, y)$
```

Fig. 5: Γ^* in Hybrid 1.

– **Hybrid 2:** the same as Hybrid 1 with the following exceptions:
  1. We replace c_2^* by $c_2^* \leftarrow\!\$\, \{0,1\}^\lambda$.
  2. We then calculate $h \leftarrow\!\$\, \texttt{PRG}(c_2^*)$.
  3. We hard-code h in $\mathit{\Gamma}^*$ and replace $\texttt{PRG}(c_2^*)$ by h on lines 2 and 4.

  This game hop reduces to PPRF- and $\texttt{iO}$-security.
– **Hybrid 3:** the same as Hybrid 2 with the following exceptions:
  1. We replace ${c'_2}^*$ by ${c'_2}^* \leftarrow\!\$\, \{0,1\}^\lambda$.
  2. We then calculate $h' \leftarrow\!\$\, \texttt{PRG}({c'_2}^*)$.
  3. We hard-code h' in $\mathit{\Gamma}^*$ and replace $\texttt{PRG}({c'_2}^*)$ by h' on lines 13 and 15.

  This game hop reduces to PPRF- and $\texttt{iO}$-security.
– **Hybrid 4:** the same as Hybrid 3 with the following exceptions:
  1. $h' \leftarrow\!\$\, \{0,1\}^{2\lambda}$.
  2. Line 14 and c^* is no longer needed in the description of $\mathit{\Gamma}^*$ since Line 13 will be satisfied only with negligible probability.

  This game hop reduces to PRG- and $\texttt{iO}$-security.
– **Hybrid 5:** the same as Hybrid 4 with the following exceptions:
  1. We replace v^* by $v^* \leftarrow\!\$\, \{0,1\}^\lambda$.

  This game hop reduces to PPRF-security.

  Finally, we use the fact that v^* is random to argue that the decrypted x^* is random in $\mathit{\Gamma}$ for both $b = 0$ and $b = 1$. This completes the proof for the privacy game.

□

In Section D we provide a construction for user specific encryptions, where the values c are only meant to be decrypted by one specific user. Such a construction is suited for a mobile payment application, where the value c corresponds to a limited use key which will be decrypted and used for generating a message authentication code. The construction is a simple extension of Construction 2 as it just adds two further PPRFs to the circuit $\mathit{\Gamma}$.

References

1. Estuardo Alpirez Bock, Alessandro Amadori, Joppe W. Bos, Chris Brzuska, and Wil Michiels. Doubly half-injective prgs for incompressible white-box cryptography. In Mitsuru Matsui, editor, *Topics in Cryptology - CT-RSA 2019 - The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4-8, 2019, Proceedings*, volume 11405 of *Lecture Notes in Computer Science*, pages 189–209. Springer, 2019.
2. Estuardo Alpirez Bock, Alessandro Amadori, Chris Brzuska, and Wil Michiels. On the security goals of white-box cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):327–357, 2020.
3. Estuardo Alpirez Bock, Chris Brzuska, Marc Fischlin, Christian Janson, and Wil Michiels. Security reductions for white-box key-storage in mobile payments. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 221–252, Cham, 2020. Springer International Publishing.
4. Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In *CRYPTO (1)*, volume 9215 of *Lecture Notes in Computer Science*, pages 308–326. Springer, 2015.
5. C. H. Baek, J. H. Cheon, and H. Hong. White-box aes implementation revisited. *Journal of Communications and Networks*, 18(3):273–287, June 2016.
6. Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, and Martin Bjerregaard Jepsen. Analysis of software countermeasures for whitebox encryption. *IACR Trans. Symmetric Cryptol.*, 2017(1):307–328, 2017.
7. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, pages 1–18, 2001.
8. Mihir Bellare and Phillip Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. pages 399–416, 1996.

9. Mihir Bellare, Igors Stepanovs, and Brent Waters. New negative results on differing-inputs obfuscation. pages 792–821, 2016.
10. Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. pages 227–240, 2004.
11. Alex Biryukov, Charles Bouillaguet, and Dmitry Khovratovich. Cryptographic schemes based on the ASASA structure: Black-box, white-box, and public-key (extended abstract). pages 63–84, 2014.
12. Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. *J. ACM*, 65(6):39:1–39:37, 2018.
13. Andrey Bogdanov and Takanori Isobe. White-box cryptography revisited: Space-hard ciphers. pages 1058–1069, 2015.
14. Andrey Bogdanov, Takanori Isobe, and Elmar Tischhauser. Towards practical whitebox cryptography: Optimizing efficiency and space hardness. pages 126–158, 2016.
15. Julien Bringer, Herve Chabanne, and Emmanuelle Dottax. White box cryptography: Another attempt. Cryptology ePrint Archive, Report 2006/468, 2006. http://eprint.iacr.org/2006/468.
16. Yilei Chen, Craig Gentry, and Shai Halevi. Cryptanalyses of candidate branching program obfuscators. In *EUROCRYPT (3)*, volume 10212 of *Lecture Notes in Computer Science*, pages 278–307, 2017.
17. Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the CLT13 multilinear map. *J. Cryptology*, 32(2):547–565, 2019.
18. Jihoon Cho, Kyu Young Choi, Itai Dinur, Orr Dunkelman, Nathan Keller, Dukjae Moon, and Aviya Veidberg. Wem: A new family of white-box block ciphers based on the even-mansour construction. In Helena Handschuh, editor, *Topics in Cryptology – CT-RSA 2017*. Springer International Publishing, 2017.
19. Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. pages 250–270, 2003.
20. Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for DRM applications. In Joan Feigenbaum, editor, *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002*, volume 2696 of *LNCS*, pages 1–15. Springer, 2003.
21. Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. White-box security notions for symmetric encryption schemes. pages 247–264, 2014.
22. Patrick Derbez, Pierre-Alain Fouque, Baptiste Lambin, and Brice Minaud. On recovering affine encodings in white-box implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):121–149, Aug. 2018.
23. EMVCo. Emv mobile payment: Software-based mobile payment security requirements, 2019. https://www.emvco.com/wp-content/uploads/documents/EMVCo-SBMP-16-G01-V1.2_SBMP_Security_Requirements.pdf.
24. Pierre-Alain Fouque, Pierre Karpman, Paul Kirchner, and Brice Minaud. Efficient and provable white-box primitives. pages 159–188, 2016.
25. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, pages 40–49, 2013.
26. Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *FOCS*, pages 404–413. IEEE Computer Society, 2014.
27. Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 578–602. Springer, 2014.
28. Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 555–564. ACM, 2013.
29. Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of white box DES implementations. pages 278–295, 2007.
30. Satoshi Hada. Zero-knowledge and code obfuscation. In *ASIACRYPT*, pages 443–457, 2000.
31. Matthias Jacob, Dan Boneh, and Edward W. Felten. Attacking an obfuscated cipher by injecting faults. In Joan Feigenbaum, editor, *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002, Washington, DC, USA, November 18, 2002, Revised Papers*, volume 2696 of *LNCS*, pages 16–31. Springer, 2003.
32. Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. *IACR Cryptol. ePrint Arch.*, 2020:1003, 2020.
33. Mohamed Karroumi. Protecting white-box AES with dual ciphers. pages 278–291, 2011.
34. Jihoon Kwon, ByeongHak Lee, Jooyoung Lee, and Dukjae Moon. FPL: white-box secure block cipher using parallel table look-ups. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The*

Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings, volume 12006 of *Lecture Notes in Computer Science*, pages 106–128. Springer, 2020.

35. Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart Preneel. Two attacks on a white-box AES implementation. pages 265–285, 2014.
36. Huijia Lin. Indistinguishability obfuscation from constant-degree graded encoding schemes. In *EUROCRYPT (1)*, volume 9665 of *Lecture Notes in Computer Science*, pages 28–57. Springer, 2016.
37. Hamilton E. Link and William D. Neumann. Clarifying obfuscation: Improving the security of white-box encoding. Cryptology ePrint Archive, Report 2004/025, 2004. http://eprint.iacr.org/2004/025.
38. Wil Michiels. Device binding from digital signatures. Personal Communication.
39. James A. Muir. A tutorial on white-box AES. Cryptology ePrint Archive, Report 2013/104, 2013. http://eprint.iacr.org/2013/104.
40. Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao-Lai white-box AES implementation. pages 34–49, 2013.
41. Yoni De Mulder, Brecht Wyseur, and Bart Preneel. Cryptanalysis of a perturbated white-box AES implementation. pages 292–310, 2010.
42. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93, 2005.
43. Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. pages 475–484, 2014.
44. Eloi Sanfelix, Job de Haas, and Cristofaro Mune. Unboxing the white-box: Practical attacks against obfuscated ciphers. Presentation at BlackHat Europe 2015, 2015. https://www.blackhat.com/eu-15/briefings.html.
45. Smart Card Alliance Mobile and NFC Council. Host card emulation 101. white paper, 2014. https://www.securetechalliance.org/wp-content/uploads/HCE-101-WP-FINAL-081114-clean.pdf.
46. Brecht Wyseur. White-box cryptography. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed*, pages 1386–1387. Springer, 2011.
47. Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. pages 264–277, 2007.
48. Yaying Xiao and Xuejia Lai. A secure implementation of white-box AES. In *2009 2nd International Conference on Computer Science and its Applications*, pages 1–6. IEEE Computer Society, 2009.

A Protecting Mobile-Payment Applications.[6]

Payment applications run on the mobile phone of the client and communicates with the terminal via near field communication (NFC). In this context, the mobile payment applications act as a substitute of traditional credit cards and they store unique client-related information used for authentication. Moreover, the applications should allow a user to perform payments even when not having access to the internet. While these applications open up to new possibilities in the payment industry, special care needs to be taken when implementing them in order to provide the desired security.

The applications run on mobile phones which support the use of many other applications and they might be subject to invasive attacks, from adversaries in the form of malware with access to the application code. Here, we wish to stop such an adversary from stealing user-related information and use it for their own purposes. Therefore, user-related information is stored in encrypted form and only decrypted at the moment they are needed, i.e. for performing a payment transaction. Ideally, cryptographic related keys are stored in hardware-based secure elements supported by the mobile phones. This way, an adversary as described above has no way of compromising such cryptographic keys since he does not get access to the hardware of the phone. However as of today, not all mobile phones support secure elements. For instance, many low end phones which use the Android operating system do not support a secure element (note however that all Apple phones do support a secure element). To achieve a broader coverage of phones, payment applications can also be implemented in software only and are able to use the NFC controller via host card emulation (HCE). Clearly, further security measures need to be taken into consideration for applications implemented in software only. Specially since the cryptographic related keys need to be stored in a secure way such that they cannot be compromised or misused. Here, white-box cryptography has been proposed as a main software countermeasure technique for protecting such keys.

Since different mobile phones provide different hardware and software components, it is not trivial to design a mobile payment application which achieves the best possible security for all phones. Moreover, a different application needs to be designed for a different operation system. Consider for instance the two biggest players in the mobile phone industry as of today. iOS software runs only and exclusively on Apple phones. As mentioned before, all Apple phones are supported by a secure element. Therefore, all mobile payment applications designed to run on iOS can be implemented in such a way that they rely on the secure elements for the secure storage of cryptographic keys. However for the Android operating system, the story looks different. Android OS runs on multiple devices of different manufacturers such as Samsung, Sony and Huawei. As mentioned above, only a fraction of the mobile phones which support Android include a secure element as part of their hardware. In this case, we could design two classes of payment applications for Android: one which relies on the secure element for the key storage and one which relies on other software countermeasure techniques, such as white-box cryptography. However, designers usually prefer to take a more flexible approach, where the same class of application can run on any type of device. In this case, the Trusted Execution Environment (TEE) from the Android OS could be of great use.

TEE and Android Key Store Android implements an isolated fraction of its operating system known as its TEE. Only specific applications with specific permissions can be exe-

[6] All trademarks are the property of their respective owners, are used for identification purposes only, and do not necessarily imply product endorsement or affiliation with Visa.

cuted on the TEE. Note that making it difficult to access the TEE for normal applications also makes it difficult to install malware running *on* the TEE. Moreover, for phones supporting a secure element, the TEE is hardware based. Designing a payment application which runs on the TEE of different manufacturers seems out of reach in many cases[7] since the payment application will come from a party separated from either Android or the hardware manufacturer. However, we could use some functionalities provided by the applications running on the TEE in order to increase the security of our payment applications. One such application is the Android KeyStore. The Android KeyStore is used for the storage of cryptographic keys, which can be used exclusively within the KeyStore. With this in mind, one idea could be to design our applications such that all cryptographic operations are performed on the KeyStore. Unfortunately, this would lead to the following drawbacks, especially for applications implementing a larger number of cryptographic operations:

– **Delays:** Accessing the KeyStore implies a context switch within the operating system. Moreover, the KeyStore allows to perform a cryptographic operation and then provides its output. For the case that our application performs more than one cryptographic operation, we would need to access the KeyStore several times.
– **Information leakage:** The TEE has a clear API interface, such that it is relatively simple to intercept the data that is output from it. This introduces a vulnerability for the cases when the output of a cryptographic operation needs to remain secret. Namely, a white-box adversary located within the operating system can easily listen to the outputs coming from the TEE and use these outputs for his own purposes.
– **Limits of the functionalities:** The KeyStore includes a rich set of cryptographic API's which allow the execution of widely used cryptographic operations (e.g. AES, RSA, ECC, etc.). However, it might be the case that our application implements some operations which are not supported by these APIs. This might be the case that our application implements non-standardized or less popular ciphers. For such cases, we are not able to count on the KeyStore for the execution of such operations and we need to look out for alternatives.
– **Dependency on the hardware vendors:** For the providers of the mobile application, it might be desired to have as much independence from the hardware manufacturers as possible. Note that Android OS is supported by phones provided by at least 10 different independent manufacturers. Delegating cryptographic operations to the Android KeyStore implies delegating such operations to the hardware provided by those manufacturers, specially for the case that the KeyStore runs on a secure element.

These drawbacks provide the motivation for re-considering the use of obfuscation and white-box cryptography in order to achieve the desired flexibility for applications running on Android phones. For achieving robustness against code-lifting attacks, we can bootstrap simple functionalities of the Android KeyStore. For example, we can compile a white-box such that it only works correctly if a valid RSA signature from the KeyStore is provided. Every time we wish to run our white-box program, we query the KeyStore and then the white-box verifies the signature. If the signature is successfully verified, the white-box performs further functionalities (decryption, payments, etc.). Alternatively, we can use the secret key material of the KeyStore for encoding the inputs to the white-box (as described in Sections 1.5 and

[7]In the case of Samsung Pay, where the producer of the payment application has control of the hardware this would be feasible

6). Here, every time we want to run a value x on the white-box, we first encode it via the RSA secret key of the KeyStore. Then our white-box uses the corresponding public key for decoding x. Many trusted components used within industrial products already support RSA-compatible signature schemes. Thus, we can assume that a white-box design based on MRS or traditional signatures will be viable in a very large amount of commercial mobile phones.

With both of the approaches described above, we obtain a white-box program which is bound to a unique Android device. The white-box can perform any complex functionality while the binding is achieved via a simple functionality, which can be performed by any KeyStore. As mentioned above, in the best case the KeyStore will be hardware based and in the worst case, it will be located in a software-based TEE. As mentioned in the Introduction, one can view this approach of using the white-box program in combination of a trusted hardware (or KeyStore) as a means to bootstrap the limited functionality of the hardware to facilitate the secure computation of a complex functionality.

B Additional cryptographic primitives

Definition 9. *A symmetric encryption scheme* SE *is a tuple of three algorithms (*$\mathsf{SE.KGen}$*,* $\mathsf{SE.Enc}$*,* $\mathsf{SE.Dec}$*) such that* $\mathsf{SE.KGen}$ *and* $\mathsf{SE.Enc}$ *are probabilistic polynomial-time algorithms (PPT),* $\mathsf{SE.Dec}$ *is a deterministic polynomial-time algorithm, and the algorithms have the following syntax:*

$$k \leftarrow_\$ \mathsf{SE.KGen}(1^\lambda)$$
$$c \leftarrow_\$ \mathsf{SE.Enc}(k, m)$$
$$m \leftarrow \mathsf{SE.Dec}(k, c).$$

Moreover, this scheme is correct*, if for all messages* $m \in \{0,1\}^*$*,*

$$\Pr[\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m] = 1$$

where the probability is over the randomness of Enc *and* $k \leftarrow_\$ \mathsf{KGen}(1^\lambda)$*.*

Definition 10. *A symmetric encryption scheme* $\mathsf{SE} = (\mathsf{SE.KGen}, \mathsf{SE.Enc}, \mathsf{SE.Dec})$ *is indistinguishable under chosen-ciphertext attacks (IND-CCA secure) if for all PPT adversaries* $\mathcal{A}$*, the advantage* $\left|\Pr\left[\mathsf{Exp}^{\mathit{IND\text{-}CCA}}_{\mathcal{A},\mathsf{SE}}(1^\lambda) = 1\right] - \frac{1}{2}\right|$ *is negligible.*

```
$\mathsf{Exp}^{IND\text{-}CCA}_{\mathcal{A},\mathsf{SE}}(1^\lambda)$
$k \leftarrow_\$ \mathsf{KGen}(1^\lambda)$
$b \leftarrow_\$ \{0,1\}$
$b^* \leftarrow_\$ \mathcal{A}^{\mathcal{O}_{\mathsf{Enc}},\mathcal{O}_{\mathsf{Dec}}}(1^\lambda)$
return $(b = b^*)$
```

```
$\mathcal{O}_{\mathsf{Enc}}(m)$
if $b = 0$
    $c \leftarrow_\$ \mathsf{SE.Enc}(k, m)$
else
    $c \leftarrow_\$ \mathsf{SE.Enc}(k, 0^{|m|})$
$T[c] \leftarrow m$
return $c$
```

```
$\mathcal{O}_{\mathsf{Dec}}(c)$
if $b = 0$
    $m \leftarrow \mathsf{SE.Dec}(k, c)$
if $b = 1$
    if $T[c]$ is defined
        $m \leftarrow T[c]$
    else
        $m \leftarrow \mathsf{SE.Dec}(k, c)$
return $m$
```

Next, we provide a definition for message-recoverable signatures, where we use a secret key for signing a message and a public key for verifying and *recovering* the message.

Definition 11. *A message-recoverable signature (MRS) scheme is a tuple of three algorithms* $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Rcvr})$ *such that* KGen *and* Sig *are probabilistic polynomial-time algorithms (PPT) and* Rcvr *is a deterministic polynomial-time algorithm, which have the following syntax:*

$$\begin{aligned}&(\mathsf{sk}, \mathsf{pk}) \leftarrow_{\$} \mathsf{KGen}(1^\lambda)\\ &\tilde{m} \leftarrow_{\$} \mathsf{Sig}(\mathsf{sk}, m)\\ &m/\bot \leftarrow \mathsf{Rcvr}(\mathsf{pk}, \tilde{m}).\end{aligned}$$

Moreover, this scheme is correct*, if for all messages* $m \in \{0,1\}^*$,

$$\Pr[\mathsf{Rcvr}(\mathsf{pk}, \mathsf{Sig}(\mathsf{sk}, m)) = m] = 1$$

where the probability is over the randomness of $(\mathsf{sk}, \mathsf{pk}) \leftarrow_{\$} \mathsf{KGen}(1^\lambda)$ *and* Sig.

Definition 12. *An MRS scheme is secure if for all PPT adversaries* $\mathcal{A}$*, their advantage* $\left|\Pr\left[\mathsf{Exp}^{\mathsf{integrity}}_{\mathcal{A},\mathsf{MRS}}(1^\lambda) = 1\right]\right|$ *is negligible.*

```
$\mathsf{Exp}^{\mathsf{integrity}}_{\mathcal{A},\mathsf{MRS}}(1^\lambda)$
$\mathcal{M} \leftarrow \emptyset$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow_{\$} \mathsf{KGen}(1^\lambda)$
$\tilde{m} \leftarrow_{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{Sig}}}(\mathsf{pk})$
$m \leftarrow \mathsf{Rcvr}(\mathsf{pk}, \tilde{m})$
if $m \notin \mathcal{M} \cup \{\bot\}$   return 1
else   return 0

$\mathcal{O}_{\mathsf{Sig}}(m)$
$\tilde{m} \leftarrow_{\$} \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{M} := \mathcal{M} \cup m$
return $\tilde{m}$
```

That is, the adversary wins if she is able to produce a signature $\tilde{m}$ *that recovers to a message* m *that was not queried to the signing oracle before.*

Let us recall the definition of puncturable PRFs.

Definition 13 (Puncturable PRF [43]). *Let* $l(\lambda)$ *and* $m(\lambda)$ *be the input and output lengths. A family of puncturable pseudorandom functions* $\mathsf{G} = \{\mathtt{PPRF}\}$ *is given by a triple of efficient functions (*Setup, Eval, Punc*), where* $\mathsf{Setup}(1^\lambda)$ *generates the key* K*, such that* $\mathtt{PPRF}$ *maps from* $\{0,1\}^{l(\lambda)}$ *to* $\{0,1\}^{m(\lambda)}$*;* $\mathsf{Eval}(K, x)$ *takes a key* K*, an input* x*, outputs* $\mathtt{PPRF}(K, x)$*;* $\mathsf{Punc}(K, x^*)$ *takes a key* K *and an input* x^**, outputs a punctured key* $K\{x^*\}$.

It satisfies the following conditions:

Functionality preserved over unpunctured points: *For all* x^* *and keys* K*, if* $K\{x^*\} = \mathsf{Punc}(K, x^*)$*, then for all* $x \neq x^*$, $\mathtt{PPRF}(K, x) = \mathtt{PPRF}(K\{x^*\}, x)$.

Pseudorandom on the punctured points: *For every input* x^**, the value of* F *on* x^* *is indistinguishable from random in the presence of the key punctured at* x^**. That is, the following two distributions are indistinguishable for every* x^**:*

$$(x^*, K\{x^*\}, G_K(x^*)) \text{ and } (x^*, K\{x^*\}, r^*),$$

where K *is output by* $\mathsf{Setup}(1^\lambda)$, $K\{x^*\}$ *is output by* $\mathsf{Punc}(K, x^*)$*, and* r^* *is uniform in* $\{0,1\}^{m(\lambda)}$.

Theorem 3 ([43]). *If one-way function exists, then for all length parameters* $l(\lambda)$, $m(\lambda)$*, there is a puncturable PRF family that maps from* $l(\lambda)$ *bits to* $m(\lambda)$ *bits.*

C TBO-Construction for user specific encryption

Construction 1 in Section 4 provides the desired security in scenarios where information is broadcast by a provider and only legitimate users should be able to decode such information. That is, all users who have registered are able to run the broadcast information through the token-generation algorithm and decode it with WB. The provider can encrypt the information using one symmetric key, which is embedded in the WB of all the users.

However, this construction does not provide the desired security in scenarios where the transmitted information should only be decoded by specific users. For instance in mobile-payment applications, we provide the users with a set of encrypted limited-use keys (LUKs) which the user stores in encrypted form and decrypts at the moment of performing a transaction. Here, only that user should be able to decrypt those keys. Therefore, we need to modify our construction, such that the ciphertexts are generated using an encryption key which is unique for each user. We do this by integrating a pseudorandom function (PRF) within our scheme. The PRF is run on the secret key ek and on the ID value corresponding to one unique user. Thus, we need to assume that for these use cases, the ID is securely transmitted to the provider along with the user's public key. Additionally, this ID needs to be stored securely in the trusted component of the provider and integrated within the token-generation process.

Construction 3. *Let F be any function of the syntax $F(x, y) = z$. Let $\mathsf{SE} = (\mathsf{SE.KGen}, \mathsf{SE.Enc}, \mathsf{SE.Dec})$ be a symmetric-encryption scheme. Let PRF be a pseudorandom function. Let $\mathsf{TBO} = (\mathsf{TBO.Obf}, \mathsf{TBO.InpGen})$ be a token-based obfuscation scheme.*

```
$C[K](w)$
    $c\|y\|\mathsf{ID} \leftarrow w$
    $k \leftarrow \mathsf{PRF}(K, \mathsf{ID})$
    $x \leftarrow \mathsf{SE.Dec}(k, c)$
    $z \leftarrow F(x, y)$
    return $z$

$\mathsf{Comp}[C](F)$
    $\mathsf{ek} \leftarrow\!\!\$\ \mathsf{SE.KGen}(1^n)$
    $\mathsf{MSK}, O \leftarrow\!\!\$\ \mathsf{TBO.Obf}(C[\mathsf{ek}])$
    $\mathsf{WB} \leftarrow O$
    $\mathsf{bk} \leftarrow \mathsf{MSK}$
    return $\mathsf{ek}, \mathsf{bk}, \mathsf{WB}$

$\mathsf{Encrypt}(\mathsf{ek}, x, \mathsf{ID})$
    $k \leftarrow \mathsf{PRF}(\mathsf{ek}, \mathsf{ID})$
    $c \leftarrow \mathsf{SE.Enc}(k, x)$
    return $c$

$\mathsf{Encode}[\mathsf{ID}](\mathsf{bk}, c, y)$
    $\mathsf{MSK} \leftarrow \mathsf{bk}$
    $w \leftarrow c\|y\|\mathsf{ID}$
    $\tilde{c} \leftarrow\!\!\$\ \mathsf{TBO.InpGen}(\mathsf{MSK}, w)$
    return $\tilde{c}$
```

Theorem 4. *If SE is a secure symmetric encryption scheme, PRF is a pseudorandom function and TBO is a secure token based obfuscation scheme, then Construction 3 is a secure GW-scheme with user-specific encryptions.*

This can be proved by simple extension to the proof of Theorem 1.

D MRS-like Construction for user specific encryptions

As with our constructions based on token-based obfuscation, we consider two classes of constructions for strong global white-boxes based on PPRFs. Namely our first construction based on the circuit $\mathit{\Gamma}$ is useful for cases where the values c correspond to information broadcasted to a number of users, such as for DRM programs. For applications such as mobile payments, we need to ensure that the values c can only be decrypted by a specific user. We recall that for mobile payment applications, c corresponds to an encrypted limited use key which, when decrypted, is used for generating a message authentication code.

We use the same approach as presented in Section 4 by integrating a KDF. All white-boxes will have the same KDF embedded on them and it will run on the key ek. During the encrypt process, the KDF will be called based on the user's ID and on ek and will be used for generating an encryption key. If an integrity check is passed, the white-box will run the KDF on ek and the ID to generate the corresponding decryption key.

$\mathit{\Gamma}_2[K_1, K_2, \mathsf{ek}, F](\mathsf{cert}, \tilde{c}, y)$

```
 1 :  $\psi, c_1, c_2 \leftarrow \mathsf{cert}$
 2 :  if $\mathtt{PRG}(c_2) \neq \mathtt{PRG}(\mathtt{PPRF}(K_2, \psi || c_1))$
 3 :    return $\bot$
 4 :  else
 5 :    $w \leftarrow \mathtt{PPRF}(K_1, \psi)$
 6 :    $\mathsf{pk}' \leftarrow c_1 \oplus w$
 7 :    $K_1' || K_2' || \tilde{\mathsf{ID}} \leftarrow \mathsf{pk}'$
 8 :    $\psi', c_1', c_2' \leftarrow \tilde{c}$
 9 :    if $\mathtt{PRG}(c_2') \neq \mathtt{PRG}(\mathtt{PPRF}(K_2', \psi' || c_1'))$
10 :      return $\bot$
11 :    else
12 :      $w' \leftarrow \mathtt{PPRF}(K_1', \psi')$
13 :      $c || \bar{\mathsf{ID}} \leftarrow c_1' \oplus w'$
14 :      $t, r \leftarrow c$
15 :      $\tilde{k} \leftarrow \mathtt{PPRF}(\mathsf{ek}, \tilde{\mathsf{ID}})$
16 :      $\bar{k} \leftarrow \mathtt{PPRF}(\mathsf{ek}, \bar{\mathsf{ID}})$
17 :      if $\mathtt{PRG}(\tilde{k}) \neq \mathtt{PRG}(\bar{k})$
18 :        return $\bot$
19 :      else
20 :        $v \leftarrow \mathtt{PPRF}(\bar{k}, r)$
21 :        $x \leftarrow t \oplus v$
22 :        return $F(x, y)$
```

The circuit describing our white-box program for this case is the same as the circuit $\mathit{\Gamma}$ described in the beginning of Section 6.1 with the following differences. On line 7, when parsing pk', there is an additional $\tilde{\mathsf{ID}}$ value appended to the end. Then, when recovering c on line 13, there is an additional $\bar{\mathsf{ID}}$ value appended to the end. The circuit then checks **if** $\mathtt{PRG}(\tilde{\mathsf{ID}}) \neq \mathtt{PRG}(\bar{\mathsf{ID}})$ **then** return $\bot$**else** $k \leftarrow \mathtt{PPRF}(\mathsf{ek}, \mathsf{ID})$. If the check goes through, the circuit then proceeds to line 15, but uses the derived key k instead of ek.

Construction 4. *Let the circuit $\mathit{\Gamma}_2$ be as described above. Let F be any puncturable function of the syntax $F(x, y) = z$. Let* $\mathtt{iO}$ *be an indistinguishability obfuscator. Let PRG be a pseudorandom generator and* $\mathtt{PPRF}$ *be a secure puncturable PRF.*

Theorem 5. *Let* $\mathtt{iO}$ *be a an indistinguishability obfuscator. Let PRG be a pseudorandom generator, and* $\mathtt{PPRF}$ *a secure puncturable pseudorandom function. Then Construction 4 is a privacy- and forgery-secure* sGW *scheme.*

Proof. The proof of Theorem 5 follows the same approach as that of Theorem 2. Recall that our forgery game has two alternative winning conditions for the adversary. Here we will prove that an adversary cannot win on either of the two. For proving the second winning condition, where an adversary tries to recover a value x^* from a chosen ciphertext c^*, we need to be

```
Comp(F)
1 : $(K_1, K_2) \leftarrow\$ \{0,1\}^\lambda$
2 : $\mathsf{rk} \leftarrow K_1||K_2$
3 : $\mathsf{ek} \leftarrow\$ \{0,1\}^\lambda$
4 : $\mathsf{WB} \leftarrow\$ \mathtt{iO}(\Gamma_2[K_1, K_2, \mathsf{ek}, F])$
5 : return $\mathsf{rk}, \mathsf{ek}, \mathsf{WB}$
```

```
Init(1^λ)
1 : $\tilde{\mathsf{ID}} \leftarrow\$ \{0,1\}^l$
2 : $(K'_1, K'_2) \leftarrow\$ \{0,1\}^\lambda$
3 : $K' \leftarrow K'_1||K'_2||\tilde{\mathsf{ID}}$
4 : $\mathsf{pk}' \leftarrow \mathtt{iO}(\mathtt{PPRF}(K'_1, \cdot)), \mathtt{iO}(\mathtt{PPRF}(K'_2, \cdot))||\tilde{\mathsf{ID}}$
5 : return $(K', \mathsf{pk}')$
```

```
Encrypt(ek, (x, ID))
1 : $k \leftarrow \mathtt{PPRF}(\mathsf{ek}, \mathsf{ID})$
2 : $r \leftarrow\$ \{0,1\}^n$
3 : $v \leftarrow \mathtt{PPRF}(k, r)$
4 : $t \leftarrow x \oplus v$
5 : $c \leftarrow (t, r)$
6 : return $c$
```

```
Enroll(rk, (pk'))
1 : $K_1||K_2 \leftarrow \mathsf{rk}$
2 : $\psi \leftarrow\$ \{0,1\}^\lambda$
3 : $w \leftarrow \mathtt{PPRF}(K_1, \psi)$
4 : $c_1 \leftarrow \mathsf{pk}' \oplus w$
5 : $c_2 \leftarrow \mathtt{PPRF}(K_2, \psi||c_1)$
6 : $\mathsf{cert} \leftarrow (\psi, c_1, c_2)$
7 : $\tilde{\mathsf{ID}} \leftarrow \mathsf{pk}'[l - |\mathsf{pk}'|]$
8 : return $\mathsf{cert}, \tilde{\mathsf{ID}}$
```

```
Encode(K'_1, K'_2, (c))
1 : $K'_1||K'_2||\tilde{\mathsf{ID}} \leftarrow K'$
2 : $\mathsf{ID} \leftarrow \tilde{\mathsf{ID}}$
3 : $\psi' \leftarrow\$ \{0,1\}^\lambda$
4 : $w' \leftarrow \mathtt{PPRF}(K'_1, \psi')$
5 : $c'_1 \leftarrow c||\tilde{\mathsf{ID}} \oplus w'$
6 : $c'_2 \leftarrow \mathtt{PPRF}(K'_2, \psi'||c'_1)$
7 : $\tilde{c} \leftarrow (\psi', c'_1, c'_2)$
8 : return $\tilde{c}$
```

specific about the function F. For this case, we will consider F to run a one-way function on x and then simply append y.

We will go through a series of hybrid which we describe next. The first hybrid corresponds to our unforgability experiment presented in Figure 4. Recall that in this game, the adversary tries to forge a valid value z, for a specific ciphertext c and context pair y. The adversary tries to forge such a value without accessing the hardware of an enrolled device.

- **Hybrid 0:** The attacker chooses a ciphertext c, a value y and the public key with ID ID^*. The attacker will try to forge a value $z*$ which should correspond to $z = F(\mathsf{Dec}(., c, \mathsf{ID}^*), y)$. The keys $K_1, K_2, \mathsf{ek}, K'$ are generated at random. Recall that the keys K_1, K_2, ek will be used for compiling the white-box program.
  The outputs of the Enroll, Init, Encrypt and Encode oracles are obtained in the same way as described for the algorithms in Construction 4 with the corresponding names. Note that during the game, the attacker will enroll the public key with ID ID^*, but the attacker will never query the encode oracle for the chosen value c *and* the public key with ID^*.
- **Hybrid 1:** The same as Hybrid 0, except that we will puncture the keys on the points defined as follows and change the program Γ_2 to a functionally equal one, which we denote Γ_2^*.
  1. Let $K'^* = {K'_1}^*||{K'_2}^*$ denote the key K' of the protected user.
  2. Let ψ'^* denote the value of ψ' used by the encode oracle when our user calls it on c^*.
  3. Let $w'^* \leftarrow \mathtt{PPRF}(K'_1, \psi'^*)$.

4. Let ${c'_1}^*$ be $c^* \oplus w'^*$.
5. Let ${c'_2}^* \leftarrow \mathtt{PPRF}(K'_2, \psi'^*||{c'_1}^*)$
6. Puncture ${K'_1}^*$ on ψ'^* and get ${K'_1}^*\{\psi'^*\}$.
7. Puncture ${K'_2}^*$ on $\psi'^*||{c'_1}^*$ and get ${K'_2}^*\{\psi'^*||{c'_1}^*\}$.
8. Let ψ^* denote the value of ψ used by the enroll oracle when our user calls it on K'^*.
9. Let w^* be $\mathtt{PPRF}(K_1, \psi^*)$.
10. Let ${c_1}^*$ be $\big(\mathtt{iO}(\mathtt{PPRF}({K'_1}^*\{\psi'^*\}, \cdot))||\mathtt{iO}(\mathtt{PPRF}({K'_2}^*\{\psi'^*||{c'_1}^*\}, \cdot))\big) \oplus w^*$.
11. Let ${c_2}^*$ be $\mathtt{PPRF}(K_2, \psi^*||{c_1}^*)$.
12. Puncture K_1 on ψ^* and get $K_1\{\psi^*\}$.
13. Puncture K_2 on $\psi^*||{c_1}^*$ and get $K_2\{\psi^*||{c_1}^*\}$.
14. Let ID^* denote the value of ID used when generating the challenge ct c^*.
15. Let k^* be $\mathtt{PPRF}(\mathsf{ek}, \mathsf{ID}^*)$.
16. Let c^* be $(x \oplus \mathtt{PPRF}(k, r)$.
17. Puncture ek on ID^* and get $\mathsf{ek}\{\mathsf{ID}^*\}$.
18. Finally, generate $\mathit{\Gamma}_2[K_1\{\psi^*\}, K_2\{\psi^*||{c_1}^*\}, {c_2}^*, {K'_1}^*\{\psi'^*\}, {K'_2}^*\{\psi'^*||{c'_1}^*\}, {c'_2}^*, c^*, \mathsf{ek}\{\mathsf{ID}^*\}, k^*, F]$ as in Figure 6. Note that all the values in the keys have been defined.

The circuits $\mathit{\Gamma}_2$ and $\mathit{\Gamma}_2^*$ in Hybrids 0 and 1 respectively are functional equivalent and this game hop reduces to $\mathtt{iO}$ security.

– **Hybrid 2:** the same as Hybrid 1 with the following exceptions:
  1. We replace c_2^* by $c_2^* \leftarrow_\$ \{0,1\}^\lambda$.
  2. We then calculate $h \leftarrow_\$ \mathtt{PRG}(c_2^*)$.
  3. We hard-code h in $\mathit{\Gamma}_2^*$ and replace $\mathtt{PRG}(c_2^*)$ by h on lines 2 and 4.

  This game hop reduces to PPRF- and $\mathtt{iO}$-security.
– **Hybrid 3:** the same as Hybrid 2 with the following exceptions:
  1. We replace ${c'_2}^*$ by ${c'_2}^* \leftarrow_\$ \{0,1\}^\lambda$.
  2. We then calculate $h' \leftarrow_\$ \mathtt{PRG}({c'_2}^*)$.
  3. We hard-code h' in $\mathit{\Gamma}_2^*$ and replace $\mathtt{PRG}({c'_2}^*)$ by h' on lines 13 and 15.

  This game hop reduces to PPRF- and $\mathtt{iO}$-security.
– **Hybrid 4:** the same as Hybrid 3 with the following exceptions:
  1. We replace k^* by $k^* \leftarrow_\$ \{0,1\}^\lambda$.
  2. We then calculate $d \leftarrow \mathtt{PRG}(k^*)$.
  3. We hard-code d in $\mathit{\Gamma}_2^*$ in line 27

  This game hop reduces to PPRF and iO-security.
– **Hybrid 5:** the same as Hybrid 4 with the following exceptions:
  1. $d \leftarrow_\$ \{0,1\}^{2\lambda}$.
  2. Now lines 30-32 will not be needed since line 27 will only be satisfied with negligible probability.

  This game hop reduces to PRG- and $\mathtt{iO}$-security.
– **Hybrid 6:** the same as Hybrid 4 with the following exceptions:
  1. After line 30, we check if $c = c^*$. If this is the case, we output $z^* = F(x^*, y) = f(x^*)||y$, else we proceed to line 31.

  This game hop reduces to OWF- and $\mathtt{iO}$-security.

□

```
$\Gamma_2^*[K_1\{\psi^*\}, K_2\{\psi^*||{c_1}^*\}, {c_2}^*, {K_1'}^*\{{\psi'}^*\}, {K_2'}^*\{{\psi'}^*||{c_1'}^*\}, {c_2'}^*, c^*, \mathsf{ek}\{r^*\}, \mathsf{ID}^*, k^*, c^*, z^* F](\mathsf{cert}, \tilde{c}, y)$

1 :  $\psi, c_1, c_2 \leftarrow \mathsf{cert}$
2 :  **if** $\psi||c_1 = \psi^*||{c_1}^*$ **and** $\mathtt{PRG}(c_2) = \mathtt{PRG}({c_2}^*)$
3 :    $\mathsf{pk}' \leftarrow {K_1'}^*\{{\psi'}^*\}, {K_2'}^*\{{\psi'}^*||{c_1'}^*\}$
4 :  **else if** $\psi = \psi^*$ **and** $(c_1 \neq {c_1}^*$ **or** $\mathtt{PRG}(c_2) \neq \mathtt{PRG}({c_2}^*))$
5 :    return $\perp$
6 :  **else if** $\mathtt{PRG}(c_2) \neq \mathtt{PRG}(\mathtt{PPRF}(K_2\{\psi^*||{c_1}^*\}, \psi||c_1))$
7 :    return $\perp$
8 :  **else**
9 :    $w \leftarrow \mathtt{PPRF}(K_1\{\psi^*\}, \psi)$
10 :    $\mathsf{pk}' \leftarrow c_1 \oplus w$
11 :    $K_1'||K_2'||\tilde{\mathsf{ID}} \leftarrow \mathsf{pk}'$
12 :    $\psi', c_1', c_2' \leftarrow \tilde{c}$
13 :    **if** $\psi'||c_1' = {\psi'}^*||{c_1'}^*$ **and** $\mathtt{PRG}(c_2') = \mathtt{PRG}({c_2'}^*)$
14 :      $c \leftarrow c^*$
15 :    **else if** $\psi' = {\psi'}^*$ **and** $(c_1' \neq {c_1'}^*$ **or** $\mathtt{PRG}(c_2') \neq \mathtt{PRG}({c_2'}^*))$
16 :      return $\perp$
17 :    **else if** $\mathtt{PRG}(c_2') \neq \mathtt{PRG}(\mathtt{PPRF}(K_2', \psi'||c_1'))$
18 :      return $\perp$
19 :    **else**
20 :      $w' \leftarrow \mathtt{PPRF}(K_1', \psi')$
21 :      $c||\bar{\mathsf{ID}} \leftarrow c_1' \oplus w'$
22 :      $t, r \leftarrow c$
23 :      **if** $\tilde{\mathsf{ID}} = \mathsf{ID}^*$
24 :        $\tilde{k} \leftarrow k^*$
25 :      **else**
26 :        $\tilde{k} \leftarrow \mathtt{PPRF}(\mathsf{ek}\{\mathsf{ID}^*\}, \tilde{\mathsf{ID}})$
27 :      $\bar{k} \leftarrow \mathtt{PPRF}(\mathsf{ek}\{\mathsf{ID}^*\}, \bar{\mathsf{ID}})$
28 :      **if** $\mathtt{PRG}(\tilde{k}) \neq \mathtt{PRG}(\bar{k})$
29 :        return $\perp$
30 :      **else**
31 :        $v \leftarrow \mathtt{PPRF}(k, r)$
32 :        $x \leftarrow t \oplus v$
33 :        return $F(x, y)$
```

Fig. 6: Γ_2^* in Hybrid 1.