Security Reductions for White-Box Key-Storage in Mobile Payments

Estuardo Alpirez Bock[1], Chris Brzuska[1], Marc Fischlin[2], Christian Janson[2], and Wil Michiels[3,4]

[1] Aalto University, Finland
{estuardo.alpirezbock,chris.brzuska}@aalto.fi
[2] Technische Universität Darmstadt, Darmstadt, Germany
{marc.fischlin,christian.janson}@cryptoplexity.de
[3] Technische Universiteit Eindhoven, Netherlands
[4] NXP Semiconductors, Netherlands
wil.michiels@nxp.com

Abstract. The goal of white-box cryptography is to provide security even when the cryptographic implementation is executed in adversarially controlled environments. White-box implementations nowadays appear in commercial products such as mobile payment applications, e.g., those certified by Mastercard. Interestingly, there, white-box cryptography is championed as a tool for secure storage of payment tokens, and importantly, the white-boxed storage functionality is bound to a hardware functionality to prevent code-lifting attacks.

In this paper, we show that the approach of using hardware-binding and obfuscation for secure storage is conceptually sound. Following security specifications by Mastercard and also EMVCo, we first define security for a *white-box key derivation functions* (WKDF) that is bound to a hardware functionality. WKDFs with hardware-binding model a secure storage functionality, as the WKDFs in turn can be used to derive encryption keys for secure storage. We then provide a proof-of-concept construction of WKDFs based on pseudorandom functions (PRF) and obfuscation. To show that our use of cryptographic primitives is sound, we perform a cryptographic analysis and reduce the security of our WKDF to the cryptographic assumptions of indistinguishability obfuscation and PRF-security. The hardware-functionality that our WKDF is bound to is a PRF-like functionality. Obfuscation helps us to hide the secret key used for the verification, essentially emulating a signature functionality as is provided by the Android key store.

We rigorously define the required security properties of a *hardware-bound white-box payment application* (WPAY) for generating and encrypting valid payment requests. We construct a WPAY, which uses a WKDF as a secure building block. We thereby show that a WKDF can be securely combined with *any* secure symmetric encryption scheme, including those based on standard ciphers such as AES.

Keywords: White-box cryptography · Key derivation function · Hardware-binding · Payment application

1 Introduction

Near-field communication (NFC) protocols have opened up new possibilities for mobile payment applications, such as those offered by Mastercard, Visa, or Google wallet [37]. Traditionally, the NFC traffic was processed by a secure hardware component in mobile devices, that performed cryptographic operations. In 2015, Android 4.4 introduced Host Card Emulation (HCE), which allows the application processor of a mobile device to use the NFC communication, too. In this case, the cryptographic functions of mobile applications are implemented *software-only*, increasing flexibility and device coverage of the applications since no secure hardware element is required. However, implementing cryptographic functions in software leads to new attack vectors.

White-box cryptography. One of the core cryptographic protection technologies for HCE (as listed by the Smart Card Alliance Mobile & NFC Council [37]) is the use of *white-box cryptography*. Cryptography in the white-box attack model was introduced by Chow, Eisen, Johnson, and van Oorschot in 2002 (CEJO [19,18]). In the white-box attack scenario, an adversary has access to the program code and the complete execution environment of a cryptographic implementation, and the goal of white-box cryptography is to remain secure despite such strong attack capabilities. Unlike in the Digital Rights Management scenario considered by CEJO where the user is considered as an adversary, in an HCE context, the goal of white-box cryptography is protect an honest user against attacks performed on their device, e.g., by malware that can observe program code and executions.

Commercial payment applications. Payment applications need to store secret information such as transaction tokens that are decrypted when a transaction is performed. In the absence of a secure element, the tokens are stored in insecure memory and likewise, the decryption operations are performed by an insecure CPU. Thus, to protect against adversaries that use their access to the storage/CPU to extract secret information and perform payment transactions on their own, over the past years, white-box cryptography has been broadly adopted by those offering commercial payment applications. The Mastercard security guidelines for payment applications, e.g., make the use of white-box cryptography *mandatory* for implementing storage protection in order to achieve an advanced security level (see *Local Database Encryption*, Chapter 5 in [31]). Similarly, EMVCo suggests the use of white-box cryptography in their requirements documentation [24] for EMV mobile payment.

For a successful payment, the user's device needs to be close to an NFC reader. An attacker on a user's device can thus only alter a payment a user aims to make, but cannot make payments independently at readers of their choice,[5] unless the attacker gains independence from the user's device. The attacker could

[5] We discuss relay attacks later.

gain independence by (a) extracting the key, or (b) performing a code-lifting attack [38]. Thus, white-box cryptography in commercial payment applications needs to achieve *hardware-binding* (also see [20,36,8] for discussions of usefulness). As a consequence, commercial applications implement white-box cryptography with a hardware anchor, essentially reaching a middle-ground between software-only and hardware-only security for cryptographic implementations.

In [14] Alpirez Bock, Amadori, Brzuska and Michiels (AABM) discuss extensively the usefulness of hardware-binding for white-box programs. They explain that hardware-binding seems to be the right mitigation technique against code-lifting attacks for white-box programs deployed on real-life applications. The authors propose to focus on hardware-binding as opposed to other techniques which are popular in the white-box literature, but seem rather theoretical or unrelated to the security of payment applications. In particular, they define a security notion for white-box encryption programs with hardware-binding.

Hardware-binding on Android. The Mastercard guidelines (see Chapter 5 in [31]) recommend, to the very least, to use a unique device fingerprint for device identification. The EMVCo documentation [24] recommends to use hardware features to bind the operation of software on a particular device. For instance, Android allows to perform checks on identifiers such as the hardware serial number, the ESN (electronic serial number) or IMEI (international mobile equipment identity) of the device via its `Build` and `TelephonyManager` classes [3,5]. This technique helps to mitigate code-lifting attacks as long as the value remains secret and/or interception of this value between hardware and software can actually be prevented. For an advanced security level, however, the guidelines suggest the use of the functionalities of the Android Key Store. The Android Key Store, e.g., implements RSA signatures, and relies on whatever secure hardware features the Android device provides. Signatures are a more useful binding functionality than single identifier values, since for each new input, they provide a different output.

Conceptual validation. In this paper, we show that the wide-spread practical approach for building secure payment applications based on white-box cryptography is conceptually sound. We split our study into two parts:

- a hardware-bound white-box key derivation function (WKDF) which provides (1) hardware-binding and (2) secure storage;
- a secure payment application that performs (3) symmetric encryption of data on top of the WKDF.

Note that the Mastercard guidelines merely specify best practices but omit a design blueprint. Our goal is to explicate how exactly a sound design shall proceed, and what security properties the underlying primitives should obey.

Hardware-bound white-box key derivation function (WKDF). Our WKDF notion builds on top of a standard (black-box) key derivation function

(KDF). We here consider a lightweight notion for the KDF that takes uniformly random keys and a second, non-random value as input, and returns pseudorandom keys of fixed length. We therefore use the terms KDF and pseudorandom function (PRF) interchangeably, abstracting away additional KDF features such as varying output lengths (cf. Krawczyk [29]). We introduce the IND-WKDF security notion for WKDFs that models the previously discussed white-box attack scenario. I.e., the adversary is given full access to the white-box implementation of the WKDF as well as limited access to the hardware. If the adversary uses its hardware access, the adversary is able to evaluate the WKDF, but if the adversary has no access to the relevant hardware values, e.g., carrying out a code-lifting attack, then the adversary learns nothing about the WKDF values, which is modeled by a real-or-random oracle for derived keys.

Existing security notions for white-box cryptography. Outside of payment applications, Delerablée, Lepoint, Paillier, and Rivain (DLPR [22]) defined the meanwhile popular notion of *incompressibility* (for constructions and definitional refinements see [22,15,16,11,25,13,30]). Here, an adversary shall not be able to compress a cryptographic program without losing part of its functionality. Incompressibility seems unrelated to achieving the goal of real-or-random key indistinguishability for derived keys, as we aim for the WKDF in our application. Additionally, DLPR define *traceability* which is a security notion intended to trace malicious users (typically in a DRM setting) if they illegally share their white-box program with others. Traceability is a helpful to mitigate code-lifting and re-distribution attacks in the case of malicious users, but not helpful to protect honest users from adversaries copying and misusing their software.

Finally, DPLR also discuss one-wayness and security against key extraction, a baseline security property for white-box cryptography, which are both implied by our IND-WKDF notion for WKDF. Namely, if an adversary can extract the key, then the adversary can evaluate the KDF on all points itself and thereby distinguish derived keys from random keys. Similarly, an adversary can use an inversion algorithm to distinguish real derived keys from random values. Thus, a IND-WKDF-secure WKDF also resists key extraction and inversion attacks.

Secure payment application. We introduce a secure hardware-bound payment application scheme (WPAY). Its basic functionality is to encrypt and to authenticate valid payment requests to a server. We model validity by a predicate that acts as filter function. E.g., the filter could only allow for certain date ranges or limits the upper bound on the payment, while the server generically accepts payments of arbitrary amounts and ranges.

Our security notion IND-WPAY gives the adversary the white-box payment application WPAY. The adversary can query a hardware oracle that provides them with the necessary hardware values to generate a request using WPAY. This models that the adversary can observe (and interfere with) honest user evaluations. As soon as the adversary loses access to the hardware, confidentiality

and integrity of the user requests should hold. IND-WPAY models both properties via an indistinguishability game. Note that IND-WPAY captures code-lifting attacks. Namely, the adversary has access to WPAY throughout the experiment, but only limited access to the hardware. IND-WPAY models that in the absence of the hardware, no valid requests can be generated, even given WPAY.

Constructions. To instantiate our approach for building a WKDF, we first need to specify a hardware functionality. One idea could be to rely on a signature functionality as provided for example by the Android Key Store. I.e., WKDF would send a request to the hardware, the hardware signs it, and WKDF then verifies the signature with the public verification key. But we need to (1) hide the software-related key of our WKDF and (2) make it inseparable from the verification algorithm that checks hardware values, which are both achieved by applying indistinguishability obfuscation techniques. This, in turn, forces us to use puncturable primitives for the security reduction to work. One option could thus be to use the puncturable signature scheme by Bellare, Stepanovs and Waters [10] which, notably, itself is based on indistinguishability obfuscation. To avoid this double form of obfuscation for the construction, one layer for the puncturable signature scheme and one for the hiding and binding of our KDF key, we instead use a faster symmetric-key primitive in form of a (puncturable) PRF (essentially as a message authentication code). This puncturable PRF is obfuscated once within the hardware-linked KDF construction to ensure the required security.

Hence, we build a WKDF and prove its IND-WKDF security, following techniques by Sahai and Waters [35]. This construction assumes puncturable PRFs (which are equivalent to one-way functions) and indistinguishability obfuscation. Given and IND-WKDF-secure WKDF we then prove that another layer of indistinguishability obfuscation can be used to bind the WKDF to an arbitrary secure symmetric encryption scheme and a filter function to obtain an IND-WPAY-secure white-box payment application WPAY.

Discussion and limitations. Note that our constructions are conceptual validations and not practically efficient due to the tremendous inefficiency of indistinguishability obfuscation (see [6]). In practice, the obfuscation needs to be implemented by a mix of efficient obfuscation techniques, combined with practical white-box techniques, e.g. [27]. Thus, our work does not allow to immediately bypass the difficulty of building white-box implementations —as apparent in the past white-box competitions [23,21], where only three design candidates submitted towards the end of the second competition remained unbroken— in practice. However, our theoretical feasibility result allows us to conclude that secure white-box implementations based on strong cryptographic assumptions —indistinguishability obfuscation is not yet a mature cryptographic primitive— are indeed possible. Our results not only affirm that building secure white-box cryptography is possible, but they also explain *how* such a secure white-box implementation can be designed.

As efficiency of indistinguishability obfuscation has not yet reached reasonable levels, let us now discuss security and efficiency of current practical white-box implementations. E.g., the winner of the first white-box competition [23] had a binary size of 17MB and needed 0.37 seconds for an encryption which is reasonably close to practical needs. It was broken eventually, but resisted key extraction attacks for up to 28 days [32]. This temporary robustness turns out to be useful. Namely, in practice, the goal is to maintain a complexity gap between the effort of the attacker and the effort of the designer. As software can be updated in a regular interval, one can achieve a reasonable practical security level by replace a white-box implementation, in each update, by a newer generation.

Two considerations are important to be taken into account: (1) The white-box implementation should not be susceptible to (variants of) automated attacks, since these can be implemented with little effort, see [1]. (2) Reverse-engineering efforts against previous generations of white-box implementations shall not help the attacker against the new generation of the white-box implementation. I.e., the designer needs to come up with a paradigm that allows to systematically inject a certain amount of creativity into the system that needs to be reverse-engineered anew each time. For example, if security requires bootstrapping security from white-boxing another cryptographic primitive such as a PRF, then one can use a different PRF each time, harvesting the large cryptographic research of PRF constructions. In our model, we bind each payment application to a fresh hardware sub key, derived from a main hardware key, and we allow the adversary to see other derived hardware keys. This models that potentially, earlier construction might have been broken and might have revealed the derived hardware key in use. An alternative is to directly bind to a signature functionality so that revealing the verification algorithm does not constitute an attack vector.

Note that our models do not consider plain relay attacks [28,26] where the adversary forwards the intended communication without altering it. These attacks need to be prevented by other means, e.g., via distance-bounding protocols [7], or at least mitigated via heuristics such as location correlation between phone and NFC reader [12]. Note however that we of course capture attacks where, say, the adversary modifies user requests or tries to create new requests by himself.

Finally, we remark that in our models we consider an adversary who attacks the application of an honest user. The adversary either tries to break security by extracting secret information from the application, or by code-lifting it and running it on a separate device. Our security notions model an adversary who obtains the white-box program but, if the program is securely implemented, the adversary can only run the program when using an oracle simulating part of the hardware of the user. We recall that when considering DRM applications on the other hand, the white-box definitions model an adversary who gets full access to a cryptographic program and to the device running that program (this being the adversary's own device) [19,18].

2 Preliminaries and Notation

By $a \leftarrow A(x)$, we denote the execution of a deterministic algorithm A on input x and the assignment of the output to a, while $a \leftarrow\!\!\$\, A(x)$ denotes the execution of a randomized algorithm and the assignment of the output to a. We denote by $:=$ the process of initializing a set, e.g. $X := \emptyset$. By $x \leftarrow\!\!\$\, X$ we denote the process of randomly and uniformly sampling an element x from a given set X. Slightly abusing notation, we also use $x \leftarrow\!\!\$\, X$ to denote the sampling of x according to probability distribution X. We then denote the probability that the event $E(x)$ happens by $\Pr_{x \leftarrow\!\!\$\, X}[E(x)]$ or sometimes simply $\Pr[E(x)]$. We write oracles as superscript to the adversary $\mathcal{A}^{\mathsf{O}}$. In cases when an adversary is granted access to a larger number of oracles, we write oracles also as subscript to the adversary $\mathcal{A}^{\mathsf{O}_1,\mathsf{O}_2}_{\mathsf{O}_3,\mathsf{O}_4}$. PPT denotes probabilistic polynomial-time and $\mathsf{poly}(n)$ is an unspecified polynomial in the security parameter. Note that all algorithms receive the security parameter 1^n in unary notation as input implicitly. We write it explicitly only occasionally for clarity.

We now review useful definitions, starting with nonce-based encryption, see Rogaway [34].

Definition 1 (Symmetric Encryption). *A* nonce-based symmetric encryption scheme SE *consists of a pair of deterministic polynomial-time algorithms* $(\mathtt{Enc}, \mathtt{Dec})$ *with the syntax* $c \leftarrow \mathtt{Enc}(k, m, nc)$ *and* $m/\perp \leftarrow \mathtt{Dec}(k, c, nc)$*. The algorithm* $\mathtt{Enc}$ *takes as input a randomly generated key* k *of length* n*, a nonce* nc*, a message* m*, and outputs a ciphertext* c*.* $\mathtt{Dec}$ *takes as input a randomly generated key* k *of length* n*, a nonce* nc*, a ciphertext* c*, and outputs either a message* m *or an error symbol* $\perp$*. Moreover, the encryption scheme* SE *satisfies* correctness*, if for all nonces* $nc \in \{0,1\}^n$ *and for all messages* $m \in \{0,1\}^*$*,*

$$\Pr[\mathtt{Dec}(k_{\mathsf{SE}}, \mathtt{Enc}(k_{\mathsf{SE}}, m, nc), nc) = m] = 1$$

where the probability is over sampling k*.*

$\mathsf{Exp}^{\mathsf{AE}}_{\mathsf{SE},\mathcal{A}}(1^n)$	$\mathsf{OENC}(m_0, m_1)$	$\mathsf{ODEC}(nc, c)$
$b \leftarrow\!\!\$\, \{0,1\}$	**assert** $\lvert m_0\rvert = \lvert m_1\rvert$	**assert** $c \notin C$
$k_{\mathsf{SE}} \leftarrow\!\!\$\, \{0,1\}^n$	$nc \leftarrow\!\!\$\, \{0,1\}^n$	**if** $b = 1$ **then**
$b' \leftarrow\!\!\$\, \mathcal{A}^{\mathsf{OENC},\mathsf{ODEC}}(1^n)$	$c \leftarrow \mathtt{Enc}(k_{\mathsf{SE}}, m_b, nc)$	**return** $\perp$
return $(b' = b)$	$C := C \cup \{c\}$	**else**
	return (nc, c)	**return** $m \leftarrow \mathtt{Dec}(k_{\mathsf{SE}}, c, nc)$

Fig. 1. The $\mathsf{Exp}^{\mathsf{AE}}_{\mathsf{SE},\mathcal{A}}(1^n)$ security game.

Definition 2 and Figure 1 specify the security of an *authenticated encryption scheme* [9,33]. Here, the adversary is provided with a left-or-right encryption

oracle and a decryption oracle where it can submit arbitrary ciphertexts except for challenge ciphertexts obtained from the encryption oracle. If $b = 0$, the decryption oracle is functional. If $b = 1$, the decryption oracle always returns $\perp$ which models ciphertext integrity. In the security game, we use **assert** as a shorthand to say that if the **assert** condition is violated, then the oracle returns an error symbol $\perp$.

Definition 2 (Authenticated Encryption). *A* nonce-based symmetric encryption scheme $\mathsf{SE} = (\mathtt{Enc}, \mathtt{Dec})$ *is called an* authenticated encryption scheme *or* AE*-secure if all PPT adversaries* $\mathcal{A}$ *have negligible distinguishing advantage in the game* $\mathsf{Exp}^{\mathsf{AE}}_{\mathsf{SE},\mathcal{A}}(1^n)$*, specified in Figure 1.*

Note that we demand the authenticated encryption scheme to be deterministic because we will later execute the algorithm in an untrusted environment and cannot count on strong randomness. This, in turn, implies that we cannot allow the adversary to re-use any of the previous queries (m, nc), or else it would be easy to determine b from two queries (m_0, m_1, nc) and (m_0, m_1', nc).

We provide a formal definition of a key derivation function that produces pseudorandom keys. Note that our definition corresponds to a PRF, i.e., it is highly simplified compared to the framework of Krawczyk [29]. In our definition a key derivation function takes as input a key $k_{\mathtt{kdf}}$, a context string e as well as the security parameter 1^n. In comparison to Krawczyk's definition, we simplify the presentation and omit the details of the smoothing step turning raw key material into random strings, the salting, and the length parameter, assuming that the key $k_{\mathtt{kdf}}$ is already appropriate and the length of the returned key is equal to $|k_{\mathtt{kdf}}|$.

Definition 3 (Key Derivation Function). *A* KDF*scheme consists of a randomized key generation algorithm* $\mathtt{Kgen}$ *and a* key derivation function $\mathtt{KDF}$ *that is a deterministic algorithm that takes as input a key* $k_{\mathtt{kdf}} \leftarrow\!\!\$\ \mathtt{Kgen}(1^n)$ *and a context string* e*. The* $\mathtt{KDF}$ *returns a key* $\hat{k}$ *of length* $|k_{\mathtt{kdf}}|$*.*

Definition 4 (IND-KDF-security). *A key derivation function* $\mathtt{KDF}$ *is said to be* $\mathsf{IND\text{-}KDF}$*-secure if all PPT adversaries* $\mathcal{A}$ *have negligible distinguishing advantage in game* $\mathsf{Exp}^{\mathsf{IND\text{-}KDF}}_{\mathsf{KDF},\mathcal{A}}(1^n)$*, specified in Figure 2.*

Next we present the definition of a length-doubling pseudorandom generator.

Definition 5 (Pseudorandom Generator). *A deterministic, polynomial-time computable function* $\mathtt{PRG} : \{0,1\}^* \to \{0,1\}^*$ *is a pseudorandom generator if:*

- ***Length-expansion:*** *For all* $x \in \{0,1\}^*$*,* $|\mathtt{PRG}(x)| = 2\,|x|$*.*
- ***Pseudorandomness:*** *For all PPT* $\mathcal{A}$*,* $\mathsf{Adv}_{\mathtt{PRG},\mathcal{A}}(n) :=$

$$\left|\Pr_{x \leftarrow\!\!\$\ \{0,1\}^n}[\mathcal{A}(\mathtt{PRG}(x)) = 1] - \Pr_{z \leftarrow\!\!\$\ \{0,1\}^{2n}}[\mathcal{A}(z) = 1]\right|$$

is negligible in n*.*

We define pseudorandom functions with identical input, output and key length.

Definition 6 (Pseudorandom Function). *A deterministic, polynomial-time computable function* PRF*, such that* $\mathtt{PRF} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ *for all* $n \in \mathbb{N}$*, is a pseudorandom function if for all PPT* $\mathcal{A}$*,* $\mathsf{Adv}_{\mathcal{A},\mathtt{PRF}}(n) :=$

$$\left|\Pr_{k \leftarrow\$ \{0,1\}^n}\left[\mathcal{A}^{\mathtt{PRF}(k,\cdot)}(1^n) = 1\right] - \Pr_{F \leftarrow\$ \{G:\{0,1\}^n \to \{0,1\}^n\}}\left[\mathcal{A}^{F(\cdot)}(1^n) = 1\right]\right|$$

is negligible in n*.*

Puncturable PRFs (PPRF) were introduced by Boneh and Waters [17]. PPRFs have a punctured key which allows to evaluate the PPRF on all inputs, except for one where the function still looks random.

Definition 7 (PPRF). *A puncturable pseudorandom function scheme* PPRF *consists of a triple* $(\mathtt{PPRF}, \mathtt{Punct}, \mathtt{Eval})$*, which are defined as follows:*

- $\mathtt{PPRF}(k, x)$: *This is a standard PRF evaluation algorithm. As before, this deterministic polynomial-time algorithm takes as input a key* k *and input* x*, both of length* n *and returns a value* y *of length* n*.*
- $\mathtt{Punct}(k, z)$: *This PPT algorithm takes as input a key* $k \in \{0,1\}^n$ *and an input value* $z \in \{0,1\}^n$*. It outputs a punctured key* $k_z \leftarrow\$ \mathtt{Punct}(k, z)$*.*
- $\mathtt{Eval}(k_z, x)$: *This* deterministic polynomial-time *algorithm takes as input a punctured key* k_z *and some input* $x \in \{0,1\}^n$ *and returns* $\perp$ *if* $x = z$*, and a value* $y \in \{0,1\}^n$ *otherwise.*

A puncturable PRF *is said to be* correct*, if for all security parameter* n*, all* $k \in \{0,1\}^n$*, every value* $z \in \{0,1\}^n$ *and all* $x \in \{0,1\}^n$*,* $x \neq z$*, it holds that*

$$\Pr[\mathtt{Eval}(\mathtt{Punct}(k, z), x) = \mathtt{PPRF}(k, x)] = 1.$$

$\underline{\mathsf{Exp}^{\mathsf{IND\text{-}KDF}}_{\mathsf{KDF},\mathcal{A}}(1^n)}$

$b \leftarrow\$ \{0,1\}$
$\mathcal{Q} := \emptyset$
$k_{\mathtt{kdf}} \leftarrow\$ \mathtt{Kgen}(1^n)$
$b' \leftarrow\$ \mathcal{A}^{\mathsf{OKDF}}(1^n)$
if $b' = b$
  return 1
else
  return 0

$\underline{\mathsf{OKDF}(e)}$

if $e \notin \mathcal{Q}$
  $\mathcal{Q} := \mathcal{Q} \cup \{e\}$
  if $b = 1$
    $\hat{k} \leftarrow \mathtt{KDF}(k_{\mathtt{kdf}}, e)$
  else
    $\hat{k} \leftarrow\$ \{0,1\}^n$
  return $\hat{k}$
else
  return $\perp$

Fig. 2. $\mathsf{Exp}^{\mathsf{IND\text{-}KDF}}_{\mathsf{KDF},\mathcal{A}}(1^n)$ security game

$\underline{\mathsf{Exp}^{\mathsf{IND\text{-}PPRF}}_{\mathsf{PPRF},\mathcal{A}}(1^n)}$

$b \leftarrow\$ \{0,1\}$
$k \leftarrow\$ \{0,1\}^n$
$(z, \mathsf{state}) \leftarrow\$ \mathcal{A}(1^n)$
$k_z \leftarrow\$ \mathtt{Punct}(k, z)$
if $b = 1$ **then**
  $y \leftarrow \mathtt{PPRF}(k, z)$
else $y \leftarrow\$ \{0,1\}^n$
$b' \leftarrow\$ \mathcal{A}(k_z, y, \mathsf{state})$
return $(b' = b)$

Fig. 3. IND-PPRF security game

PPRF security requires that the PPRF value on k and z is indistinguishable from random, even when given the punctured key k_z. Note that for our purposes, we only rely on security for *random* inputs rather than adversarially chosen ones, i.e., we use a less powerful assumption which makes our result stronger.

Definition 8 (IND-PPRF-security). *A* PPRF *scheme is said to be* IND-PPRF-secure *if all probabilistic polynomial-time adversaries $\mathcal{A}$ have negligible distinguishing advantage in the* IND-PPRF *game defined in Figure 3.*

An indistinguishability obfuscator (iO) ensures that the obfuscation of any two functionally equivalent programs (i.e. circuits) are computationally indistinguishable. In the following definition, a distinguisher $\mathcal{D}$ is an adversary that aims at identifying which of the two programs has been obfuscated.

Definition 9 (Admissible Circuit Sampler). *Let p be a polynomial. A PPT algorithm $\mathcal{S}$ is called a* p-admissible circuit sampler *if*

$$1 - \Pr_{(\mathsf{C}_0,\mathsf{C}_1) \leftarrow\!\!\$\, \mathcal{S}(1^n)}[\forall x \in \{0,1\}^n\ \mathsf{C}_0(x) = \mathsf{C}_1(x)]$$

is negligible in n and for all $n \in \mathbb{N}$ and all pairs $(\mathsf{C}_0, \mathsf{C}_1)$ in the range of $\mathcal{A}(1^n)$, it holds that the size of C_0 and the size of C_1 is upper bounded by $p(n)$.

Definition 10 (Indistinguishability Obfuscator). *A PPT algorithm* iO*, parameterized by a polynomial p, is called an* indistinguishability obfuscator *if for any p-admissible circuit sampler $\mathcal{S}$ the following conditions are satisfied:*

Correctness. *For all circuits* C *and for all inputs x to the circuit,*

$$\Pr[\mathsf{C}'(x) = \mathsf{C}(x) : \mathsf{C}' \leftarrow\!\!\$\, \mathtt{iO}(\mathsf{C})] = 1,$$

where the probability is over the randomness of iO*.*

Security. *For all p-admissible $\mathcal{S}$ and any PPT distinguisher $\mathcal{D}$, the following distinguishing advantage is negligible:*

$$\left|\Pr_{(\mathsf{C}_0\mathsf{C}_1) \leftarrow\!\!\$\, \mathcal{S}}[\mathcal{D}(\mathtt{iO}(\mathsf{C}_0)) = 1] - \Pr_{(\mathsf{C}_0\mathsf{C}_1) \leftarrow\!\!\$\, \mathcal{S}}[\mathcal{D}(\mathtt{iO}(\mathsf{C}_1)) = 1]\right| \leq \mathsf{negl}(n)\,,$$

where the probabilities are over the randomness of the algorithms.

When obfuscating cryptographic algorithms with keys it is often convenient to use the notation $\mathsf{C}[k](x)$ to denote the circuit with fixed encoded key(s) k and variable input x.

3 Hardware-Bound White-box Key Derivation Function

In this section, we first introduce our notion of a hardware module and explain how we instantiate it in our setting. Then, we provide the syntax and security notion for a hardware-bound key derivation function, present our construction and provide a security reduction to indistinguishability obfuscation and PPRFs.

3.1 Hardware Module

A schematic overview over the hardware module functionalities executed on the secure hardware is given in Figure 4. Namely, a hardware module comes with a key generation algorithm that generates the hardware main key $k_{\mathtt{HWm}}$.

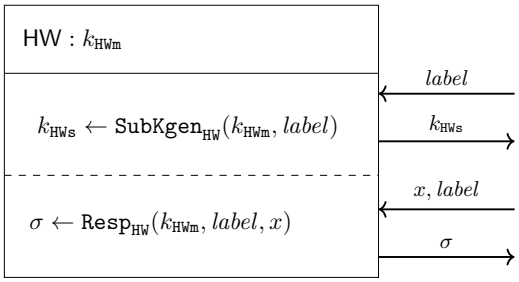


Fig. 4. Functionalities of the hardware module performed in the hardware. The `Check` operation is performed by the software program corresponding to the *label*.

This key generation algorithm is run at the manufacturer of the hardware and thus not depicted in Figure 4. The secure hardware allows to export a sub-key via querying the secure hardware with a *label*. The hardware then runs the *sub-key generation algorithm* $\mathtt{SubKgen}_{\mathtt{HW}}$ on $k_{\mathtt{HWm}}$ and *label* and returns the resulting sub-key $k_{\mathtt{HWs}}$. In addition, the secure hardware can be queried with a pair $(x, label)$. The hardware, then, uses the algorithm $\mathtt{Resp}_{\mathtt{HW}}$ to generate a PRF/MAC value σ for x under $k_{\mathtt{HWm}}$. In order to avoid storing $k_{\mathtt{HWs}}$ for all values *label*, the hardware re-derives $k_{\mathtt{HWs}}$ anew each time $\mathtt{Resp}_{\mathtt{HW}}$ is called. The PRF/MAC value can be checked outside of the secure hardware by running $\mathtt{Check}_{\mathtt{SW}}$ on $k_{\mathtt{HWs}}$ and the pair (x, σ).

Remarks. In this paper, we assume that the hardware module is secure and, as the only part of the device, not subject to white-box attacks. Even more, we assume that the hardware looks like a secure black-box to the white-box adversary and is not subject to side-channel attacks.

With regard to the white-box program, it is important that the verification key $k_{\mathtt{HWs}}$ is not stored in plain since, otherwise, one might derive PRF/MAC values using $k_{\mathtt{HWs}}$ rather than querying the hardware. Thus, the $\mathtt{Check}_{\mathtt{SW}}$ functionality will need to be white-boxed (and will later be bound to another functionality such as our white-box KDF), essentially making it asymmetric. Note that the syntax and correctness of our hardware module also allows to be directly implemented by a (standard, asymmetric) signature scheme such as provided by the Android Keystore [4]. In that case, the verification key does not need additional protection. We chose to implement the hardware with a symmetric primitive for efficiency and simplicity of the proof. Note that regardless of whether one uses MACs or signature schemes as a hardware functionality, the verification functionality needs to be cryptographically *bound to a software program* to be useful; else the software program can be code-lifted and run without performing the verification check.

Regardless of efficiency, both approaches, using signature schemes directly or making MAC verification asymmetric, are sound approaches. Importantly, in both cases, the soundness of the approach relies on *domain separation*, i.e., signatures/MAC for one *label* should not be mixed up with signatures/MACs for

a different *label*. Finally, recall that fixed device identifiers (that do not depend on the input x) tend to provide very weak hardware-binding guarantees only, since once intercepted, they can be emulated for a code-lifted software program. We now give the syntax for our hardware-module.

Definition 11 (Hardware Module HWM). *A* hardware module HWM *consists of four algorithms* $(\mathtt{Kgen}_{\mathtt{HW}}, \mathtt{SubKgen}_{\mathtt{HW}}, \mathtt{Resp}_{\mathtt{HW}}, \mathtt{Check}_{\mathtt{SW}})$*, where* $\mathtt{Kgen}_{\mathtt{HW}}$ *is a PPT algorithm, and the algorithms* $\mathtt{SubKgen}_{\mathtt{HW}}$*,* $\mathtt{Resp}_{\mathtt{HW}}$ *and* $\mathtt{Check}_{\mathtt{SW}}$ *are deterministic algorithms with the following syntax:*

$$k_{\mathtt{HWm}} \leftarrow\!\!\$\ \mathtt{Kgen}_{\mathtt{HW}}(1^n) \qquad \sigma \leftarrow \mathtt{Resp}_{\mathtt{HW}}(k_{\mathtt{HWm}}, \mathit{label}, x)$$
$$k_{\mathtt{HWs}} \leftarrow \mathtt{SubKgen}_{\mathtt{HW}}(k_{\mathtt{HWm}}, \mathit{label}) \qquad b \leftarrow \mathtt{Check}_{\mathtt{SW}}(k_{\mathtt{HWs}}, x, \sigma),$$

Correctness requires that for all security parameters $n \in \mathbb{N}$*,*

$$\Pr[\mathtt{Check}_{\mathtt{SW}}(\mathtt{SubKgen}_{\mathtt{HW}}(k_{\mathtt{HWm}}, \mathit{label}), x, \mathtt{Resp}_{\mathtt{HW}}(k_{\mathtt{HWm}}, \mathit{label}, x)) = 1] = 1,$$

where the probability is over the sampling of $k_{\mathtt{HWm}} \leftarrow\!\!\$\ \mathtt{Kgen}_{\mathtt{HW}}(1^n)$*.*

We do not define or prove security of the hardware module as a standalone primitive, since we will later prove security of our white-box KDF directly based on the puncturable PRFs used in the hardware module construction. Note that a standalone security definition for a hardware module could not capture MACs/PRFs and signature security simultaneously. In our hardware module construction below, key generation samples a random key, sub-key generation applies a PRF to it to derive a sub-key, and $\mathtt{Resp}_{\mathtt{HW}}$ and $\mathtt{Check}_{\mathtt{SW}}$ essentially implement a PRF-based MAC. Note that the additional PRG evaluation in $\mathtt{Check}_{\mathtt{SW}}$ is merely used to enable the proof technique by Sahai and Waters [35].

Construction 1. *Let* PRG *be a pseudorandom generator,* PRF *be a pseudorandom function and* PPRF *be a puncturable pseudorandom function. We construct a hardware module* HWM *as follows:*

$\underline{\mathtt{Kgen}_{\mathtt{HW}}(1^n)}$
1 : $k_{\mathtt{HWm}} \leftarrow\!\!\$\ \{0,1\}^n$
2 : **return** $k_{\mathtt{HWm}}$

$\underline{\mathtt{SubKgen}_{\mathtt{HW}}(k_{\mathtt{HWm}}, \mathit{label})}$
1 : $k_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label})$
2 : **return** $k_{\mathtt{HWs}}$

$\underline{\mathtt{Resp}_{\mathtt{HW}}(k_{\mathtt{HWm}}, \mathit{label}, x)}$
1 : $\sigma \leftarrow \mathtt{PPRF}(\mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}), x)$
2 : **return** σ

$\underline{\mathtt{Check}_{\mathtt{SW}}(k_{\mathtt{HWs}}, x, \sigma)}$
1 : **if** $\mathtt{PRG}(\sigma) = \mathtt{PRG}(\mathtt{PPRF}(k_{\mathtt{HWs}}, x))$
2 : **return** 1 **else return** 0

Hardware-bound White-box Key Derivation Function. We now define and construct a hardware-bound white-box key derivation function WKDF. We here build on the previously introduced hardware module and a traditional KDF. In the compiling phase, a compiler Comp takes as input the KDF key $k_{\mathtt{kdf}}$ and the sub-key $k_{\mathtt{HWs}}$ for $\mathtt{Check}_{\mathtt{SW}}$. The compiler generates a program WKDF which

takes as input a pair (e, σ) and, intuitively, first checks whether σ is valid for e under $k_{\mathtt{HWs}}$ and, if so, evaluates the KDF on $k_{\mathtt{kdf}}$ and e. The role of the compiler, conceptually, is to return a program where the KDF operation is *bound* to the verification operation, i.e., the two functionalities cannot be separated from each other and the (outcome of the) verification cannot be manipulated.

Definition 12 (WKDF). *A* white-box key derivation scheme with hardware binding WKDF *consists of a* hardware module HWM*, a* key derivation function KDF*, and a PPT compiling algorithm* Comp*:*

$$\mathtt{WKDF} \leftarrow\!\!\$\, \mathtt{Comp}(k_{\mathtt{kdf}}, k_{\mathtt{HWs}}).$$

For all genuine $k_{\mathtt{HWm}}$*, for all* $k_{\mathtt{kdf}}$*, for all label, for all* e*, for all* $k_{\mathtt{HWs}} = \mathtt{SubKgen}_{\mathtt{HW}}(k_{\mathtt{HWm}}, label)$ *and* $\sigma = \mathtt{Resp}_{\mathtt{HW}}(k_{\mathtt{HWm}}, label, e)$*, we have*

$$\Pr[\mathtt{KDF}(k_{\mathtt{kdf}}, e) = \mathtt{WKDF}(e, \sigma)] = 1$$

where the probability is taken over compiling $\mathtt{WKDF} \leftarrow\!\!\$\, \mathtt{Comp}(k_{\mathtt{kdf}}, k_{\mathtt{HWs}})$*.*

Security Model. We now define security for a WKDF via the IND-WKDF security game, illustrated in Figure 5. We want to capture the pseudorandomness of keys derived from the WKDF, i.e., an adversary should not be able to distinguish between a key produced from the WKDF and an equally long key sampled at random. As a white-box adversary, the game provides the adversary with the capability of inspecting the WKDF itself (i.e. its circuit or implementation code). Recall that additionally, we want to capture the notion of hardware-binding: if the adversary tries to run the WKDF without having access to its designated hardware, the WKDF should not be executable anymore.

We model this by giving the adversary hardware access via a OResp oracle, which the adversary queries by providing a context value e and which the adversary can query a polynomial number of times. With the reply σ from the hardware component, the adversary is able to run WKDF on the context value e used for querying OResp. Additionally, we grant the adversary access to an oracle OSubKgen. This oracle produces hardware sub-keys which might be used to generate new hardware-bound white-box key derivation functions. To avoid trivial attacks, the adversary is not allowed to request a sub-key under the same *label* that was used to generate the initial WKDF. To capture the pseudorandomness of derived keys, the adversary has access to a real-or-random oracle OKDF. The oracle OKDF first samples a new context value e and then, depending on a random bit b, it either returns the output of the KDF (under key $k_{\mathtt{kdf}}$) or a random string of equal length. To avoid trivial attacks, the adversary is not allowed to query the OResp oracle with the same context value that was sampled by the challenger. Finally, $\mathcal{A}$ outputs a bit b' and wins if $b' = b$.

Definition 13 (IND-WKDF). *We say that a hardware-bound white-box key derivation scheme* WKDF *is* IND-WKDF*-secure if all PPT adversaries* $\mathcal{A}$ *have a negligible distinguishing advantage in game* $\mathsf{Exp}^{\mathsf{IND\text{-}WKDF}}_{\mathtt{WKDF},\mathcal{A}}(1^n)$*, see Figure 5.*

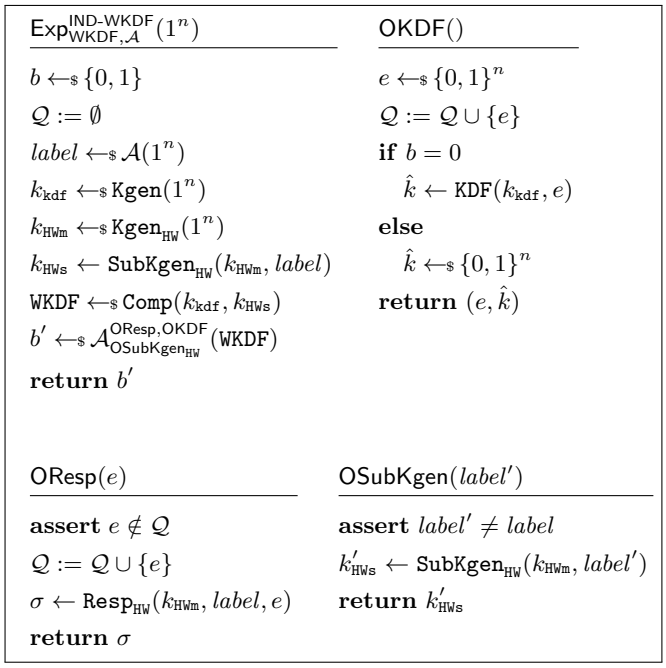


Fig. 5. The $\mathsf{Exp}^{\mathsf{IND\text{-}WKDF}}_{\mathsf{WKDF},\mathcal{A}}(1^n)$ security game.

Note that one could also provide the adversary with a recompilation oracle [22] so that the adversary can request several (independent) copies of the WKDF based on the same key. We refrain from including this feature into our model, but note that our construction can be shown to achieve also this stronger notion, as indistinguishability obfuscation makes recompilation adversarially simulatable.

3.2 Construction of a WKDF

We now construct a WKDF, based on a traditional KDF and the previously introduced hardware module HWM. As discussed before, the compiler Comp, on input the KDF key $k_{\mathtt{kdf}}$ and the hardware sub-key $k_{\mathtt{HWs}}$ binds the hardware $\mathtt{Check}_{\mathtt{SW}}$ procedure to the KDF evaluation. Concretely, the compiler constructs a circuit $\mathsf{C}[k_{\mathtt{kdf}}, k_{\mathtt{HWs}}]$ and obfuscates it using indistinguishability obfuscation. The circuit $\mathsf{C}[k_{\mathtt{kdf}}, k_{\mathtt{HWs}}]$, on input (e, σ) first checks whether $\mathtt{Check}_{\mathtt{SW}}(k_{\mathtt{HWs}}, e, \sigma)$ equals 1. If yes, it returns the output of $\mathtt{KDF}(k_{\mathtt{kdf}}, e)$. Else, it returns the all-zero string. The reason that the construction is secure, is, intuitively, that the obfuscation of $\mathsf{C}[k_{\mathtt{kdf}}, k_{\mathtt{HWs}}]$ achieves the desired binding property. We now first give the KDF construction and then the WKDF construction directly below.

Construction 2. *Let* PPRF *be a puncturable pseudorandom function scheme, then we construct our* KDF *as follows:*

$\underline{\mathtt{Kgen}_{\mathtt{kdf}}(1^n)}$

1 : $k_{\mathtt{kdf}} \leftarrow\!\!\$\ \{0,1\}^n$
2 : **return** $k_{\mathtt{kdf}}$

$\underline{\mathtt{KDF}(k_{\mathtt{kdf}}, e)}$

1 : $\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$
2 : **return** $\hat{k}$

Construction 3. *Let* iO *be an indistinguishability obfuscator. Based on the hardware module* HWM *given in Construction 1 and the key derivation scheme* KDF *given in Construction 2, we construct* WKDF *by defining the following compiler* Comp*:*

$\underline{\mathsf{C}[k_{\mathtt{kdf}}, k_{\mathtt{HWs}}](e, \sigma)}$

1 : $v \leftarrow \mathtt{Check}_{\mathtt{SW}}(k_{\mathtt{HWs}}, e, \sigma)$
2 : **if** $v = \perp$ **return** 0^n
3 : **else**
4 : $\quad \hat{k} \leftarrow \mathtt{KDF}(k_{\mathtt{kdf}}, e)$
5 : $\quad$ **return** $\hat{k}$

$\underline{\mathtt{Comp}(k_{\mathtt{kdf}}, k_{\mathtt{HWs}})}$

1 : $\mathtt{WKDF} \leftarrow\!\!\$\ \mathtt{iO}(\mathsf{C}[k_{\mathtt{kdf}}, k_{\mathtt{HWs}}])$
2 : **return** WKDF

Theorem 1. *Let* PRG *be a pseudorandom generator, let* PRF *be a pseudorandom function, let* PPRF *be a puncturable PRF, and* iO *be an indistinguishability obfuscator for appropriate p-admissible samplers. Then Construction 3 is a secure white-box KDF scheme* WKDF*.*

Proof. Let $\mathcal{A}$ be a PPT adversary. Let $\mathsf{Exp}^{\mathsf{IND\text{-}WKDF}}_{\mathcal{A},b}$ denote the IND-WKDF game with a value $b \in \{0,1\}$ hardcoded. We show that

$$\mathsf{Exp}^{\mathsf{IND\text{-}WKDF}}_{\mathcal{A},0}(1^n) \approx \mathsf{Exp}^{\mathsf{IND\text{-}WKDF}}_{\mathcal{A},1}(1^n).$$

Overview: The proof is a hybrid argument over the number of queries q that $\mathcal{A}$ makes to the OKDF oracle which either evaluates a puncturable PRF and returns its output or a random string of the same length. Our hybrid games maintain a counter j that increases by 1 whenever the adversary queries the OKDF oracle. The i-th hybrid game Game^i_1 returns a random string whenever the counter $j > i$, otherwise it returns the evaluation of the PPRF. In other words, whenever we move to the next hybrid, the oracle returns an additional random string such that we sequentially replace PPRF values by random strings of appropriate size. After at most polynomial steps we have replaced all OKDF outputs by random values and obtain $\mathsf{Exp}^{\mathsf{IND\text{-}WKDF}}_{\mathcal{A},1}(1^n)$.
Detailed proof: Let $q(n)$ be a polynomial which is a strict upper bound on the number of queries that $\mathcal{A}$ makes to oracle OKDF. We define a sequence of adversary-dependent hybrid games Game^0_1 to $\mathsf{Game}^{q(n)}_1$ such that

$$\mathsf{Exp}^{\mathsf{IND\text{-}WKDF}}_{\mathcal{A},0} \approx \mathsf{Game}^0_1 \tag{1}$$
$$\mathsf{Exp}^{\mathsf{IND\text{-}WKDF}}_{\mathcal{A},1} \approx \mathsf{Game}^{q(n)}_1. \tag{2}$$

Using 18 game hops we show that for $0 \leq i \leq q(n) - 1$:

$$\mathsf{Game}^i_1 \approx \mathsf{Game}^i_{18} \tag{3}$$
$$\mathsf{Game}^i_{18} \approx \mathsf{Game}^{i+1}_1. \tag{4}$$

Indistinguishability of Game^0_1 and $\mathsf{Game}^{q(n)}_1$ then follows by a standard hybrid argument, guessing the hybrid index at random. We define the games and specify the game-hops below and the required circuit definitions are depicted on the right-hand side of this page. Equation 4 follows by inspection of the definitions of Game^i_{18} and Game^{i+1}_1. We now turn to showing Equation 3 which is the technical heart of the theorem.

$\mathsf{C}_1[k_{\mathtt{kdf}}, k_{\mathtt{HWs}}](e, \sigma)$

if $\mathtt{PRG}(\sigma) = \mathtt{PRG}(\mathtt{PPRF}(k_{\mathtt{HWs}}, e))$
  $\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$
  return $\hat{k}$
else return 0^n

$\mathsf{C}_2[k_{\mathtt{kdf}}, z, k_z, \tau](e, \sigma)$

if $e = z$ and $\mathtt{PRG}(\sigma) = \mathtt{PRG}(\tau)$
or if $\mathtt{PRG}(\sigma) = \mathtt{PRG}(\mathtt{Eval}(k_z, e))$
  $\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$
  return $\hat{k}$
else return 0^n

$\mathsf{C}_3[k_{\mathtt{kdf}}, z, k_z, y](e, \sigma)$

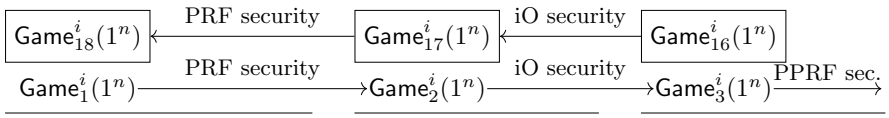
if $e = z$ and $\mathtt{PRG}(\sigma) = y$
or if $\mathtt{PRG}(\sigma) = \mathtt{PRG}(\mathtt{Eval}(k_z, e))$
  $\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$
  return $\hat{k}$
else return 0^n

$\mathsf{C}_4[k^z_{\mathtt{kdf}}, z, k_z, y, k](e, \sigma)$

if $e = z$ and $\mathtt{PRG}(\sigma) = y$
  return k
if $\mathtt{PRG}(\sigma) = \mathtt{PRG}(\mathtt{Eval}(k_z, e))$
  $\hat{k} \leftarrow \mathtt{Eval}(k^z_{\mathtt{kdf}}, e)$
  return $\hat{k}$
else return 0^n

$\mathsf{C}_5[k^z_{\mathtt{kdf}}, z, k_z, y](e, \sigma)$

if $e = z$ and $\mathtt{PRG}(\sigma) = y$
  return 0^n
if $\mathtt{PRG}(\sigma) = \mathtt{PRG}(\mathtt{Eval}(k_z, e))$
  $\hat{k} \leftarrow \mathtt{Eval}(k^z_{\mathtt{kdf}}, e)$
  return $\hat{k}$
else return 0^n

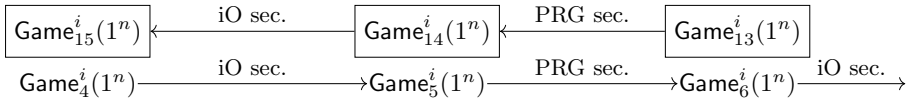


$\mathsf{Game}^i_{18}(1^n)$	$\mathsf{Game}^i_{17}(1^n)$	$\mathsf{Game}^i_{16}(1^n)$
$z \leftarrow\!\!\$\ \{0,1\}^n$, $\mathcal{Q} := \{z\}$	$z \leftarrow\!\!\$\ \{0,1\}^n$, $\mathcal{Q} := \{z\}$	$z \leftarrow\!\!\$\ \{0,1\}^n$, $\mathcal{Q} := \{z\}$
$j \leftarrow 0$	$j \leftarrow 0$	$j \leftarrow 0$
$\mathit{label} \leftarrow\!\!\$\ \mathcal{A}(1^n)$	$\mathit{label} \leftarrow\!\!\$\ \mathcal{A}(1^n)$	$\mathit{label} \leftarrow\!\!\$\ \mathcal{A}(1^n)$
$k_{\mathtt{kdf}} \leftarrow\!\!\$\ \{0,1\}^n$	$k_{\mathtt{kdf}} \leftarrow\!\!\$\ \{0,1\}^n$	$k_{\mathtt{kdf}} \leftarrow\!\!\$\ \{0,1\}^n$
$k \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, z)$	$k \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, z)$	$k \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, z)$
$k_{\mathtt{HWm}} \leftarrow\!\!\$\ \{0,1\}^n$	$k_{\mathtt{HWm}} \leftarrow\!\!\$\ \{0,1\}^n$	$k_{\mathtt{HWm}} \leftarrow\!\!\$\ \{0,1\}^n$
$k_{\mathtt{HWs}} \leftarrow\!\!\$\ \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label})$	$k_{\mathtt{HWs}} \leftarrow\!\!\$\ \{0,1\}^n$	$k_{\mathtt{HWs}} \leftarrow\!\!\$\ \{0,1\}^n$
		$k_z \leftarrow\!\!\$\ \mathtt{Punct}(k_{\mathtt{HWs}}, z)$
		$\tau \leftarrow \mathtt{PPRF}(k_{\mathtt{HWs}}, z)$
$\mathsf{C} \leftarrow \mathsf{C}_1[k_{\mathtt{kdf}}, k_{\mathtt{HWs}}]$	$\mathsf{C} \leftarrow \mathsf{C}_1[k_{\mathtt{kdf}}, k_{\mathtt{HWs}}]$	$\mathsf{C} \leftarrow \mathsf{C}_2[k_{\mathtt{kdf}}, z, k_z, \tau]$
$\mathtt{WKDF} \leftarrow\!\!\$\ \mathtt{iO}(\mathsf{C}, 1^n)$	$\mathtt{WKDF} \leftarrow\!\!\$\ \mathtt{iO}(\mathsf{C}, 1^n)$	$\mathtt{WKDF} \leftarrow\!\!\$\ \mathtt{iO}(\mathsf{C}, 1^n)$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Resp}_{\mathtt{HW}}, \mathsf{OKDF}}_{\mathsf{OSubKgen}}(\mathtt{WKDF})$	$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Resp}_{\mathtt{HW}}, \mathsf{OKDF}}_{\mathsf{OSubKgen}}(\mathtt{WKDF})$	$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Resp}_{\mathtt{HW}}, \mathsf{OKDF}}_{\mathsf{OSubKgen}}(\mathtt{WKDF})$
return b'	**return** b'	**return** b'

$\mathsf{OResp}(e)$	$\mathsf{OResp}(e)$	$\mathsf{OResp}(e)$
assert $e \notin \mathcal{Q}$	**assert** $e \notin \mathcal{Q}$	**assert** $e \notin \mathcal{Q}$
$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$
$y \leftarrow \mathtt{PPRF}(k_{\mathtt{HWs}}, e, 1^n)$	$y \leftarrow \mathtt{PPRF}(k_{\mathtt{HWs}}, e, 1^n)$	$y \leftarrow \mathtt{Eval}(k_z, e, 1^n)$
return y	**return** y	**return** y

$\mathsf{OKDF}()$	$\mathsf{OKDF}()$	$\mathsf{OKDF}()$
$j \leftarrow j + 1$	$j \leftarrow j + 1$	$j \leftarrow j + 1$
if $i = j$ $\boxed{k' \leftarrow\!\!\$\ \{0,1\}^n}$	**if** $i = j$ $\boxed{k' \leftarrow\!\!\$\ \{0,1\}^n}$	**if** $i = j$ $\boxed{k' \leftarrow\!\!\$\ \{0,1\}^n}$
return (z,k) $\boxed{(z,k')}$	**return** (z,k) $\boxed{(z,k')}$	**return** (z,k) $\boxed{(z,k')}$
else $e \leftarrow\!\!\$\ \{0,1\}^n$	**else** $e \leftarrow\!\!\$\ \{0,1\}^n$	**else** $e \leftarrow\!\!\$\ \{0,1\}^n$
$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$
if $j > i$	**if** $j > i$	**if** $j > i$
$\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$	$\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$	$\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$
else $\hat{k} \leftarrow\!\!\$\ \{0,1\}^n$	**else** $\hat{k} \leftarrow\!\!\$\ \{0,1\}^n$	**else** $\hat{k} \leftarrow\!\!\$\ \{0,1\}^n$
return $(e, \hat{k})$	**return** $(e, \hat{k})$	**return** $(e, \hat{k})$

$\mathsf{OSubKgen}_{\mathtt{HW}}(\mathit{label}')$	$\mathsf{OSubKgen}_{\mathtt{HW}}(\mathit{label}')$	$\mathsf{OSubKgen}_{\mathtt{HW}}(\mathit{label}')$
assert $\mathit{label}' \neq \mathit{label}$	**assert** $\mathit{label}' \neq \mathit{label}$	**assert** $\mathit{label}' \neq \mathit{label}$
$k'_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}')$	$k'_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}')$	$k'_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}')$
return $k'_{\mathtt{HWs}}$	**return** $k'_{\mathtt{HWs}}$	**return** $k'_{\mathtt{HWs}}$



$\mathsf{Game}^i_{15}(1^n)$	$\mathsf{Game}^i_{14}(1^n)$	$\mathsf{Game}^i_{13}(1^n)$
$z \leftarrow\!\!\$\, \{0,1\}^n$, $\mathcal{Q} := \{z\}$	$z \leftarrow\!\!\$\, \{0,1\}^n$, $\mathcal{Q} := \{z\}$	$z \leftarrow\!\!\$\, \{0,1\}^n$, $\mathcal{Q} := \{z\}$
$j \leftarrow 0$	$j \leftarrow 0$	$j \leftarrow 0$
$\mathit{label} \leftarrow\!\!\$\, \mathcal{A}(1^n)$	$\mathit{label} \leftarrow\!\!\$\, \mathcal{A}(1^n)$	$\mathit{label} \leftarrow\!\!\$\, \mathcal{A}(1^n)$
$k_{\mathtt{kdf}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{kdf}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{kdf}} \leftarrow\!\!\$\, \{0,1\}^n$
$k \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, z)$	$k \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, z)$	$k \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, z)$
$k_{\mathtt{HWm}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{HWm}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{HWm}} \leftarrow\!\!\$\, \{0,1\}^n$
$k_{\mathtt{HWs}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{HWs}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{HWs}} \leftarrow\!\!\$\, \{0,1\}^n$
$k_z \leftarrow\!\!\$\, \mathtt{Punct}(k_{\mathtt{HWs}}, z)$	$k_z \leftarrow\!\!\$\, \mathtt{Punct}(k_{\mathtt{HWs}}, z)$	$k_z \leftarrow\!\!\$\, \mathtt{Punct}(k_{\mathtt{HWs}}, z)$
$\tau \leftarrow\!\!\$\, \{0,1\}^n$	$\tau \leftarrow\!\!\$\, \{0,1\}^n$; $y \leftarrow \mathtt{PRG}(\tau)$	$y \leftarrow\!\!\$\, \{0,1\}^{2n}$
$\mathsf{C} \leftarrow \mathsf{C}_2[k_{\mathtt{kdf}}, z, k_z, \tau]$	$\mathsf{C} \leftarrow \mathsf{C}_3[k_{\mathtt{kdf}}, z, k_z, y]$	$\mathsf{C} \leftarrow \mathsf{C}_3[k_{\mathtt{kdf}}, z, k_z, y]$
$\mathtt{WKDF} \leftarrow\!\!\$\, \mathtt{iO}(\mathsf{C}, 1^n)$	$\mathtt{WKDF} \leftarrow\!\!\$\, \mathtt{iO}(\mathsf{C}, 1^n)$	$\mathtt{WKDF} \leftarrow\!\!\$\, \mathtt{iO}(\mathsf{C}, 1^n)$
$b' \leftarrow\!\!\$\, \mathcal{A}^{\mathsf{Resp}_{\mathtt{HW}}, \mathsf{OKDF}}_{\mathsf{OSubKgen}}(\mathtt{WKDF})$	$b' \leftarrow\!\!\$\, \mathcal{A}^{\mathsf{Resp}_{\mathtt{HW}}, \mathsf{OKDF}}_{\mathsf{OSubKgen}}(\mathtt{WKDF})$	$b' \leftarrow\!\!\$\, \mathcal{A}^{\mathsf{Resp}_{\mathtt{HW}}, \mathsf{OKDF}}_{\mathsf{OSubKgen}}(\mathtt{WKDF})$
return b'	**return** b'	**return** b'

$\mathsf{OResp}(e)$

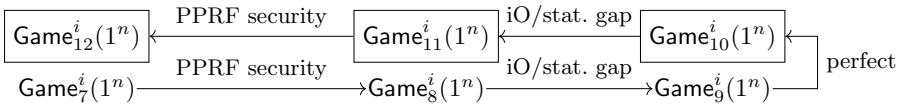
assert $e \notin \mathcal{Q}$	**assert** $e \notin \mathcal{Q}$	**assert** $e \notin \mathcal{Q}$
$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$
$y \leftarrow \mathtt{Eval}(k_z, e, 1^n)$	$y \leftarrow \mathtt{Eval}(k_z, e, 1^n)$	$y \leftarrow \mathtt{Eval}(k_z, e, 1^n)$
return y	**return** y	**return** y

$\mathsf{OKDF}()$

$j \leftarrow j + 1$	$j \leftarrow j + 1$	$j \leftarrow j + 1$
if $i = j$ $\boxed{k' \leftarrow\!\!\$\, \{0,1\}^n}$	**if** $i = j$ $\boxed{k' \leftarrow\!\!\$\, \{0,1\}^n}$	**if** $i = j$ $\boxed{k' \leftarrow\!\!\$\, \{0,1\}^n}$
return (z, k) $\boxed{(z, k')}$	**return** (z, k) $\boxed{(z, k')}$	**return** (z, k) $\boxed{(z, k')}$
else $e \leftarrow\!\!\$\, \{0,1\}^n$	**else** $e \leftarrow\!\!\$\, \{0,1\}^n$	**else** $e \leftarrow\!\!\$\, \{0,1\}^n$
$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$
if $j > i$	**if** $j > i$	**if** $j > i$
$\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$	$\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$	$\hat{k} \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, e)$
else $\hat{k} \leftarrow\!\!\$\, \{0,1\}^n$	**else** $\hat{k} \leftarrow\!\!\$\, \{0,1\}^n$	**else** $\hat{k} \leftarrow\!\!\$\, \{0,1\}^n$
return $(e, \hat{k})$	**return** $(e, \hat{k})$	**return** $(e, \hat{k})$

$\mathsf{OSubKgen}_{\mathtt{HW}}(\mathit{label}')$

assert $\mathit{label}' \neq \mathit{label}$	**assert** $\mathit{label}' \neq \mathit{label}$	**assert** $\mathit{label}' \neq \mathit{label}$
$k'_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}')$	$k'_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}')$	$k'_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}')$
return $k'_{\mathtt{HWs}}$	**return** $k'_{\mathtt{HWs}}$	**return** $k'_{\mathtt{HWs}}$



$\mathsf{Game}^i_{12}(1^n)$	$\mathsf{Game}^i_{11}(1^n)$	$\mathsf{Game}^i_{10}(1^n)$
$z \leftarrow\!\!\$\, \{0,1\}^n$, $\mathcal{Q} := \{z\}$	$z \leftarrow\!\!\$\, \{0,1\}^n$, $\mathcal{Q} := \{z\}$	$z \leftarrow\!\!\$\, \{0,1\}^n$, $\mathcal{Q} := \{z\}$
$j \leftarrow 0$	$j \leftarrow 0$	$j \leftarrow 0$
$\mathit{label} \leftarrow\!\!\$\, \mathcal{A}(1^n)$	$\mathit{label} \leftarrow\!\!\$\, \mathcal{A}(1^n)$	$\mathit{label} \leftarrow\!\!\$\, \mathcal{A}(1^n)$
$k_{\mathtt{kdf}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{kdf}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{kdf}} \leftarrow\!\!\$\, \{0,1\}^n$
$k^z_{\mathtt{kdf}} \leftarrow\!\!\$\, \mathtt{Punct}(k_{\mathtt{kdf}}, z)$	$k^z_{\mathtt{kdf}} \leftarrow\!\!\$\, \mathtt{Punct}(k_{\mathtt{kdf}}, z)$	$k^z_{\mathtt{kdf}} \leftarrow\!\!\$\, \mathtt{Punct}(k_{\mathtt{kdf}}, z)$
$k \leftarrow \mathtt{PPRF}(k_{\mathtt{kdf}}, z)$	$k \leftarrow \{0,1\}^n$	$k \leftarrow\!\!\$\, \{0,1\}^n$
$k_{\mathtt{HWm}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{HWm}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{HWm}} \leftarrow\!\!\$\, \{0,1\}^n$
$k_{\mathtt{HWs}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{HWs}} \leftarrow\!\!\$\, \{0,1\}^n$	$k_{\mathtt{HWs}} \leftarrow\!\!\$\, \{0,1\}^n$
$k_z \leftarrow\!\!\$\, \mathtt{Punct}(k_{\mathtt{HWs}}, z)$	$k_z \leftarrow\!\!\$\, \mathtt{Punct}(k_{\mathtt{HWs}}, z)$	$k_z \leftarrow\!\!\$\, \mathtt{Punct}(k_{\mathtt{HWs}}, z)$
$y \leftarrow\!\!\$\, \{0,1\}^{2n}$	$y \leftarrow\!\!\$\, \{0,1\}^{2n}$	$y \leftarrow\!\!\$\, \{0,1\}^{2n}$
$\mathsf{C} \leftarrow \mathsf{C}_4[k^z_{\mathtt{kdf}}, z, k_z, y, k]$	$\mathsf{C} \leftarrow \mathsf{C}_4[k^z_{\mathtt{kdf}}, z, k_z, y, k]$	$\mathsf{C} \leftarrow \mathsf{C}_5[k^z_{\mathtt{kdf}}, z, k_z, y]$
$\mathtt{WKDF} \leftarrow\!\!\$\, \mathtt{iO}(\mathsf{C}, 1^n)$	$\mathtt{WKDF} \leftarrow\!\!\$\, \mathtt{iO}(\mathsf{C}, 1^n)$	$\mathtt{WKDF} \leftarrow\!\!\$\, \mathtt{iO}(\mathsf{C}, 1^n)$
$b' \leftarrow\!\!\$\, \mathcal{A}^{\mathsf{Resp}_{\mathtt{HW}}, \mathsf{OKDF}}_{\mathsf{OSubKgen}}(\mathtt{WKDF})$	$b' \leftarrow\!\!\$\, \mathcal{A}^{\mathsf{Resp}_{\mathtt{HW}}, \mathsf{OKDF}}_{\mathsf{OSubKgen}}(\mathtt{WKDF})$	$b' \leftarrow\!\!\$\, \mathcal{A}^{\mathsf{Resp}_{\mathtt{HW}}, \mathsf{OKDF}}_{\mathsf{OSubKgen}}(\mathtt{WKDF})$
return b'	**return** b'	**return** b'

$\mathsf{OResp}(e)$	$\mathsf{OResp}(e)$	$\mathsf{OResp}(e)$
assert $e \notin \mathcal{Q}$	**assert** $e \notin \mathcal{Q}$	**assert** $e \notin \mathcal{Q}$
$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$
$y \leftarrow \mathtt{Eval}(k_z, e, 1^n)$	$y \leftarrow \mathtt{Eval}(k_z, e, 1^n)$	$y \leftarrow \mathtt{Eval}(k_z, e, 1^n)$
return y	**return** y	**return** y

$\mathsf{OKDF}()$	$\mathsf{OKDF}()$	$\mathsf{OKDF}()$
$j \leftarrow j + 1$	$j \leftarrow j + 1$	$j \leftarrow j + 1$
if $i = j$ $\boxed{k' \leftarrow\!\!\$\, \{0,1\}^n}$	**if** $i = j$ $\boxed{k' \leftarrow\!\!\$\, \{0,1\}^n}$	**if** $i = j$ $\boxed{k' \leftarrow\!\!\$\, \{0,1\}^n}$
return (z,k) $\boxed{(z,k')}$	**return** (z,k) $\boxed{(z,k')}$	**return** (z,k) $\boxed{(z,k')}$
else $e \leftarrow\!\!\$\, \{0,1\}^n$	**else** $e \leftarrow\!\!\$\, \{0,1\}^n$	**else** $e \leftarrow\!\!\$\, \{0,1\}^n$
$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$	$\mathcal{Q} := \mathcal{Q} \cup \{e\}$
if $j > i$	**if** $j > i$	**if** $j > i$
$\hat{k} \leftarrow \mathtt{Eval}(k^z_{\mathtt{kdf}}, e)$	$\hat{k} \leftarrow \mathtt{Eval}(k^z_{\mathtt{kdf}}, e)$	$\hat{k} \leftarrow \mathtt{Eval}(k^z_{\mathtt{kdf}}, e)$
else $\hat{k} \leftarrow\!\!\$\, \{0,1\}^n$	**else** $\hat{k} \leftarrow\!\!\$\, \{0,1\}^n$	**else** $\hat{k} \leftarrow\!\!\$\, \{0,1\}^n$
return $(e, \hat{k})$	**return** $(e, \hat{k})$	**return** $(e, \hat{k})$

$\mathsf{OSubKgen}_{\mathtt{HW}}(\mathit{label}')$	$\mathsf{OSubKgen}_{\mathtt{HW}}(\mathit{label}')$	$\mathsf{OSubKgen}_{\mathtt{HW}}(\mathit{label}')$
assert $\mathit{label}' \neq \mathit{label}$	**assert** $\mathit{label}' \neq \mathit{label}$	**assert** $\mathit{label}' \neq \mathit{label}$
$k'_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}')$	$k'_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}')$	$k'_{\mathtt{HWs}} \leftarrow \mathtt{PRF}(k_{\mathtt{HWm}}, \mathit{label}')$
return $k'_{\mathtt{HWs}}$	**return** $k'_{\mathtt{HWs}}$	**return** $k'_{\mathtt{HWs}}$

We now reduce each game-hop to the underlying assumption. We omit boilerplate code for simulations and focus instead on describing the conceptual argument that underlies the reduction. We discuss the game-hops in the forward direction. The reductions for the backward direction proceed analogously.

Game 1 to Game 2. This game hop reduces to the PRF security of the PRF keyed with $k_{\texttt{HWm}}$. We here rely on the fact that $\mathsf{OSubKgen}_{\texttt{HW}}(\mathit{label}')$ does not allow to query $\texttt{PRF}(k_{\texttt{HWm}}, \cdot)$ on *label*.

Game 2 to Game 3. This game hop reduces to iO security and relies on the correctness of the PPRF. The correctness of the PPRF implies that the first two lines of C_1 and C_2 are equivalent, which allows to apply iO security. Note that the oracle OResp cannot be queried on z since z is added to $\mathcal{Q}$ in the very beginning of the game. Therefore, it suffices to use the punctured key k_z in OResp.

Game 3 to Game 4. This game hop reduces to the IND-PPRF security of PPRF keyed with $k_{\texttt{HWs}}$ and punctured at z. For the reduction, it is important to note that throughout the game, only punctured versions k_z of $k_{\texttt{HWs}}$ are used, except for calculating τ.

Game 4 to Game 5. This game hop replaces $\mathsf{C}_2[k_{\texttt{kdf}}, z, k_z, \tau]$ by $\mathsf{C}_3[k_{\texttt{kdf}}, z, k_z, y]$ and relies on iO security. Instead of hardcoding τ into C_2 and computing y as $\texttt{PRG}(\tau)$ within circuit C_2, the value $y = \texttt{PRG}(\tau)$ is directly hardcoded into circuit C_3. As the two circuits are functionally equivalent, the game hop reduces to iO security.

Game 5 to Game 6. This game hop replaces $y = \texttt{PRG}(\tau)$ by a randomly sampled value y. This game hop reduces to PRG security, since the variable τ that is sampled in $\mathsf{Game}_5^i(1^n)$ is not used anywhere else in the game(s).

Game 6 to Game 7. This game hop reduces to iO security with an additional negligible statistical loss. In detail, the uniformity of the sampling of z from $\{0,1\}^n$ ensures that, with overwhelming probability, the oracle OKDF does not return $(e, *)$ for a different than the ith query of $\mathcal{A}$ to the OKDF oracle, i.e., the change to OKDF yields a negligible statistical difference between $\mathsf{Game}_6^i(1^n)$ and $\mathsf{Game}_7^i(1^n)$. The more important change is the use of $k^z_{\texttt{kdf}}$ in C_4. Due to the correctness of the puncturable PRF, the circuits C_3 and C_4 are functionally equivalent and thus, this game hop can be reduced to iO security.

Game 7 to Game 8. This game hop reduces to the IND-PPRF security of PPRF, keyed with $k_{\texttt{kdf}}$ and punctured at z. For the reduction, it is important to note that throughout the game, only the punctured version $k^z_{\texttt{kdf}}$ of $k_{\texttt{kdf}}$ is used, except for calculating k.

Game 8 to Game 9. Note that with overwhelming probability, y is not in the image of the $\texttt{PRG}$, since the image of the $\texttt{PRG}$ is of size 2^n only, whereas y is sampled from a set of size 2^{2n}. Therefore, y is most likely outside the image of the $\texttt{PRG}$. If it is, then the circuits C_4 and C_5 are functionally equivalent as the **if** condition in the first line of C_4 cannot be satisfied by any input. Thus, this game hop reduces to iO security.

Game 9 to Game 10. Importantly, C_5 does not depend on k anymore and thus, it is perfectly indistinguishable for the adversary whether the OKDF uses k or an independently drawn value k' that OKDF samples in the moment of the ith query.

Game 10 to Game 18. These game hops are analogous in the backward direction. Note that k is not used in OKDF anymore in the game hops from Game 10 to Game 18.

Connecting the hybrids. We now show that Equation 4 holds, which we recall is

$$\mathsf{Game}_{18}^{i} \approx \mathsf{Game}_{1}^{i+1}$$

On a high-level, Game_{18}^{i} and Game_{1}^{i+1} are identical since they both sample the first $i+1$ keys in the OKDF oracle at random and compute the remaining keys using the $\texttt{PPRF}$. Note that Game_{18}^{i} and Game_{1}^{i+1} are only identical up to a negligible statistical difference since the pre-sampled value z is consumed at query i to the OKDF oracle in Game_{18}^{i} and only in query $i+1$ in Game_{1}^{i+1}. However, as z is sampled uniformly at random from $\{0,1\}^n$, z remains statistically hidden from the adversary until it is returned as an output from the OKDF oracle. Thus, it is infeasible to determine when the pre-sampled value z was returned. We omit a detailed code-comparison, since it is quite simple.

Connecting the hybrids to the original game. Finally, we show Equation 1 and Equation 2. We start with the former which we recall is

$$\mathsf{Exp}_{\mathcal{A},0}^{\mathsf{IND\text{-}WKDF}} \approx \mathsf{Game}_{1}^{0}.$$

On a high-level, $\mathsf{Exp}_{\mathcal{A},0}^{\mathsf{IND\text{-}WKDF}}$ and Game_{1}^{0} are identical since they both compute all keys in the OKDF oracle using the $\texttt{PPRF}$. Moreover, due to the correctness of the hardware module, it is functionally equivalent to use $\texttt{PPRF}(k_{\texttt{HWs}}, e, 1^n)$ and $\texttt{PPRF}(\texttt{PRF}(k_{\texttt{HWm}}, e, 1^n))$ in oracle OResp. However, there is a negligible statistical difference between $\mathsf{Exp}_{\mathcal{A},0}^{\mathsf{IND\text{-}WKDF}}$ and Game_{1}^{0}, since Game_{1}^{0} pre-samples a uniformly random value z from $\{0,1\}^n$, while $\mathsf{Exp}_{\mathcal{A},0}^{\mathsf{IND\text{-}WKDF}}$ does not. However, as z is sampled uniformly at random, it remains statistically hidden from the adversary until it is returned as an output from the OKDF oracle. We omit a detailed code-comparison, since it is quite simple. The reasoning for Equation 2 is analogous, which concludes the proof of Theorem 1. □

4 Secure Payment Application

In this section we build a secure payment scheme, assuming an IND-WKDF-secure WKDF, as constructed in the previous section. As mentioned before, the main idea of the construction of the payment application is to bind a symmetric encryption scheme (that is only known to provide black-box security) on top of the WKDF via a layer of indistinguishability obfuscation and thereby bootstrap the security of the WKDF to the security of the symmetric encryption scheme and the entire payment application.

We start with describing the process flow of an abstract payment application, illustrated in Figure 6. Note that this abstract payment application might also be implementable differently than assuming a WKDF. A payment application relies on a hardware module (see Definition 11), which we recall has a main hardware key $k_{\mathtt{HWm}}$, allows to derive sub-keys $k_{\mathtt{HWs}} \leftarrow \mathtt{SubKgen}(k_{\mathtt{HWm}}, label)$ from the main hardware key and allows to obtain MAC/PRF values $\sigma \leftarrow \mathtt{Resp}_{\mathtt{HW}}(k_{\mathtt{HWm}}, label, x)$ that can be verified outside of the hardware via the algorithm $\mathtt{Check}_{\mathtt{SW}}(k_{\mathtt{HWs}}, x, \sigma)$ that returns 0 or 1.

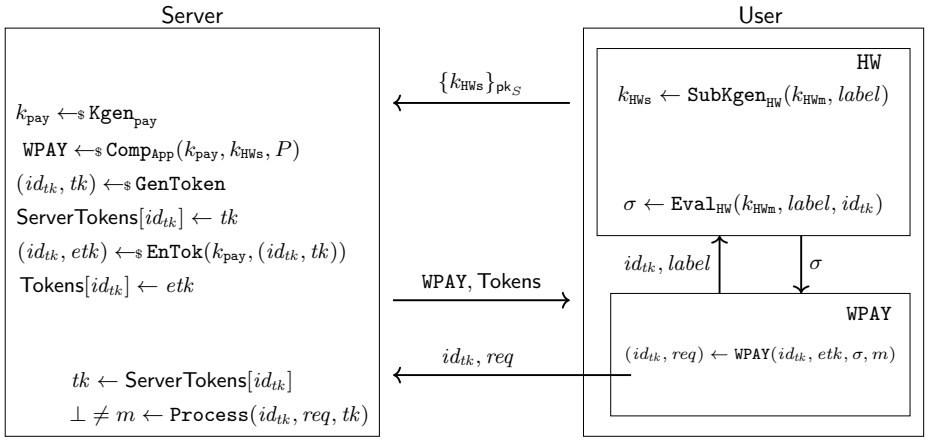


Fig. 6. Diagram of our payment scheme. The hardware calculates the value σ via the $\mathtt{Eval}_{\mathtt{HW}}$ function on input id_{tk}, the label and the master key. WPAY executes the $\mathtt{Check}_{\mathtt{SW}}$ function on input σ and on the sub-key and id_{tk}.

As the payment application is bound to a hardware module, the user starts by deriving a hardware sub-key $k_{\mathtt{HWs}} \leftarrow \mathtt{SubKgen}(k_{\mathtt{HWm}}, label)$ in their hardware and transmit it securely to the server. In Figure 6, we hint at this secure transmission of $k_{\mathtt{HWs}}$ via an encryption under the server's public-key pk_S. In our model, we later refrain from modeling this off-band transmission of $k_{\mathtt{HWs}}$ and simply assume that it is implemented securely.

The server then draws a symmetric key $k_{\mathtt{pay}}$ for the user and binds it to the user's hardware sub-key $k_{\mathtt{HWs}}$ via the compilation algorithm $\mathtt{Comp}_{\mathtt{App}}$:

$$\mathtt{WPAY} \leftarrow\!\!\$\ \mathtt{Comp}_{\mathtt{App}}(k_{\mathtt{pay}}, k_{\mathtt{HWs}}, P).$$

The predicate function P restricts the set of valid messages that can be encrypted via WPAY. An example for useful restrictions are limits on the amount of the payment or hardcoding of the user's payment data. Note that potentially, P can also contain cryptographic functionalities (which we do not model).

As we consider a tokenized payment scheme, in addition to WPAY the server also generates several tokens, encrypts them under $k_{\mathtt{pay}}$ and stores the encrypted tokens in an array Tokens that is indexed by token identifiers. It then transmits WPAY to the user, together with the array Tokens, see Figure 6.

Now, the user can use WPAY to generate requests to the server. To do so, WPAY takes as input a message m as well as a pair of a token identifier id_{tk} and its corresponding encrypted token etk. Conceptually, the goal of WPAY is to return a request req to the server that contains an encryption of m under the unencrypted token contained in etk. To facilitate verification on the server's side, the user's WPAY will also return the token identifier id_{tk}:

$$(id_{tk}, req) \leftarrow \mathtt{WPAY}(id_{tk}, etk, \sigma, m)$$

Importantly, in addition to the aforementioned inputs, WPAY also takes as input σ, which is a hardware value obtained from making an $\mathtt{Resp}_{\mathtt{HW}}$ query to the hardware for $(label, id_{tk})$. To ensure the hardware-binding, conceptually, WPAY needs to evaluate the algorithm $\mathtt{Check}_{\mathtt{SW}}(k_{\mathtt{HWs}}, id_{tk}, \sigma)$ internally and only perform the desired operations based on this check succeeding. Intuitively, the $\mathtt{Check}_{\mathtt{SW}}$ operation also needs to be bound to all further operations of WPAY.

Finally, upon receiving (id_{tk}, req), the server retrieves the token tk corresponding to id_{tk} and processes the request req via the algorithm Process. If the request is accepted, Process returns the message m that the client encrypted. Else, Process returns an error symbol $\perp$.

Note that we require the server to know the secret key $k_{\mathtt{pay}}$ to encrypt the tokens under $k_{\mathtt{pay}}$ before sending it to the user. The advantage of this design is that the values of the tokens are not exposed before being stored. Note that we consider the server to be a trusted and secure party which is a necessary assumption: As the server is in charge of generating the tokens, the server knows the token values anyway. Thus, the server additionally knowing the value of the secret key $k_{\mathtt{kdf}}$ of the user does not compromise the security of the mobile payment application from the perspective of the user.

Definition 14 (Hardware-Bound White-Box Payment Scheme). *A* hardware-bound white-box payment scheme WPAY *is parameterized with a message length parameter $\ell(n)$ and consists of a hardware module* HWM *and the following algorithms:*

- $k_{\mathtt{pay}} \leftarrow\!\!\$\ \mathtt{Kgen}_{\mathtt{pay}}(1^n)$: *This randomized algorithm takes as input the security parameter and outputs the secret payment key $k_{\mathtt{pay}}$;*

- $(id_{tk}, tk) \leftarrow_{\$} \texttt{GenToken}(1^n)$: *This randomized algorithm takes as input the security parameter and returns a token tk and a token identifier* id_{tk}*, where we assume that identifiers are unique with overwhelming probability;*
- $(id_{tk}, etk) \leftarrow_{\$} \texttt{EnTok}(k_{\texttt{pay}}, (id_{tk}, tk))$: *This randomized algorithm takes as input a token tk together with its corresponding identifier* id_{tk} *and outputs an encrypted token etk, together with its corresponding token identifier* id_{tk}*;*
- $\texttt{WPAY} \leftarrow_{\$} \texttt{Comp}_{\texttt{App}}(k_{\texttt{pay}}, k_{\texttt{HWs}}, P)$: *This randomized algorithm takes as input a payment key* $k_{\texttt{pay}}$*, a hardware-binding key* $k_{\texttt{HWs}}$ *and a message filtering predicate* $P : \{0,1\}^{\ell(n)} \rightarrow \{0,1\}$*. It outputs a white-box payment application* WPAY *with syntax* $(id_{tk}, req) \leftarrow \texttt{WPAY}(id_{tk}, etk, \sigma, m)$*, where* σ *denotes a hardware value* σ*,* $m \in \{0,1\}^{\ell(n)}$ *denotes a payment message, and req constitutes a payment request;*
- $m \leftarrow \texttt{Process}(id_{tk}, req, tk)$: *This deterministic algorithm takes as input a token identifier* id_{tk}*, a token value tk and a a request req. It outputs a message* m *or* $\perp$*.*

Moreover, we require that the following correctness property holds: For all keys $k_{\texttt{pay}}$*, for all main hardware keys* $k_{\texttt{HWm}}$*, for all pairs of tokens and token identifier* (id_{tk}, tk)*, for all predicates* P*, for all messages* $m \in \{0,1\}^{\ell(n)}$ *such that* $P(m) = 1$*, for* $\sigma = \texttt{Resp}_{\texttt{HW}}(k_{\texttt{HWm}}, label, id_{tk})$*, it holds that*

$$\Pr[\texttt{Process}(\texttt{WPAY}(id_{tk}, etk, \sigma, m), tk) = m] = 1,$$

where the probability is taken over compiling $\texttt{WPAY} \leftarrow_{\$} \texttt{Comp}_{\texttt{App}}(k_{\texttt{pay}}, k_{\texttt{HWs}}, P)$ *and encrypting the token* $(id_{tk}, etk) \leftarrow_{\$} \texttt{EnTok}(k_{\texttt{pay}}, (id_{tk}, tk))$*.*

4.1 Security of White-Box Payment Applications

We now specify security of a white-box payment application scheme WPAY. Correctness of WPAY ensures that when having access to the hardware, the application WPAY is useful to generate payment requests. In turn, hardware-binding security ensures that when not having access to the hardware, then the application becomes useless. In other words, in absence of the hardware, the adversary cannot generate new requests and does not learn anything about the content of the requests sent to the server. Thus, the desired security properties in the absence of the hardware are the following:

(1) Integrity of the requests transmitted from user to server.
(2) Confidentiality of the messages contained in the requests transmitted from user to server.

We capture both properties via the IND-WPAY security game, depicted in Figure 7. IND-WPAY starts with a setup phase where the relevant keys are sampled, first for the hardware (line 3 and 4) and then for the payment application (line 5). Then, WPAY is compiled (line 6). Note that we allow the adversary to choose the filter function P, modeling that security should hold for all possible filter functions. We also allow the adversary to choose the hardware *label*. In

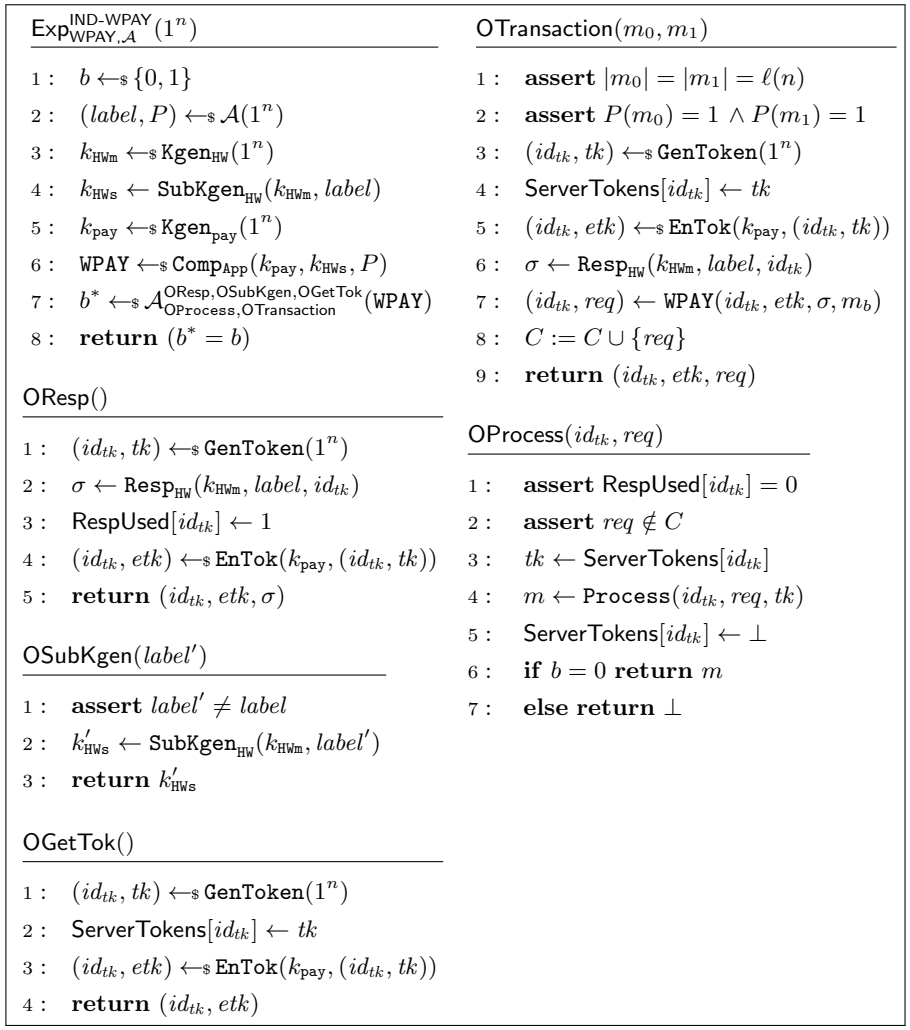


Fig. 7. $\mathsf{Exp}^{\mathsf{IND\text{-}WPAY}}_{\mathsf{WPAY},\mathcal{A}}(1^n)$ game capturing integrity and confidentiality.

practice, neither P nor *label* are adversarially chosen, but giving this ability to the adversary in the setup phase only makes our model stronger. Note that we consider our adversary as stateful, i.e., the adversary in the setup phase (line 2) shares state with the adversary that accesses oracles (line 7) in order to find out the secret bit b. As the adversary is a white-box adversary, it receives the compiled `WPAY` as input (line 7). We now turn to the explanation of the oracles.

Oracles OResp and OSubKgen model the adversary's hardware access. Upon querying OResp, a pair of token and its respective identifier are sampled at random. The value of the token identifier is used for generating the hardware value σ, and the oracle returns all three values. The adversary is thus able to run their own `WPAY` to generate a request message for that specific token, token id and hardware value.

OSubKgen lets the adversary observe hardware sub-key values. Such values could be values used by different applications or values from older versions of a payment application. To avoid trivial attacks, the adversary is not allowed to obtain the sub-key $k_{\texttt{HWs}}$ that was used for compiling the white-box application `WPAY` (line 6 in the main $\mathsf{Exp}^{\mathsf{IND\text{-}WPAY}}_{\mathsf{WPAY},\mathcal{A}}$ procedure).

Oracle OGetTok models generation and encryption of the tokens on the server's side and storing them in a list. We recall that a white-box adversary might be able to access the encrypted tokens stored on a user's phone and thus, OGetTok returns the encrypted token and its identifier to the adversary.

We now turn to the two remaining oracles that encode the desired security properties of confidentiality and integrity. We start with the transaction oracle OTransaction, which encodes the confidentiality property. On a conceptual level, OTransaction plays a similar role as the left-or-right encryption oracle in the security game for authenticated encryption (cf. Definition 2): It encrypts either the left or the right message, depending on whether the secret bit b is 0 or 1. Upon submitting the two messages to the OTransaction oracle, the oracle randomly samples a pair of a token and its respective identifier and the token value is encrypted via the `EnTok` algorithm. The oracle then generates the necessary hardware value σ based on the sampled token identifier, and then generates a request message based on the token identifier, the encrypted token, the value σ and one of the two messages submitted by the adversary. The oracle then saves the generated request on a list C and returns the token identifier, the encrypted token and the request message.

Note that the adversary is not able to use their own app to generate the same request message received by the transaction oracle. Namely, while the adversary is able to choose which message to encrypt with their own app, the token used for encrypting the message is chosen at random and only with negligible probability will both request messages look the same.

The process oracle OProcess encodes the integrity of the request messages similarly to the decryption oracle in authenticated encryption. The adversary can submit arbitrary values as long as those were not obtained from OTransaction (check if they are in the set C) or if they were not generated using an id_{tk} generated by the OResp oracle for generating the hardware value σ. The server

retrieves (line 3) the token tk corresponding to id_{tk} from the token list, decrypts (line 4) the request req with id_{tk} and tk using the $\texttt{Process}$ algorithm and deletes tk from the token list. Authenticity is modeled by only returning the message to the adversary if $b = 0$ and returning an error if $b = 1$. Thereby, the adversary is able to learn the secret bit b whenever the adversary is able to forge a fresh request.

Remark. Note that many useful properties are implied by our security definition. For instance, IND-WPAY security implies that the token values remain secret, unless the adversary queries the hardware on id_{tk}.

Definition 15 (IND-WPAY Security). *A hardware-bound white-box payment application scheme* WPAY *is said to be* IND-WPAY*-secure if all PPT adversaries $\mathcal{A}$ have negligible distinguishing advantage in the game* $\mathsf{Exp}^{\text{IND-WPAY}}_{\text{WPAY},\mathcal{A}}(1^n)$ *as specified in Figure 7.*

4.2 Construction of White-box Payment Scheme

We now construct a white-box payment scheme WPAY, which is IND-WPAY-secure (see Figure 7), assuming the IND-WKDF-security of a white-box key derivation function $\texttt{WKDF}$ (see Figure 5). We first give an overview of the algorithms of our construction. First, $\texttt{Kgen}_{\texttt{pay}}$ randomly samples a payment key $k_{\texttt{pay}}$ and the $\texttt{GenToken}$ algorithm randomly samples a token tk and its respective identifier id_{tk}. $\texttt{EnTok}$ encrypts a token tk in the following way: For each token with the identifier id_{tk}, it generates a key $\hat{k} \leftarrow \texttt{KDF}(k_{\texttt{pay}}, id_{tk})$, and uses an authenticated encryption scheme to encrypt the token using $\hat{k}$. That is, each token is encrypted using a different key. Note that each key is generated based on the same key $k_{\texttt{pay}}$, but based on a different context value id_{tk} since each token has a unique identifier.

The compilation algorithm $\texttt{Comp}_{\texttt{App}}$ of the payment scheme takes as input a payment key $k_{\texttt{pay}}$, a sub-key value $k_{\texttt{HWs}}$ as well as a message filtering predicate P. It uses the compilation algorithm $\texttt{Comp}$ of the $\texttt{WKDF}$ and runs it on $(k_{\texttt{pay}}, k_{\texttt{HWs}})$ to obtain a hardware-bound white-box program $\texttt{WKDF}$. It then runs indistinguishability obfuscation on a circuit $\mathsf{C}[\texttt{WKDF}, P]$ and returns the output of the obfuscation as $\texttt{WPAY}$. The circuit $\mathsf{C}[\texttt{WKDF}, P]$ and $\texttt{WPAY}$ provide the same functionality, but in $\texttt{WPAY}$, due to the layer of obfuscation, one should not be able to separate the different operations from each other (see the discussion below). $\mathsf{C}[\texttt{WKDF}, P]$ takes as input a pair (id_{tk}, etk) of an encrypted token and its token identifier as well as a hardware value σ and a message m. It first runs $\texttt{WKDF}$ on (id_{tk}, σ) to obtain an output $\hat{k}$, and recall that the security of $\texttt{WKDF}$ ensures that $\hat{k}$ is only a KDF (and not an error value) if σ is the correct hardware value that yields $\texttt{Check}_{\texttt{SW}}(k_{\texttt{HWs}}, id_{tk}, \sigma) = 1$. Thus, the hardware-binding of $\texttt{WPAY}$ is directly inherited from the hardware-binding of $\texttt{WKDF}$. Now, $\mathsf{C}[\texttt{WKDF}, P]$ uses $\hat{k}$ to decrypt the etk (line 5), checks whether $P(m) = 1$ (line 6) and, if so, encrypts the message m using the token tk as key (line 7-8) and returns the resulting ciphertext as a request message req along with the token identifier (line 10). Note that

$\underline{\mathtt{Kgen}_{\mathtt{pay}}(1^n)}$

1 : $k_{\mathtt{kdf}} \leftarrow\!\!\$\ \mathtt{Kgen}(1^n)$
2 : $k_{\mathtt{pay}} \leftarrow k_{\mathtt{kdf}}$
3 : **return** $k_{\mathtt{pay}}$

$\underline{\mathtt{Comp}_{\mathtt{App}}(k_{\mathtt{pay}}, k_{\mathtt{HWs}}, P)}$

1 : $\mathtt{WKDF} \leftarrow\!\!\$\ \mathtt{Comp}(k_{\mathtt{pay}}, k_{\mathtt{HWs}})$
2 : $\mathtt{WPAY} \leftarrow\!\!\$\ \mathtt{iO}(\mathsf{C}[\mathtt{WKDF}, P])$
3 : **return** WPAY

$\underline{\mathtt{Process}(id_{tk}, req, tk)}$

1 : $nc \leftarrow id_{tk}$
2 : $c_2 \leftarrow req$
3 : $m' \leftarrow \mathtt{Dec}_2(tk, c_2, nc)$
4 : **return** m'

$\underline{\mathtt{GenToken}(1^n)}$

1 : $tk \leftarrow\!\!\$\ \{0,1\}^n$
2 : $id_{tk} \leftarrow\!\!\$\ \{0,1\}^n$
3 : **return** (id_{tk}, tk)

$\underline{\mathtt{EnTok}(k_{\mathtt{pay}}, (id_{tk}, tk))}$

1 : $nc \leftarrow id_{tk}$
2 : $e \leftarrow id_{tk}$
3 : $\hat{k} \leftarrow \mathtt{KDF}(k_{\mathtt{pay}}, e)$
4 : $c_1 \leftarrow \mathtt{Enc}_1(\hat{k}, tk, nc)$
5 : $etk \leftarrow c_1$
6 : **return** (id_{tk}, etk)

$\underline{\mathsf{C}[\mathtt{WKDF}, P](id_{tk}, etk, \sigma, m)}$

1 : $e \leftarrow id_{tk}$
2 : $nc \leftarrow id_{tk}$
3 : $c_1 \leftarrow etk$
4 : $\hat{k} \leftarrow \mathtt{WKDF}(e, \sigma)$
5 : $tk \leftarrow \mathtt{Dec}_1(\hat{k}, c_1, nc)$
6 : **if** $|m| = \ell(n)$ **and** $P(m) = 1$
7 : $\quad c_2 \leftarrow \mathtt{Enc}_2(tk, m, nc)$
8 : $\quad req \leftarrow c_2$
9 : **else** $req \leftarrow \perp$
10 : **return** (id_{tk}, req)

Fig. 8. Construction of a WPAY scheme

in the construction, we need to avoid the use of randomness, since we cannot rely on the randomness being honestly generated. Thus, we use a nonce-based encryption scheme, and the nonce nc used for encryption of the token is not only retrieved (line 2) for decryption of the token (line 5), but also re-used for encrypting the request (line 7). Note that the nonce in etk is not malleable since we encrypt the tokens with an authenticated encryption scheme which provides ciphertext integrity.

Recall that WPAY returns not only req but also the token identifier id_{tk} of the token that was used to encrypt m. This is because the use of tk authenticates the user towards the server which, on its side, retrieves the token tk corresponding to id_{tk}, and runs the Process algorithm on (req, tk) which decrypts req with tk and returns the result.

Construction 4. *Based on two authenticated encryption schemes* $(\mathtt{Enc}_1, \mathtt{Dec}_1)$, $(\mathtt{Enc}_2, \mathtt{Dec}_2)$ *and the* WKDF *in Construction 3, we construct a* white-box payment scheme with hardware-binding $\mathsf{WPAY} = (\mathtt{Kgen}_{\mathtt{pay}}, \mathtt{GenToken}, \mathtt{EnTok}, \mathtt{Comp}_{\mathtt{App}}, \mathtt{Process})$ *as detailed in Figure 8.*

On the use of iO. As mentioned above, the compiler of the payment application in our construction applies indistinguishability obfuscation to the circuit describing the application (see line 2 of $\mathtt{Comp}_{\mathtt{App}}$ in Figure 8). We obfuscate WPAY with the following purposes. First we may wish to ensure the confidentiality of internal variables, such as the outputs of the WKDF and the raw value of the tokens. Second, by obfuscating the program we can also ensure that no operation can be separated from the rest, achieving thus a form of application binding. We note however that in the security proof provided for the theorem below, we do not prove any of these properties. Namely our IND-WPAY security model does

not capture any form of application binding for the WPAY and only captures confidentiality for tokens and other internal variables for the cases that an adversary does not have access to the determined hardware.

Thus, our construction directly derives its security from the WKDF and could also be proven secure even without using any form of obfuscation. However we choose to obfuscate WPAY still, given that in practice, one would usually apply one layer of obfuscation to the application in order to increase its robustness. We note however that our model could be extended in order to capture some type of application binding property, which could then be achieved by using iO on our construction, as long as the relevant primitives used within the applications are puncturable. For instance, one could challenge the adversary with providing the output $\hat{k}$ of the WKDF for a given context value e. Here, we could puncture the WKDF (which is itself constructed from puncturable PRFs) and also puncture the decryption algorithm as follows. For one ciphertext c^*, hardcode its corresponding token value tk^* and output tk^* every time c^* is provided as input. For all other ciphertexts, perform a normal decryption. Note that hardcoding the corresponding token value of a given ciphertext is possible, since the tokens are generated and encrypted in advance (see the GenToken and EnTok processes). Given both, the puncturable WKDF and the puncturable decryption program, we can effectively apply indistinguishability obfuscation and ensure that an adversary cannot separate the WKDF from the decryption and cannot extract any value $\hat{k}$.

Theorem 2. *Let* $(\mathtt{Enc}_1, \mathtt{Dec}_1)$ *and* $(\mathtt{Enc}_2, \mathtt{Dec}_2)$ *be two* AE*-secure symmetric encryption schemes, let* WKDF *be a* IND-WKDF*-secure white-box key derivation scheme, and let* iO *be an indistinguishability obfuscator for appropriate p-admissible samplers. Then the white-box payment scheme* WPAY *in Construction 4 is* IND-WPAY*-secure.*

In the following we provide a short sketch of the proof . The full details can be found on the full version of this paper [2].

Proof sketch. Let $\mathcal{A}$ be a PPT adversary. Let $\mathsf{Exp}^{\mathsf{IND\text{-}WPAY}}_{\mathcal{A},0}$ denote the IND-WPAY game with $b = 0$ hardcoded and let $\mathsf{Exp}^{\mathsf{IND\text{-}WPAY}}_{\mathcal{A},1}$ denote the IND-WPAY game with $b = 1$ hardcoded. We need to show that $\mathcal{A}$ has negligible distinguishing advantage, i.e., that the probability that $\mathcal{A}$ returns 1 in $\mathsf{Exp}^{\mathsf{IND\text{-}WPAY}}_{\mathcal{A},0}$ differs from the probability that $\mathcal{A}$ returns 1 in $\mathsf{Exp}^{\mathsf{IND\text{-}WPAY}}_{\mathcal{A},1}$ at most by a negligible amount.

On a high-level, the security proof proceeds as follows. (1) We replace all keys generated by the WKDF with random keys in OTransaction and OGetTok and reduce this game hop to the security of the WKDF. (2) In the OTransaction and OGetTok oracles, instead of encrypting the tokens tk with SE_1, we encrypt $0^{|tk|}$. To do so, we make a hybrid argument over the number of queries that the adversary makes to OTransaction and OGetTok and for each such query make a reduction to the AE security of SE_1. (3) In the OTransaction oracle, instead of encrypting m_0, we now encrypt m_1. At the same time, in the OProcess oracle, we do not perform decryptions anymore but rather answer all adversarial queries

by $\perp$. This proof again proceeds via a hybrid argument over the number of queries to $\mathsf{OTransaction}$ and $\mathsf{OGetTok}$, since one token value is generated in each of these calls and we need to reduce to the security to each of them. (4) We now de-idealize SE_1 again and encrypt the real token values. (5) We de-idealize the WKDF. The full technical details appear in the full version of the paper. □

Acknowledgments

Marc Fischlin has been [co-]funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) 251805230/GRK 2050. Christian Janson has been [co-]funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297.

References

1. E. Alpirez Bock, J. W. Bos, C. Brzuska, C. Hubain, W. Michiels, C. Mune, E. Sanfelix Gonzalez, P. Teuwen, and A. Treff. White-box cryptography: Don't forget about grey-box attacks. *Journal of Cryptology*, Feb 2019.
2. E. Alpirez Bock, C. Brzuska, M. Fischlin, C. Janson, and W. Michiels. Security reductions for white-box key-storage in mobile payments. Cryptology ePrint Archive, Report 2019/1014, 2019. `https://eprint.iacr.org/2019/1014`.
3. Android Developers. Build. Class Documentation, Last retrieved: October 2018. `https://developer.android.com/reference/android/os/Build`.
4. Android Developers. Keystore. Class Documentation, Last retrieved: October 2018. `https://developer.android.com/reference/java/security/KeyStore`.
5. Android Developers. Telephony manager. Class Documentation, Last retrieved: October 2018. `https://developer.android.com/reference/android/telephony/TelephonyManager`.
6. D. Apon, Y. Huang, J. Katz, and A. J. Malozemoff. Implementing cryptographic program obfuscation. Cryptology ePrint Archive, Report 2014/779, 2014. `http://eprint.iacr.org/2014/779`.
7. G. Avoine, M. A. Bingöl, I. Boureanu, S. Capkun, G. P. Hancke, S. Kardas, C. H. Kim, C. Lauradoux, B. Martin, J. Munilla, A. Peinado, K. B. Rasmussen, D. Singelée, A. Tchamkerten, R. Trujillo-Rasua, and S. Vaudenay. Security of distance-bounding: A survey. *ACM Comput. Surv.*, 51(5):94:1–94:33, 2019.
8. S. Banik, A. Bogdanov, T. Isobe, and M. B. Jepsen. Analysis of software countermeasures for whitebox encryption. *IACR Trans. Symm. Cryptol.*, 2017(1):307–328, 2017.
9. M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 531–545. Springer, Heidelberg, Dec. 2000.
10. M. Bellare, I. Stepanovs, and B. Waters. New negative results on differing-inputs obfuscation. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 792–821. Springer, Heidelberg, May 2016.
11. A. Biryukov, C. Bouillaguet, and D. Khovratovich. Cryptographic schemes based on the ASASA structure: Black-box, white-box, and public-key (extended abstract). In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 63–84. Springer, Heidelberg, Dec. 2014.

12. T. Bocek, C. Killer, C. Tsiaras, and B. Stiller. An nfc relay attack with off-the-shelf hardware and software. In R. Badonnel, R. Koch, A. Pras, M. Drašar, and B. Stiller, editors, *Management and Security in the Age of Hyperconnectivity*, pages 71–83, Cham, 2016. Springer International Publishing.
13. E. A. Bock, A. Amadori, J. W. Bos, C. Brzuska, and W. Michiels. Doubly half-injective PRGs for incompressible white-box cryptography. In M. Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 189–209. Springer, Heidelberg, Mar. 2019.
14. E. A. Bock, A. Amadori, C. Brzuska, and W. Michiels. On the security goals of white-box cryptography. *IACR TCHES*, 2020(2):327–357, 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8554`.
15. A. Bogdanov and T. Isobe. White-box cryptography revisited: Space-hard ciphers. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 1058–1069. ACM Press, Oct. 2015.
16. A. Bogdanov, T. Isobe, and E. Tischhauser. Towards practical whitebox cryptography: Optimizing efficiency and space hardness. In J. H. Cheon and T. Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 126–158. Springer, Heidelberg, Dec. 2016.
17. D. Boneh and B. Waters. Constrained pseudorandom functions and their applications. In K. Sako and P. Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, Dec. 2013.
18. S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. White-box cryptography and an AES implementation. In K. Nyberg and H. M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270. Springer, Heidelberg, Aug. 2003.
19. S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. A white-box DES implementation for DRM applications. In J. Feigenbaum, editor, *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002*, volume 2696 of *LNCS*, pages 1–15. Springer, 2003.
20. T. Cooijmans, J. de Ruiter, and E. Poll. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 11–20. ACM, 2014.
21. cybercrypt. Ches 2019 capture the flag challenge - the whibox contest - edition 2, 2019. `https://www.cyber-crypt.com/whibox-contest/`.
22. C. Delerablée, T. Lepoint, P. Paillier, and M. Rivain. White-box security notions for symmetric encryption schemes. In T. Lange, K. Lauter, and P. Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 247–264. Springer, Heidelberg, Aug. 2014.
23. ECRYPT. Ches 2017 capture the flag challenge - the whibox contest, 2017. `https://whibox.cr.yp.to/`.
24. 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`.
25. P.-A. Fouque, P. Karpman, P. Kirchner, and B. Minaud. Efficient and provable white-box primitives. In J. H. Cheon and T. Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 159–188. Springer, Heidelberg, Dec. 2016.
26. L. Francis, G. Hancke, K. Mayes, and K. Markantonakis. Practical relay attack on contactless transactions by using NFC mobile phones. Cryptology ePrint Archive, Report 2011/618, 2011. `http://eprint.iacr.org/2011/618`.
27. L. Goubin, P. Paillier, M. Rivain, and J. Wang. How to reveal the secrets of an obscure white-box implementation. Cryptology ePrint Archive, Report 2018/098, 2018. `https://eprint.iacr.org/2018/098`.

28. G. Hancke. A practical relay attack on iso 14443 proximity cards. Technical report, 2005.
29. H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, Aug. 2010.
30. J. Kwon, B. Lee, J. Lee, and D. Moon. FPL: White-box secure block cipher using parallel table look-ups. In S. Jarecki, editor, *CT-RSA 2020*, volume 12006 of *LNCS*, pages 106–128. Springer, Heidelberg, Feb. 2020.
31. Mastercard. Mastercard mobile payment sdk, 2017. `https://developer.mastercard.com/media/32/b3/b6a8b4134e50bfe53590c128085e/mastercard-mobile-payment-sdk-security-guide-v2.0.pdf`.
32. M. Rivain. White-box cryptography. Presentation CARDIS 2017, 2017. `http://www.matthieurivain.com/files/slides-cardis17.pdf`.
33. P. Rogaway. Authenticated-encryption with associated-data. In V. Atluri, editor, *ACM CCS 2002*, pages 98–107. ACM Press, Nov. 2002.
34. P. Rogaway. Nonce-based symmetric encryption. In B. K. Roy and W. Meier, editors, *FSE 2004*, volume 3017 of *LNCS*, pages 348–359. Springer, Heidelberg, Feb. 2004.
35. A. Sahai and B. Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In D. B. Shmoys, editor, *46th ACM STOC*, pages 475–484. ACM Press, May / June 2014.
36. E. Sanfelix, J. de Haas, and C. Mune. Unboxing the white-box: Practical attacks against obfuscated ciphers. Presentation at BlackHat Europe 2015, 2015. `https://www.blackhat.com/eu-15/briefings.html`.
37. Smart Card Alliance Mobile and NFC Council. Host card emulation 101. white paper, 2014. `http://www.smartcardalliance.org/downloads/HCE-101-WP-FINAL-081114-clean.pdf`.
38. B. Wyseur. White-box cryptography: Hiding keys in software. 2012. `http://www.whiteboxcrypto.com/research.php`.