Verifiable Verification in Cryptographic Protocols

Marc Fischlin[1] Felix Günther[2]

[1] Cryptoplexity, Technische Universität Darmstadt, Darmstadt, Germany
[2] Department of Computer Science, ETH Zürich, Zürich, Switzerland
marc.fischlin@cryptoplexity.de mail@felixguenther.info

August 10, 2023

Abstract. Common verification steps in cryptographic protocols, such as signature or message authentication code checks or the validation of elliptic curve points, are crucial for the overall security of the protocol. Yet implementation errors omitting these steps easily remain unnoticed, as often the protocol will function perfectly anyways. One of the most prominent examples is Apple's goto fail bug where the erroneous certificate verification skipped over several of the required steps, marking invalid certificates as correctly verified. This vulnerability went undetected for at least 17 months.

We propose here a mechanism which supports the detection of such errors on a cryptographic level. Instead of merely returning the binary acceptance decision, we let the verification return more fine-grained information in form of what we call a confirmation code. The reader may think of the confirmation code as disposable information produced as part of the relevant verification steps. In case of an implementation error like the goto fail bug, the confirmation code would then miss essential elements.

The question arises now how to verify the confirmation code itself. We show how to use confirmation codes to tie security to basic functionality at the overall protocol level, making erroneous implementations be detected through the protocol not functioning properly. More concretely, we discuss the usage of confirmation codes in secure connections, established via a key exchange protocol and secured through the derived keys. If some verification steps in a key exchange protocol execution are faulty, then so will be the confirmation codes, and because we can let the confirmation codes enter key derivation, the connection of the two parties will eventually fail. In consequence, an implementation error like goto fail would now be detectable through a simple connection test.

A shortened version of this paper appears in the proceedings of the *30th ACM Conference on Computer and Communications Security (CCS 2023)*. This is the full version.

Contents

1 Introduction

Programming errors in software code keep on reoccurring. While detecting such bugs is already a tedious task in general, the situation gets exacerbated in the context of security and cryptographic code in particular. The reason is that cryptographic implementations are extremely brittle: bugs can be entirely unnoticeable from a functionality point of view, yet lead to a complete failure of security. Such bugs are most hidden when cryptography is used within a single implementation; a prominent recent example being a faulty key rotation mechanism at Amazon AWS using all-zero keys for TLS session ticket encryption [HNM+23], leading to a perfectly functional and simultaneously perfectly insecure cryptosystem.

But even when a cryptographic implementation is interoperating with others, implementation mistakes can go unnoticed, for as long as months or years. Some of the best-known examples include Apple's goto fail bug [Pou14] where some certificate validation steps had been skipped due to a superflous `goto fail;` code line, causing even invalid certificates to pass verification. A similar case of erroneous certificate validation happened in the GnuTLS library [Tea14] where a procedure could return negative values, but only zero values were interpreted as an error by the calling procedure. As yet another example, such misinterpretation of parameters also caused programmers to use restricted certificate validation in cURL, as pointed out in [GIJ+12], when setting the parameters unintentionally to `true` (interpreted as 1) instead of integer value 2 for a full certificate check, including matching the name in the certificate with the requested name.

Similarly widely exploited and remaining issues arise from missing soundness checks of cryptographic parameters. The most prominent examples are presumably the small-subgroup attacks on discrete logarithms [vW96, AV96, LL97] where the verifier does not check that the received values belong to the sufficiently large subgroup. The attack keeps on resurfacing in various forms, e.g., [ABD+15, VAS+17, AMPS18], and can also be applied to RSA-based settings, e.g., see [AHMP23] for a recent attack on the MEGA encryption system. Another nasty variation of this attack is the fixed-coordinate attack on Bluetooth [BN19].

1.1 Tying Security to Functional Correctness

One may hope that testing of cryptographic code, e.g., via NIST's Cryptographic Algorithm Validation Program [oST23a], or verified implementations of cryptographic protocols (such as the miTLS implementation of TLS [BFK+13]) may mitigate the problem. This, however, does not match the community's experience with the persistence of such weaknesses. As Adam Langley put it in connection with the Poodle attack [Lan14b]: "the Internet is vast and full of bugs." We are not going to discuss potential reasons for this, but instead ask ourselves whether it is inherent that cryptographic implementation errors remain unnoticeable. Put differently, what if we could make crypto bugs surface through inflicted, noticeable errors in a program's functionality? Such functional errors would in turn be easily detected already during basic interoperability testing of a cryptographic implementation, leading to an early mitigation of the underlying bug in the cryptographic code.

Our approach is inspired by a talk by Nadia Heninger [Hen19] at the Workshop on Attacks in Cryptography 2 (WAC2), affiliated with Crypto 2019. In her talk she proposed to minimize the damage caused by inevitable human errors by "tying security to basic functionality." Our work is perfectly aligned with this idea. If the implementations enable attacks like the goto fail bug or the fixed-coordinate attack on Bluetooth, then the goal is to make this visible via the functional behavior of the cryptographic protocol. More concretely, an example —which we will explore in more detail— is a secure connection protocol in which the two parties *fail* to connect if verification steps are erroneous.

While the idea of linking functionality to security is applicable in general, our focus here lies on cryptographic verification steps in protocols, as discussed above. In particular, we assume that the unwary

implementer of the verification step may have introduced accidental errors, such as skipping some steps. We do not protect against fully malicious behavior like fault injection attacks [BDL97, BS97]. Our approach focuses on detecting bugs by simple interoperability testing, where we presume that the faulty implementation tries to establish a connection with another, sound implementation.

1.2 Enter Confirmation Codes

Recall the issue with the goto fail bug: the verification routine simply jumped over some steps and always claimed verification was successful. Suppose now that, instead of having an error-prone decision bit returned by the verification step, we would have a more precise description of which of the essential steps of the full verification have actually been carried out. One observation is that such verification steps usually compute some intermediate values and base their decision on comparing these values. One option, which we also follow here, is thus to gather (some relevant subset of) these intermediate values in what we call a "*confirmation code*". Instead of returning merely the decision, the verification procedure will then also hand back this confirmation code. Implementation errors like skipping some verification steps as in the goto fail bug, misinterpretation of return values as in GnuTLS or parameters as in cURL, or reading invalid inputs as in the Bluetooth attack, are hence surfaced through (changes in) the confirmation code.

Note that with the above approach the computation of the confirmation code basically comes for free as part of the actual verification. A side condition, when envisioning the usage of the same confirmation code by two parties, is that the creator (e.g., the signer of a message or the sampler of an elliptic curve point) should also be able to derive the same confirmation code when performing its calculation. Ideally, this should also come without significant overhead for the creator, e.g., without having to run verification itself.

The next step is to take advantage of the extra information gathered in the confirmation code. A naive attempt would be to have two communicating parties check whether they derived the same value. This however simply introduces another layer of verification, likewise prone to implementation errors. Instead, we will use the confirmation code implicitly in the overall protocol. This enables verifiability of the verification steps of the cryptographic protocol, on a cryptographic level.

Since the use of confirmation codes is specific to the type of protocol, in this work we focus on secure connection establishment through key exchange protocols and consider confirmation codes derived through usual verification steps in such settings. This includes signature and MAC verifications or membership tests for group elements. In a key exchange protocol, we can now accumulate the confirmation codes $c_1, \ldots, c_m$ produced by its components and include them in a single, additional key derivation step: we use the established key K in a key derivation function KDF, with the collection of confirmation codes as the label, $K' \leftarrow \mathsf{KDF}(K, c_1 \| \ldots \| c_m)$. The resulting key K' is the final key of the augmented key exchange protocol. Note that, if the confirmation codes on either side differ (e.g., due to an implementation error), then the parties in the key exchange protocol will now derive different session keys as well. This in turn makes the subsequent connection fail, leading to a clearly noticeable functionality error.

1.3 A Cryptographic Task

What is at hand now is the cryptographic task to make precise what properties confirmation codes should have and what their effects on protocols employing them should be. We approach it as such, through formal definitions, constructions, and security proofs, always keeping a focus on practice by aiming for simple, low-overhead, and deployable solutions for existing cryptographic settings. Concretely, our contributions are as follows.

1.3.1 Defining confirmation codes and their goal

We begin, in Section 2, by generalizing classical verification algorithms like signatures, MACs, or parameter validation under a common syntax for verification schemes, involving algorithms for creating and verifying tokens (e.g., signatures, MACs, ...) for objects (e.g., messages, elliptic curve points, ...). We then augment these classical schemes by adding the concept of confirmation codes to our generic syntax, allowing the creation and verification algorithms to output a confirmation code (in form of a generic vector $\vec{c}$), in addition to the usual token resp. decision bit output.

The next step is to define what security property we expect from confirmation codes, in order to make sure that verification errors translate into changes in confirmation codes which can then be noticed in a higher-level protocol. At first glance, this might seem unachievable: since confirmation codes are computable by the verifier, in the case for example of signatures even from fully public knowledge, a malicious implementation may simply ignore the actual verification procedure and just make sure to compute the confirmation code correctly. We cannot protect against such malicious implementations, and the crucial point here is that we actually do not intend to. Our aim is rather to safeguard the benign implementer from *accidentally* introducing an error in the verification implementation (without intend to cover up their mistake by retrofitting confirmation codes).

We model such accidental but benign faults in verification steps cryptographically via a non-adaptive adversary. More precisely, we let the non-adaptive adversary decide beforehand which parts of the confirmation code to modify, reflecting the idea that these are the erroneous parts in the program which are put there independently of the actual cryptographic data.[1] Quantifying over all non-adaptive adversaries means to capture all such potential programming errors. This leads to a notion of *unpredictability* of confirmation codes, which we formalize in Section 3, meaning that the non-adaptively placed errors cannot let the faulty confirmation code accidentally coincide with the actual confirmation code.

1.3.2 Adding confirmation codes to practical verification schemes

The next task is then to show that common cryptographic schemes support such unpredictable confirmation codes. In Section 4, we discuss and prove secure concrete instances for the RSA-PSS signature scheme, the HMAC message authentication code, and the validation of elliptic curve points. In each case, our constructions' goal is minimal overhead: we base the confirmation codes on intermediate values already computed in the original schemes, allowing to add these codes in practice with little to no additional computation and zero communication effort.[2]

1.3.3 Making connections fail noticeably

Finally, we apply the idea of verifiable verification through tying security to functionality to the setting of secure connection establishment via key exchange protocols. In Section 5, we show that via a generic transform adding a single extra key derivation step to the protocol, confirmation codes can be bound to the derived session key in such a way that implementation errors in verification (like Apple's goto fail bug) break basic functionality: a verification error leading to non-matching confirmation codes now translates to non-matching keys, and those in turn to connection establishment failing, which the implementer will immediately discover during a simple connection test.

[1] We do allow dependency on the public key though, e.g., to capture the key being hardcoded into the program.

[2] We acknowledge that integrating confirmation codes into deployed libraries also requires system-level efforts, which are beyond the scope of this work. For example, confirmation codes may be surfaced via additional API calls for verification, allowing backwards compatibility for "classical verification" usage. Applications can then leverage this interface to tie in confirmation codes in such a way that mismatches are surfaced through failing functionality.

We formalize this through measuring correct functionality of the key exchange under faulty verification implementations, again capturing accidental implementation errors as modifications by a non-adaptive adversary. We then prove that confirmation-code unpredictability of verification schemes ensures that a key exchange protocol with our transform applied will fail (via keys with high probability not matching) whenever there are faults in the implementation of those verification schemes. Finally, we argue that the so-augmented key exchange protocol is as secure as the original protocol. While seemingly intuitive, showing this formally turns out to be non-trivial due to our transform —purposefully— collecting the confirmation codes from the verification steps in the original key exchange protocol in a black-box manner, without access to the verification keys, requiring the reduction to rely on publicly computable codes.

1.4 Related Work

While we are not aware of prior attempts to cryptographically tie security to basic functionality, our work both in concepts and techniques connects to various other areas.

Proof-carrying code. Proof-carrying code, introduced by Necula and Lee [NL97, Nec97], should allow to check if a program from an untrusted source satisfies certain safety properties. For this, the verifier provides a safety policy and the code producer creates an additional safety proof which the verifier can use to check efficiently that the program obeys the policy. One may view our approach with confirmation codes as an implementation of this idea, but where the programmer is usually simultaneously producer and verifier and where verification of the safety proof is done implicitly. Indeed, Necula [Nec97] in his original paper argued that cryptography mechanisms rather implement trust relationships between parties and do not necessarily allow to detect coding errors. Our solution bridges to the possibility that cryptography can also be used to check that parts of the code are sound.

Cryptographic Algorithm Validation Program. NIST runs the Cryptographic Algorithm Validation Program (CAVP) [oST23a] to support tests for recommended algorithms. This includes common block ciphers such as AES but also public-key schemes like DSA. Recently, NIST added hash functions to the suite of algorithms [MC20], due to vulnerabilities found in Apple's CoreCrypto library. The idea of the program is to specify tests —including also false inputs— and to have a Cryptographic & Security Testing (CST) lab check the validity. This blends in into other testing methods for cryptographic algorithms, and yet does not seem to be able to capture all potential bugs.

Verifiable computation. The notion of verifiable computation dates back to a work of Gennaro et al. [GGP10] and aims at verifying the results of outsourced computations. The idea of checking such computations already appears in earlier works, especially in [BFLS91], and boils down to be able to check if a derived value y coincides with some expected value $F(x)$ for input x. The approach is to let the verifier transform the input (e.g., by encrypting it under a fully homomorphic encryption scheme as in [GGP10]), then letting the computing party perform the computation, and allowing the verifier to check (interactively or non-interactively) that the result is correct. The most promising direction is to deploy efficient zero-knowledge proofs where the verifying party augments the input by paramters for such a proof system, and the computing party eventually prepares a proof of correctness of the result [GKR08].

While our approach here is related to verifiable computations in the sense that one should be able to check that steps have been carried out, it is fundamentally different, though. In our setting we do not outsource the computation to another party but run the verification locally with potential errors in the code. We neither check for the correctness of result of the computation—in our case the decision bit—but instead augment the output by more information. This output, however, needs to be synchronized with the other party whose data we verify, e.g., both signer and verifier need to compute the same confirmation

code. Finally, verifying verifications such as for signatures should be lightweight, such that transforming inputs via fully homomorphic encryption or using zero-knowledge proofs to verify appear to be prohibitively expensive.

Progressive verification. The notion of progressive verification [Fis03, Fis04] of cryptographic primitives relates the error probability in verification to the actual time spent on verification: the more verification steps are carried out, the more reliable is the result. Talen and Vergnaud [TV21] recently showed that progressive verification is possible with ECDSA, RSA, and the GPV lattice-based signature scheme. Boschini et al. [BFP22] discuss solutions for further lattice-based and multivariate signature schemes. Le et al. [LKK19] considered the related concept of flexible signature schemes and showed than Lamport's one-time signature scheme is flexible.

While the goal of progressive verification, relating time investment to confidence, is different from the one of verifiable verification for detecting programming errors, in some cases the techniques coincide. To progressively verify hash chains, for example, Fischlin [Fis04] uses intermediate outputs when iterating through the chain, resembling the approach we use here to put results of the verification into the confirmation code. However, other approaches in [Fis03, LKK19, TV21, BFP22] rather use random ordering of sub-steps to achieve progressiveness.

Simultaneous verification. In the context of hybrid (post-quantum and classical) signature schemes, Bindel and Hale [BH23] informally discuss a concept called "simultaneous verification", which asks that, when combining two signature schemes, the joint verifier may not "quit" before both sub-verification processes completed. Confirmation codes could be seen as one approach to ensure that the execution of a verification algorithm completes correctly.

Checksums. Our security goal for confirmation codes, unpredictability, aims to capture a non-malicious implementer introducing verification errors and, implicitly, relies on honestly created tokens to have certain entropy to make them unpredictable. This conceptually resembles non-cryptographic checksums like cyclic redundancy checks, which likewise aim to detect unintentional flaws (in data) and which should catch, e.g., random bit flips.

Attacks skipping verification. Related in their effect of bypassing proper verification are protocol-level attacks that make a protocol skip verification entirely, like the "Early ChangeCipherSuite" attack on OpenSSL [Lan14a]. While those would also trigger non-matching confirmation codes in our setting, our goal is not to defend against malicious adversaries, but to make unintentional implementation errors that always skip some verification steps become noticeable.

2 Defining Verifiable Verification

Our concept of verifiable verification applies to different types of verification schemes like signatures, MACs, or validity checks. We hence first introduce a common notation for verification schemes, then introduce verifiable verification on top of it. Before we do so, let us define some basic notation first.

Notation. We write a bit as $b \in \{0,1\}$, a (bit) string as $s \in \{0,1\}^*$ with $|s|$ indicating its (binary) length, and $s \in \{0,1\}^{l*}$ for a string whose length is a multiple of l. By $s\|t$ we denote the concatenation and by $s \oplus t$ the bit-wise XOR of two strings s, t. For a vector v, we write $v[i]$ for the i-th entry of v.

We write $y \leftarrow x$ for assignments and $z \xleftarrow{\$} Z$ for sampling z uniformly at random from a finite set Z. By $y \xleftarrow{\$} A^{\mathcal{O}}(x)$ and $y \leftarrow A^{\mathcal{O}}(x)$ we denote the output y of a randomized resp. deterministic algorithm

A run on input x with oracle access to $\mathcal{O}$, where the probability is over A's internal randomness. For a security experiment Expt, we write $\mathsf{Expt} = 1$ for the experiment returning 1.

2.1 Classical Verification Schemes

We distinguish between three common cases of verification: signature verification (where the verifier uses a public key matching the signer's secret key), message authentication (where the verifier uses the same key as the signer), and validity verification such as checking validity of elliptic curve points (where no keys are involved). Generalizing the three, we thus use an abstract syntax consisting of:

- A key generation algorithm outputting a triple (ck, vk, pk) of a secret creation key, a secret verification key, and a public key; the latter will be given to the adversary in security games. For signatures ck is the signing key, vk is empty, and pk is the public verification key, whereas for MACs we will set $ck = vk$ to be the secret key and pk to be empty. For validity checks $ck = vk = \bot$ are empty and pk may for example contain public parameters (like the description of an elliptic curve).
- A creation algorithm which takes as input a creation key ck, a public key pk, and some input in, e.g., an input message $in = m$ for signatures and MACs. The algorithm outputs a (cryptographic) token tok together with an object obj which should be verified, e.g., for signatures and MACs the object $obj = m$ is the input message itself.
- A verification algorithm taking a verification key vk, a public key pk, an object obj, and a token tok, returning a decision bit d.

We note that different, and equivalent, choices in syntax are possible; we opt for the distinction of secret and public keys as above, with public keys being inputs to both creation and verification algorithms, for clear indication of what are secret and what are public values in a scheme.

Definition 2.1 (Verification scheme)**.** *A* (classical) verification scheme $\mathsf{V} = (\mathsf{KGen}, \mathsf{Create}, \mathsf{Vrfy})$ *with associated spaces for inputs, tokens, and objects $\mathcal{I}$, $\mathcal{T}$, resp. $\mathcal{O}$ consists of three efficient algorithms defined as follows.*

- $\mathsf{KGen}() \xrightarrow{\$} (ck, vk, pk)$. *This probabilistic algorithm outputs a secret creation key ck, a secret verification key vk, as well as a public key pk. (Either key may be also set to $\bot$.)*
- $\mathsf{Create}(ck, pk, in) \xrightarrow{\$} (obj, tok)$. *For a creation key ck, a public key pk, and an input $in \in \mathcal{I}$, this (possibly) probabilistic algorithm outputs an object $obj \in \mathcal{O}$ and a token $tok \in \mathcal{T}$.*
- $\mathsf{Vrfy}(vk, pk, obj, tok) \to d$. *On input a verification key vk, a public key pk, an object obj, and a token tok, this deterministic algorithm outputs a decision bit $d \in \{0, 1\}$ (where $d = 1$ indicates validity of the token for the object).*

Let $(ck, vk, pk) \xleftarrow{\$} \mathsf{KGen}()$. We call V symmetric *if always $ck = vk \neq \bot$,* asymmetric *if always $vk = \bot$, and* public *if always $ck = vk = \bot$.*

Correctness. We say that a verification scheme V is *correct* if for any $in \in \mathcal{I}$ it holds that

$$\Pr\left[d = 1 \;\middle|\; (ck, vk, pk) \xleftarrow{\$} \mathsf{KGen}(); (obj, tok) \xleftarrow{\$} \mathsf{Create}(ck, pk, in); d \leftarrow \mathsf{Vrfy}(vk, pk, obj, tok)\right] = 1.$$

Canonical representation of signatures and MACs. We can canonically capture signature and MAC schemes as (asymmetric, resp. symmetric) verification schemes, as shown in Figure 1. See Appendix A.1 for the standard definitions of signature and MAC schemes.

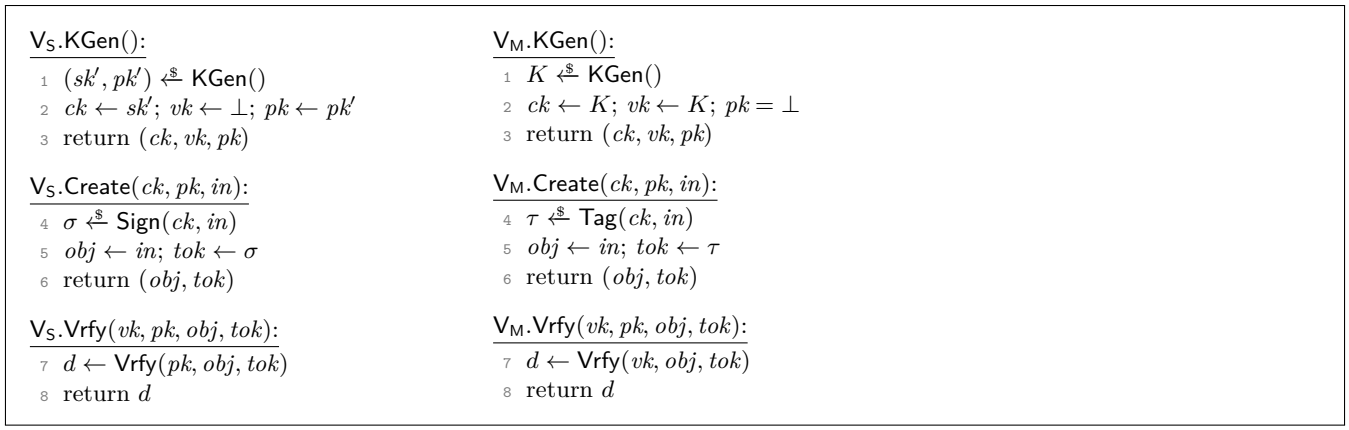

Figure 1: Canonical representation of (left) a signature scheme $\mathsf{S} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vrfy})$ and (right) a MAC scheme $\mathsf{M} = (\mathsf{KGen}, \mathsf{Tag}, \mathsf{Vrfy})$ as verification schemes $\mathsf{V_S}$, resp. $\mathsf{V_M}$.

2.2 Verification Schemes with Confirmation

To introduce the concept of confirmation in verification procedures, we augment the outputs of the creation algorithm and the verification algorithm by vectors of confirmation codes. This leads to the following syntax.

Definition 2.2 (Verification scheme with confirmation)**.** *A* verification scheme with confirmation $\mathsf{VC} = (\mathsf{KGenC}, \mathsf{CreateC}, \mathsf{VrfyC})$ *with associated spaces for inputs, tokens, objects, and confirmation codes $\mathcal{I}$, $\mathcal{T}$, $\mathcal{O}$, resp. $\mathcal{C}$ consists of three efficient algorithms defined as follows.*

- $\mathsf{KGenC}() \xrightarrow{\$} (ck, vk, pk)$*. This probabilistic algorithm outputs a secret creation key ck, a secret verification key vk, as well as a public key pk. (Either key may be also set to $\perp$.)*
- $\mathsf{CreateC}(ck, pk, in) \xrightarrow{\$} (obj, tok, \vec{c})$*. On input a creation key ck, a public key pk, and an input $in \in \mathcal{I}$, this (possibly) probabilistic algorithm outputs an object $obj \in \mathcal{O}$, a token $tok \in \mathcal{T}$, and a vector of confirmation codes $\vec{c} \in \mathcal{C}^*$.*
- $\mathsf{VrfyC}(vk, pk, obj, tok) \rightarrow (d, \vec{c})$*. On input a verification key vk, a public key pk, an object obj, and a token tok, this deterministic algorithm outputs a decision bit $d \in \{0, 1\}$ (where $d = 1$ indicates validity of the token for the object) and a vector of confirmation code $\vec{c} \in \mathcal{C}^* \cup \{\perp\}$ (where $\vec{c} = \perp$ iff $d = 0$).*

Correctness. We say that a verification scheme with confirmation, VC, is *correct* if for any $in \in \mathcal{I}$ it holds that

$$\Pr\left[d = 1 \wedge \vec{c}_c = \vec{c}_v \;\middle|\; \begin{array}{l}(ck, vk, pk) \xleftarrow{\$} \mathsf{KGenC}(); (obj, tok, \vec{c}_c) \xleftarrow{\$} \mathsf{CreateC}(ck, pk, in); \\ (d, \vec{c}_v) \leftarrow \mathsf{VrfyC}(vk, pk, obj, tok)\end{array}\right] = 1.$$

Augmenting classical verification schemes with confirmation codes. Observe that any classical verification scheme can canonically be written as one with confirmation, by simply letting $\mathsf{CreateC}$ and VrfyC output empty confirmation code vectors $\vec{c} = ()$.

In practice, we are interested in *augmenting* a classical verification scheme V to also output (non-empty) confirmation codes, in such a way that the other outputs remain unmodified, for compatibility reasons. We call a verification scheme with confirmation VC a *confirmation-augmented version* of V, formally defined as follows.

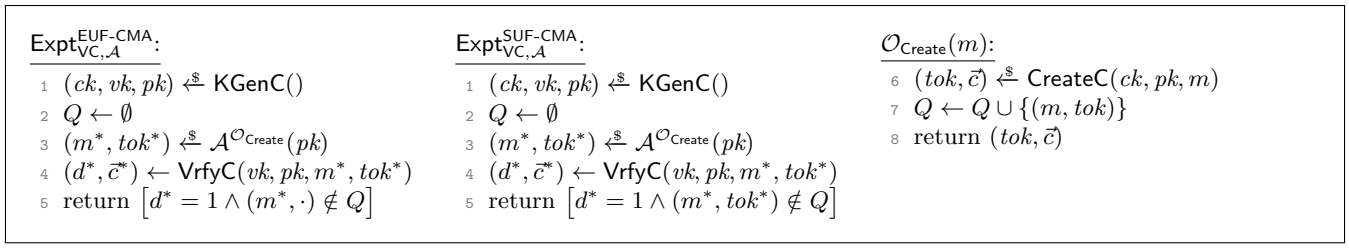

Figure 2: Security experiments for *existential and strong unforgeability under chosen-message attacks* (EUF-CMA, resp. SUF-CMA) for verification schemes with confirmation. We write $(a, \cdot) \notin Q$ if $\nexists b$ s.t. $(a, b) \in Q$.

Definition 2.3 (Confirmation-augmented verification scheme)**.** *Let* $\mathsf{V} = (\mathsf{KGen}, \mathsf{Create}, \mathsf{Vrfy})$ *be a (classical) verification scheme and* $\mathsf{VC} = (\mathsf{KGenC}, \mathsf{CreateC}, \mathsf{VrfyC})$ *be a verification scheme with confirmation with the same spaces for inputs* $\mathcal{I}$*, tokens* $\mathcal{T}$*, objects* $\mathcal{O}$*, as well as random coins* $\mathcal{R}$ *and* $\mathcal{R}'$ *for* Create *resp.* CreateC*. We say that* VC *is a* confirmation-augmented version *of* V *if*

- $\mathsf{KGenC} = \mathsf{KGen}$*,*
- $\mathcal{R} = \mathcal{R}'$*,*
- *for all keys generated as* $(ck, vk, pk) \xleftarrow{\$} \mathsf{KGen}()$*, inputs* $in \in \mathcal{I}$*, and random coins* $r \in \mathcal{R}$*, letting* $(obj, tok, \vec{c}) \leftarrow \mathsf{CreateC}(ck, pk, in; r)$ *and* $(obj', tok') \leftarrow \mathsf{Create}(ck, pk, in; r)$*, it holds that* $obj = obj'$ *and* $tok = tok'$*, and*
- *for all keys generated as* $(ck, vk, pk) \xleftarrow{\$} \mathsf{KGen}()$*, objects* $obj \in \mathcal{O}$*, and tokens* $tok \in \mathcal{T}$*, letting* $(d, \vec{c}) \leftarrow \mathsf{VrfyC}(vk, pk, obj, tok)$ *and* $d' \leftarrow \mathsf{Vrfy}(vk, pk, obj, tok)$*, it holds that* $d = d'$*.*

Example. Note that the above definition coincides with the common notions for signature and MAC schemes with regard to messages $m = in = obj$ and tokens (the latter capturing signatures resp. MAC tags), augmented with confirmation codes. We have seen in Figure 1 how signature and MAC schemes can canonically be represented as verification schemes, and hence augmented with verification codes. Here, we briefly discuss how validity checks for randomly generated elliptic curve points fit in, albeit we revisit the approach in more detail later in Section 4.3. In the case of elliptic curves we assume that key generation creates the description of an elliptic curve, with equation $x^3 + ax + b = y^2$ over a prime field $\mathbb{F}$. (The reader may think of a concrete curve like the FIPS 186-4 curve P-256.) The creation algorithm is supposed to output a random curve point $obj = (x, y) \in \mathbb{F}^2$ and ignore the input in. The token itself tok is empty in this case. We neglect the confirmation code here, since we discuss this in more detail in light of the security requirements of such values later. The verification algorithm, on receiving an object (x, y), checks that indeed $x^3 + ax + b = y^2$ over $\mathbb{F}$ and outputs this result as its decision bit.

Unforgeability. The classical (existential and strong) unforgeability notions for signatures and MACs (given in Appendix A.1 for completeness) generalize to verification schemes without change beyond accounting for confirmation codes being output. These unforgeability notions are not suitable for validity checks and other public verification schemes without secret creation key ($ck = \perp$), as the adversary can then simply forge arbitrary "tokens" (which the verifier accepts by correctness). We can nonetheless define the notion for verification schemes in general, as follows.

Definition 2.4 (Existential and strong unforgeability of verification schemes with confirmation)**.** *Let* VC *be a verification scheme with confirmation and let experiments* $\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{VC},\mathcal{A}}$ *and* $\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{VC},\mathcal{A}}$ *for an adversary* $\mathcal{A}$ *be defined as in Figure 2.*

We define the advantage of an adversary $\mathcal{A}$ *against the* existential (resp. strong) unforgeability under chosen-message attacks *(*$\mathsf{EUF\text{-}CMA}$*, resp.* $\mathsf{SUF\text{-}CMA}$*) of* VC *as*

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{VC},\mathcal{A}} := \Pr\left[\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{VC},\mathcal{A}} = 1\right], \quad \textit{resp.} \quad \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{VC},\mathcal{A}} := \Pr\left[\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{VC},\mathcal{A}} = 1\right].$$

3 Confirmation-code Unpredicatability

We next define a simple yet sufficient security notion for verification schemes with confirmation, which we call *confirmation-code unpredictability*. It should be able to thwart faulty implementations up to a certain severeness. To capture this we consider an adversary playing against our notion of unpredictablity, formalized in Figure 3, albeit one may rather think of this adversary as a benign implementer possibly introducing errors in the verification code. In particular, we assume that such errors are introduced while implementing the protocol, before executions take place, such that we model this benign adversary in a non-adaptive way.

Recall that we are interested in using confirmation codes in the context of some overall protocol (e.g., key exchange), whose execution should fail if we have a mismatch in the confirmation codes (on the sender's side running $\mathsf{CreateC}$ and on the receiver's side running VrfyC). Hence, to capture the bad cases, we are interested in the chance of sender and receiver accidentally agreeing on the confirmation codes although there are programming errors in the verification step while the creation step of the sender is correctly implemented.[3] This can happen via two cases:

- Either the benign implementer introduces **programming errors** in the verification step (e.g., skipping instructions as in the goto fail bug [Pou14] or misinterpreting parameters leading to verification not be correctly executed as in the GnuTLS [Tea14] and cURL [GIJ+12] bugs). This is described by having the adversary $\mathcal{A}$ non-adaptively output a vector of modifications $\vec{\delta}^*_c$ to the confirmation code, either of the type "$\triangleright$" to describe that this step of the verification procedure remains intact, or a new confirmation code entry $c \in \mathcal{C}$, e.g., to denote that this verification step has not been executed and the confirmation code is set to some default value c. For non-trivial programming errors at least one entry in $\vec{\delta}^*_c$ needs to be different from $\triangleright$. Since these programming errors should ideally be detectable for any subsequent input, the adversary also chooses the potentially "bad" input *in* for the test run on the sender's side.

- Or, the implementer introduces **input errors** in the verification step for *obj*, like reading only the x-coordinate of an elliptic curve point and defaulting the y-coordinate to 0 (akin to the Bluetooth attack [BN19]). Unlike in the first case, this error may now depend on the actual object *obj* or the other values created by the sender. To ensure that we gave a read error we require that the substituted object obj^* is different from *obj*. This may be even combined with further programming errors, i.e., the adversary may again output $\vec{\delta}^*_c$ as in the first case, only that we allow for non-modifying changes (all entries set to $\triangleright$) this time.

More formally, in the confirmation-code unpredictability ($\mathsf{c\text{-}UP}$) experiment in Figure 3, we define the modified confirmation codes, representing the flawed execution, as $\vec{c}^* \leftarrow \mathsf{Sub}(\vec{c}, \vec{\delta}^*_c)$ for equal-length vectors

[3] A typical scenario we envision is that the verification step is part of a newly-written implementation of, say, a secure connection protocol, which is tested against some other, existing implementation of that protocol. We want that bugs in the new implementation should surface immediately, assuming the other implementation is correct.

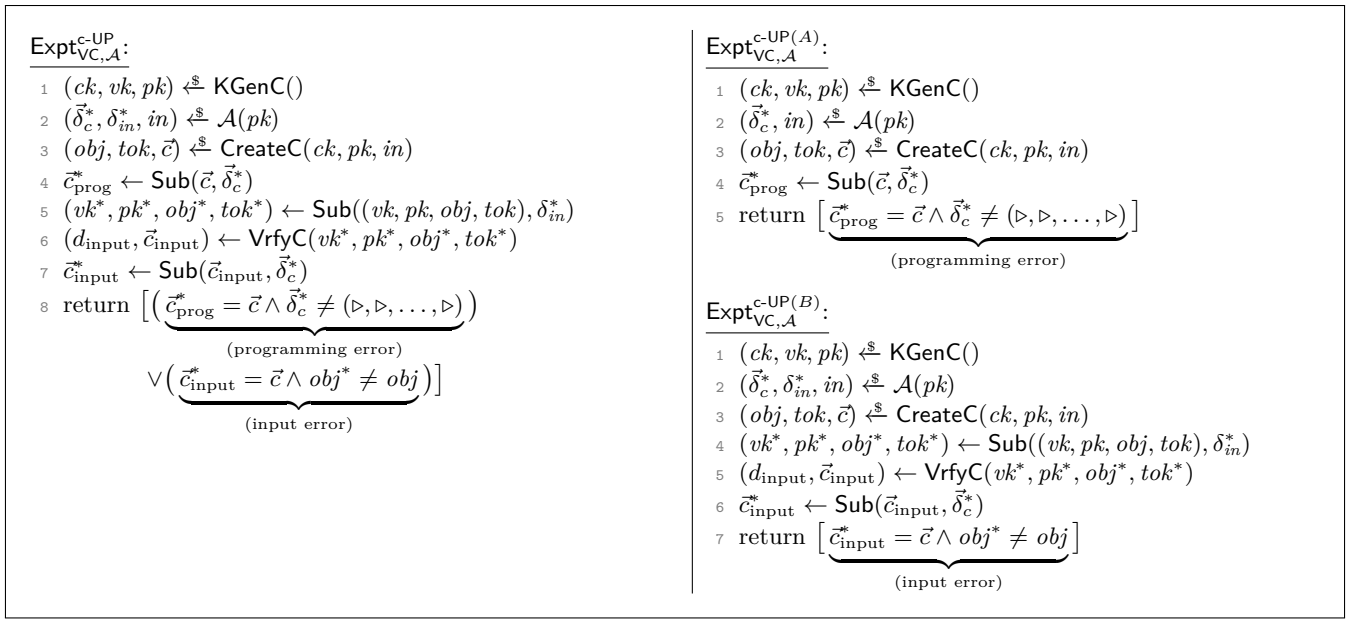

Figure 3: Security experiment for *confirmation-code unpredictability* (c-UP) for verification schemes with confirmation. Combined experiment on left, split-up experiment on right; see text for the definition of Sub. Note that the first case (A) covers errors in the verification program, and the second case (B) covers input errors and, optionally, further programming errors.

$\vec{c}$ and $\vec{\delta}_c$, with algorithm Sub either leaving entries in $\vec{c}$ unchanged and proceeding as expected (for symbol $\triangleright$), or overwriting the value in $\vec{c}[i]$ with the erroneous code $\vec{\delta}_c^*[i]$, possibly $\vec{\delta}_c^*[i] = \perp$:

$$\vec{c}^*[i] = \begin{cases} \vec{c}[i] & \text{if } \vec{\delta}_c^*[i] = \triangleright \\ \vec{\delta}_c^*[i] & \text{else.} \end{cases}$$

As mentioned above, the adversary now succeeds if it correctly predicts the confirmation code $\vec{c}^* = \vec{c}$ of the verification algorithm, in a non-trivial way. Here, non-trivial means that

- either some verification step modified the confirmation code (e.g., due to an error this particular step is skipped, setting the corresponding confirmation code to some default value): $\vec{\delta}_c \neq (\triangleright, \triangleright, \ldots, \triangleright)$,
- or there is a "bad" object obj^* used in verification causing the same confirmation code as on the creator's side which used a different object $obj \neq obj^*$. Here, the code may on top also contain further errors, once more described by a modification of confirmation codes.

It may appear that the second condition (input errors, possibly with program errors) implies the first one. Note, however, that both properties are incomparable, since the first case requires some modification in the verification step for an unaltered object, whereas the second property requires some modification in the object. In proofs, it will be helpful to treat both cases separately, which is why in Figure 3 we display both the overall ("combined") unpredictability experiment $\mathsf{Expt}^{\mathsf{c\text{-}UP}}_{\mathsf{VC},\mathcal{A}}$, as well as separate experiments for case (A) "programming errors", $\mathsf{Expt}^{\mathsf{c\text{-}UP}(A)}_{\mathsf{VC},\mathcal{A}}$, and case (B) "input errors" (with optional programming errors), $\mathsf{Expt}^{\mathsf{c\text{-}UP}(B)}_{\mathsf{VC},\mathcal{A}}$.

Definition 3.1 (Confirmation-code unpredictability of verification schemes with confirmation)**.** *Let* VC *be a verification scheme with confirmation and let experiment* $\mathsf{Expt}^{\mathsf{c\text{-}UP}}_{\mathsf{VC},\mathcal{A}}$ *for an adversary* $\mathcal{A}$ *be defined as*

in Figure 3. We define the advantage of an adversary $\mathcal{A}$ against the confirmation-code unpredictability *(*c-UP*) of* VC *as*

$$\mathsf{Adv}^{\mathsf{c\text{-}UP}}_{\mathsf{VC},\mathcal{A}} := \Pr\left[\mathsf{Expt}^{\mathsf{c\text{-}UP}}_{\mathsf{VC},\mathcal{A}} = 1\right].$$

Generally speaking, confirmation-code unpredictability covers the class of any "hard-coding" in computing confirmation codes, be it due to skipped instructions and hence uninitialized codes returned, or input values wrongly read. We emphasize that protection against arbitrary programming errors is elusive, as this would include the "malicious" error of merely computing the confirmation code (and possibly even a proof of following the verification steps) while ignoring the verification decision. Exploring notions which cover broader classes of implementation errors is an interesting direction for future research.

To illustrate the definition let us argue that simply putting *obj* into $\vec{c} = (obj)$ to thwart predictability attacks in general does not work. Take a signature scheme as an example, where $in = obj = m$ is the message to be protected via the signature *tok*. Note that this message is chosen by the adversary. In the first step, our successful attacker $\mathcal{A}$ against unpredictability chooses an arbitrary message $in = m$ and also sets $\vec{\delta}^*_c = (m)$. In the next step it chooses a different message $m^* \neq m$ and sets $obj^* \leftarrow m^*$. Then, running VrfyC on obj^* yields the confirmation code $\vec{c}_{\text{input}} = (m^*)$, but the $\vec{\delta}^*_c$ value overwrites the only confirmation code again with m in $\vec{c}^*_{\text{input}}$. Hence, the adversary has successfully created a different object obj^* for which VrfyC creates, due to a non-adaptive programming error, the same confirmation code as CreateC, letting $\mathcal{A}$ win the unpredictability game.

As another example, let us consider using the signature or MAC tag of a signature resp. MAC scheme as the confirmation code, $\vec{c} = (tok)$. This generically achieves unpredictability, assuming the scheme is (existentially) unforgeable: if the adversary can predict the token for a message (be it the creator's code $\vec{c}$ overwritten by $\vec{\delta}^*_c$ or the verifier's code $\vec{c}_{\text{input}}$), then this prediction together with the corresponding message is indeed a valid forgery. Using the *tok* as confirmation code however is bad solution in practice: a verification implementation would naturally copy the confirmation code, counteracting the idea that the confirmation code should capture the verification steps taken.[4] We hence do not consider this approach further, but instead next turn to better, practical constructions of confirmation codes based on actual intermediate computation values that cannot simply be copied.

4 Practical Verification with Unpredictable Confirmation Codes

Recall the idea that, ideally, confirmation codes should be computable on the verifier's side as a by-product of the verification steps. In this section we give three examples where one can indeed achieve this, one each for asymmetric, symmetric, and public verification schemes. The asymmetric example is the RSA-PSS signature scheme, for the symmetric example we consider the HMAC message authentication algorithm, and in the public verification setting we discuss curve-point validation in elliptic curves, both that the point is on the curve and also that it belongs to the right subgroup.

4.1 RSA-PSS Signatures

As an example of an asymmetric verification scheme, we consider the RSA-PSS signature scheme, proposed by Bellare and Rogaway [BR96], standardized by the IETF in PKCS #1 v2.1 [JK03] and NIST [oST23b], and mandated in major protocols like TLS 1.3 [Res18]. Figure 4 shows the RSA-PSS signature scheme,

[4] One might try to rule out such bad solutions in the unpredictability notion, e.g., by allowing a flawed VrfyC implementation to copy input values (like *tok*). However, similar trivial codes would exist under such definition, like splitting the signature in halves, or flipping a bit in it. This issue appears to be inherent in the general way a cryptographic notion like c-UP treats algorithms, namely, as atomic operations. Only if one talks about specific schemes and their individual steps, one can determine more fine-grained confirmation codes.

RSA-PSS.Sign(sk, m), RSA-PSS^{+c}.CreateC(sk, pk, m):

1 $r \xleftarrow{\$} \{0,1\}^n$
2 $s \leftarrow H(\langle\text{padding}\rangle \| m \| r)$
3 $mask \leftarrow G(s)$
4 $t \| u \leftarrow mask \oplus (r \| 0^l)$
5 $\sigma \leftarrow (0 \| s \| t \| u)^d \bmod N$
6 return σ $\boxed{(m, \sigma, (s))}$

RSA-PSS.Vrfy(pk, m, σ), RSA-PSS^{+c}.VrfyC(vk, pk, m, σ):

7 $b \| s \| t \| u \leftarrow \sigma^e \bmod N$
8 $mask \leftarrow G(s)$
9 $r \leftarrow t \oplus mask$
10 $s' \leftarrow H(\langle\text{padding}\rangle \| m \| r)$
11 If $s' = s$ and $mask[n : \ldots] = u$ and $b = 0$
12 then return 1 $\boxed{(1, (s'))}$
13 else return 0 $\boxed{(0, \perp)}$

Figure 4: The RSA-PSS signature scheme using functions $G\colon \{0,1\}^\ell \to \{0,1\}^{n+l}$, $H\colon \{0,1\}^* \to \{0,1\}^\ell$, and RSA-PSS^{+c} augmenting it with confirmation codes. Output for RSA-PSS in gray boxes, output for RSA-PSS^{+c} in framed boxes.

both the classical version (mostly following RFC 3447 [JK03]), as well as the confirmation-augmented version. For the latter, we leverage that in RSA-PSS, the verifier essentially recomputes the computational steps performed upon signing, re-deriving in particular the signer's randomness. This is allows us to use a single confirmation code, in which the re-derived randomness r and signed message m are bundled together in the hash value $H(\langle\text{padding}\rangle \| m \| r)$.

Confirmation code rationale. When deploying verification schemes with confirmation, the main property we are interested in achieving is confirmation-code unpredictability (c-UP). For RSA-PSS, the confirmation code leverages the randomness r drawn when signing (which is re-derived when verifying) as an a-priori unpredictable value. One might wonder whether using r alone as confirmation code, $\vec{c} = (r)$, already yields unpredictability. This would indeed be sufficient to satisfy the "programming errors" condition of c-UP (cf. Figure 3).

The "input errors" condition of c-UP however demands that changes to the object *obj*, i.e., the signed message, lead to unpredictable changes in the confirmation code, too; this is not the case when $\vec{c} = (r)$. Instead, we employ the hash value $s = H(\langle\text{padding}\rangle \| m \| r)$ computed upon signing. As we will show next, the r value entering the hash ensures unpredictability while the inclusion of m ensures that modifications of the message lead to confirmation code changes. With the confirmation code involving both r and m, we can indeed fully establish confirmation code unpredictability.[5]

It remains to show that the chosen confirmation code does not negatively affect unforgeability of the augmented scheme RSA-PSS^{+c}. Concretely, even when outputting confirmation codes along with the signatures in response to signing oracle queries in the unforgeability games (cf. Definition 2.4), RSA-PSS^{+c} should remain unforgeable. This is indeed easy to see: Since by correctness VrfyC computes the same confirmation code as CreateC given only public information ($vk = pk$), the message m and the signing oracle's token output *tok*, an unforgeability adversary can simply compute the confirmation codes corresponding to the signatures generated by the signing oracle itself.

Theorem 4.1. *The* RSA-PSS^{+c} *verification scheme with confirmation augmenting the* RSA-PSS *signature scheme, given in Figure 4, is confirmation-value unpredictable when modeling H as random oracle. Concretely, for any (even possibly unbounded) adversary $\mathcal{A}$ we have*

$$\mathsf{Adv}^{\text{c-UP}}_{\text{RSA-PSS}^{+c},\mathcal{A}} \leq 2 \cdot 2^{-\min(n,\ell)}.$$

[5] A possibly more robust approach could be to leverage that H is an extendable output function and generate further output bits ("beyond" s) to be used as confirmation code. This may avoid implemenation errors copying s for s' in verification. We leave studying unpredictability notions that differentiate such errors for future work.

Proof. We separately consider the two parts (A) "programming errors" and (B) "input errors" of c-UP, so that we have

$$\mathsf{Adv}^{\text{c-UP}}_{\text{RSA-PSS}^{+c},\mathcal{A}} \leq \mathsf{Adv}^{\text{c-UP}(A)}_{\text{RSA-PSS}^{+c},\mathcal{A}} + \mathsf{Adv}^{\text{c-UP}(B)}_{\text{RSA-PSS}^{+c},\mathcal{A}}.$$

To win in part (A) "programming errors" of c-UP, adversary $\mathcal{A}$ must correctly predict, overwriting via $\vec{\delta}^*_c$, the confirmation code $H(\langle\text{padding}\rangle\|m\|r)$. We leverage that the randomness r sampled within RSA-PSS^{+c}.CreateC generating the confirmation code $\vec{c}$ to be predicted, is a uniformly random string of length n bits. Modeling H as random oracle, $\mathcal{A}$ must hence either predict $r \in \{0,1\}^n$ (in a query to H), or otherwise the hash value itself (which in that case is a random ℓ-bit output of H). Hence, we can bound $\mathcal{A}$'s advantage in this case as $\mathsf{Adv}^{\text{c-UP}(A)}_{\text{RSA-PSS}^{+c},\mathcal{A}} \leq 2^{-\min(n,\ell)}$.

In part (B) "input errors" of c-UP, $\mathcal{A}$ may arbitrarily overwrite the inputs to VrfyC computing $\vec{c}_{\text{input}}$ in the c-UP experiment, as long as this overwriting modifies the message, i.e., $obj^* \not\equiv obj$. Further, $\mathcal{A}$ can but need not overwrite the confirmation code $\vec{c}_{\text{input}}$. We analyze $\mathcal{A}$'s advantage based on whether it overwrites the confirmation code (i.e., whether $\vec{\delta}^*_c = (\triangleright)$ or not).

1. Overwriting ($\vec{\delta}^*_c \neq (\triangleright)$): Since there is only a single entry in $\vec{c}$, if $\mathcal{A}$ overwrites $\vec{c}_{\text{input}}$, it actually needs to predict $\vec{c}$ as for part (A), so this case is already covered above.

2. No overwriting ($\vec{\delta}^*_c = (\triangleright)$): Without overwriting, the confirmation codes $\vec{c}$ and $\vec{c}_{\text{input}}$ output by CreateC resp. VrfyC must collide for $\mathcal{A}$ to win. Note that the two algorithms use different message inputs m and m^*, which get output as objects $obj^* \not\equiv obj$, where $obj = m = in$ and obj^* is the result of applying δ^*_{in} to obj. Hence the $H(\langle\text{padding}\rangle\|m\|r)$ values in $\vec{c}$ and $\vec{c}_{\text{input}}$ must collide for distinct inputs for m. Observe that although $r \in \{0,1\}^n$ is drawn at random in CreateC, $\mathcal{A}$ may opt to not overwrite the signature via δ^*_{in}, in which case VrfyC will use the same value r when computing $\vec{c}_{\text{input}}$. Nevertheless, r serves as a kind of "salt" in the computation: $\mathcal{A}$ either needs to predict the value of $r \in \{0,1\}^n$ used in CreateC (from which it may pre-compute a collision under H for distinct messages), or guess the ℓ-bit output of H directly (for pre-computation). This leaves $\mathcal{A}$ with chance of $\mathsf{Adv}^{\text{c-UP}(B)}_{\text{RSA-PSS}^{+c},\mathcal{A}} \leq 2^{-\min(n,\ell)}$ for part (B). □

4.2 HMAC

Turning towards symmetric verification scheme, we next show how the message authentication code algorithm HMAC [BCK96a], standardized by IETF [KBC97] and NIST [oST08], and universally used as a MAC, PRF, and for key derivation, can be augmented with a confirmation code. Recall that HMAC, in its most simple form for some key K (of full block length of the underlying compression function) and message m, is defined as

$$\mathsf{HMAC}(K,m) := \mathsf{H}(K \oplus \texttt{opad}\|\mathsf{H}(K \oplus \texttt{ipad}\|m)),$$

where H is a Merkle–Damgård hash function [Mer90, Dam90] (e.g., SHA-256) and `opad`, `ipad` are two distinct (full-block) constants.

If one wanted to opt for a minimalist approach while sending HMAC values, one could use the inner hash value $C = \mathsf{H}(K \oplus \texttt{ipad}\|m)$ as confirmation code. We show next that this confirmation-augmented version HMAC^{+c} of HMAC, given in Figure 5, achieves confirmation-code unpredictability based on assumptions used already in establishing PRF security of the regular HMAC function [Bel06, Bel15]: the dual-PRF security of the underlying compression function h, asking that both h and $\overline{h}(x,y) := \mathsf{h}(y,x)$, i.e., when keyed via the second input, are secure PRFs (see Appendix A.2 for the definition of PRF security, PRF-sec).

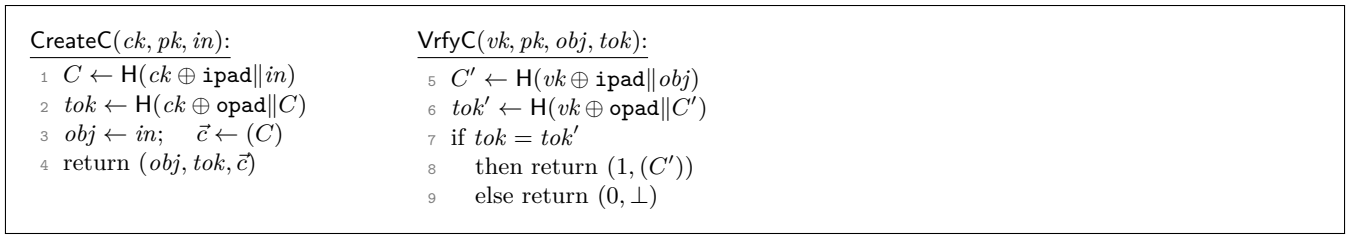

Figure 5: The tag creation and verification algorithms of HMAC^{+c}, the confirmation-augmented version of HMAC. Key generation as for HMAC samples a b-bit key $ck \xleftarrow{\$} \{0,1\}^b$.

For notation, recall that a Merkle–Damgård hash function involves iterating a compression function $\mathsf{h}\colon \{0,1\}^c \times \{0,1\}^b \to \{0,1\}^c$, mapping a c-bit chaining value and b-bit input block to a c-bit output. The iteration, or cascade, $\mathsf{h}^*\colon \{0,1\}^c \times \{0,1\}^{b*} \to \{0,1\}^c$ then processes a multi-block input as $\mathsf{h}^*(C_0, m_1 \| \ldots \| m_\ell) := C_\ell$ where $C_i := \mathsf{h}(C_{i-1}, m_i)$ and $m_i \in \{0,1\}^b$ for $1 \le i \le \ell$. We can hence write a Merkle–Damgård hash function as $\mathsf{H}(M) := \mathsf{h}^*(IV, \overline{M})$, where $\overline{M}$ denotes padding M with its length and to a multiple of b bits. As a useful intermediate security notion, the advantage of an almost-universal (AU) adversary against h^* is defined as

$$\mathsf{Adv}^{\mathsf{AU}}_{\mathsf{h}^*,\mathcal{A}} := \Pr\left[\mathsf{h}^*(X, M_1) = \mathsf{h}^*(X, M_2) \wedge M_1 \neq M_2 \;\middle|\; (M_1, M_2) \xleftarrow{\$} \mathcal{A}; X \xleftarrow{\$} \{0,1\}^c\right].$$

We are now ready to state the confirmation-code unpredictability of HMAC^{+c}.

Theorem 4.2. *The verification scheme with confirmation* HMAC^{+c} *augmenting the* HMAC *MAC scheme, given in Figure 5, is confirmation-value unpredictable.*

Concretely, we construct reductions $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$, $\mathcal{B}_4$ *such that*

$$\mathsf{Adv}^{\mathsf{c\text{-}UP}}_{\mathsf{HMAC}^{+c},\mathcal{A}} \le \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\overline{h},\mathcal{B}_1} + \ell_1 \cdot \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{h},\mathcal{B}_1} + \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\overline{h},\mathcal{B}_3} + (\ell_1 + \ell_2) \cdot \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{h},\mathcal{B}_4} + 2 \cdot 2^{-c},$$

where ℓ_1, ℓ_2 *denotes the block-length of the padded message in output resp. message object* obj^* *modified by* $\mathcal{A}$.

Proof. We separately consider the two parts (A) "programming errors" and (B) "input errors" of c-UP, so that we have

$$\mathsf{Adv}^{\mathsf{c\text{-}UP}}_{\mathsf{HMAC}^{+c},\mathcal{A}} \le \mathsf{Adv}^{\mathsf{c\text{-}UP}(A)}_{\mathsf{HMAC}^{+c},\mathcal{A}} + \mathsf{Adv}^{\mathsf{c\text{-}UP}(B)}_{\mathsf{HMAC}^{+c},\mathcal{A}}.$$

For part (A) "programming errors", we use that h^* is a PRF for prefix-free queries [BCK96b, Theorem 3.1], which allows us to replace its output C making up the confirmation code output by CreateC in the c-UP game with a random c-bit value. For the case of HMAC, we obtain the prefix-free PRF result for h^* via PRF security of $\overline{h}$ (processing the key block as $\mathsf{h}(IV, ck \oplus \texttt{ipad})$) and then PRF security of the ℓ_1-block sequence of h processing the input message.

After this step, the adversary $\mathcal{A}$ has a remaining chance of 2^{-c} in guessing the now-random confirmation value. Thus, the bound for part (A) is

$$\mathsf{Adv}^{\mathsf{c\text{-}UP}(A)}_{\mathsf{HMAC}^{+c},\mathcal{A}} \le \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\overline{h},\mathcal{B}_1} + \ell_1 \cdot \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{h},\mathcal{B}_1} + 2^{-c}.$$

For part (B) "input errors", we leverage the AU security of h^* and the PRF security of $\overline{h}$ (the latter again for processing the key block). Recall that in part (B), $\mathcal{A}$ must modify the message object *obj* to be verified and may optionally overwrite parts of the confirmation code resulting from verifying the modified object obj^*. Similar to the proof for RSA-PSS (cf. Theorem 4.1), since there is only one component in the

confirmation code, overwriting $\vec{c}_{\text{input}}$ puts $\mathcal{A}$ in the same setting as for part (A), so we can focus on $\mathcal{A}$ not overwriting it (i.e., $\vec{\delta}_c^* = (\triangleright)$).

The confirmation code is computed as $C = \mathsf{H}(ck \oplus \texttt{ipad} \| in) = \mathsf{h}^*(\mathsf{h}(IV, ck \oplus \texttt{ipad}), \overline{in})$. We first replace the value $\mathsf{h}(IV, ck \oplus \texttt{ipad})$ within the $\mathsf{CreateC}$ and VrfyC calls by a uniformly random string $X \xleftarrow{\$} \{0,1\}^c$, applying PRF security of $\overline{h}$. Observe that we are now precisely in the AU setting for h^*: $\mathcal{A}$ is tasked to produce two different message objects $in = obj$ and obj^* such that $\vec{c} = (\mathsf{h}^*(X, M_1)) = (\mathsf{h}^*(X, M_2)) = \vec{c}^*_{\text{input}} = \vec{c}_{\text{input}}$, where $M_1 = \overline{in}$ and $M_2 = \overline{obj^*}$ are the padded message objects. Applying the AU bound for h^* by Bellare [Bel15, Lemma 3.1], together with the dual-PRF step above we can hence bound part (B) as

$$\mathsf{Adv}^{\mathsf{c\text{-}UP}(B)}_{\mathsf{HMAC}^{+c},\mathcal{A}} \leq \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\overline{h},\mathcal{B}_3} + (\ell_1 + \ell_2) \cdot \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{h},\mathcal{B}_4} + 2^{-c}.$$

Summing up the terms for both parts yields the claim. □

It remains to argue that HMAC^{+c} is still a secure MAC, although the unforgeability adversary learns the confirmation code, i.e., the inner hash value $\mathsf{H}(K \oplus \texttt{ipad} \| m)$. To show this we consult the tight security proof of Gazi et al. [GPR14] for HMAC, showing that the augmented version is also tightly secure. They first consider the independent-key variant NMAC of HMAC:

$$\mathsf{NMAC}((K_{\text{in}}, K_{\text{out}}), m) = \mathsf{H}(K_{\text{out}}, \mathsf{H}(K_{\text{in}}, m)).$$

Then they argue that one can replace the outer hash function by a truly random function r. This is admissible if one assumes that H is a pseudorandom function. The reduction actually computes the inner hash values, such that knowing our confirmation values does not lend additional power to the adversary.

In the next step of their proof, Gazi et al. argue that one cannot distinguish $r(\mathsf{H}(K_{\text{in}}, \cdot))$ from a random function R, unless the adversary finds a weak collision in the inner hash function. The reduction proceeds in multiple steps, first turning the adaptive adversary into a non-adaptive one, then into a prefix-free one, and finally arguing that this prefix-free adversary cannot distinguish the inner hash function from a random one. In all intermediate steps the adversaries may learn the results of the inner hash evaluations, showing that the argument is independent of the question if the adversary against NMAC learns the intermediate values or not.

The final step is to lift the security proof from NMAC to HMAC, which can be found for example in [Bel06]. This follows from the assumption that the compression function h underlying H is a dual-PRF and also holds if an adversary learns the inner hash values. Overall it follows that the output of HMAC still looks random, even if the adversary gets to learn the inner hash value. This also implies that the augmented HMAC version is still a secure MAC.

Implicit verification of MACs. As an interesting side note, we remark that for a PRF-based MAC scheme like HMAC, the MAC value itself can act as a good confirmation code as long as the MAC is *actually not sent* in the higher-level protocol, but only implicitly checked, for example by letting it enter a key derivation step in a key exchange protocol. While some key exchange protocols do not sent MAC values explicitly (e.g., the SIGMA [Kra03] protocol variant with MACs under the signature), we are not aware of one enforcing computation of the implicit MAC by binding it to key derivation. We speculate that this is mainly since MACs are usually explicitly added for key confirmation, and an implicit re-computation lacks the explicit verification decision (bit). We leave exploring implicit verification of MACs and the definition of confirmation codes in this context as an interesting avenue for future work.

4.3 Validity Checks for Elliptic Curve Points

For the public verification setting, let us consider a scheme for checking if a pair (x, y) is a point on the elliptic curve $x^3 + ax + b = y^2$ over a field $\mathbb{F}$. As a motivational example consider the fixed-coordinate

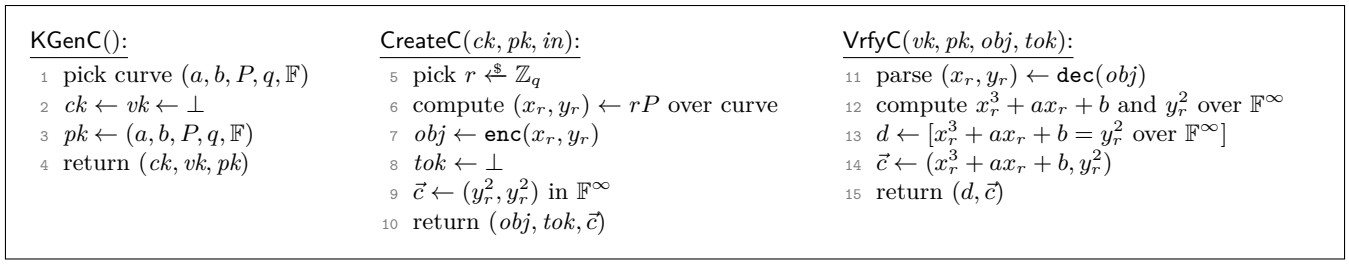

Figure 6: Construction of verification scheme with confirmation $\mathsf{VC} = (\mathsf{KGenC}, \mathsf{CreateC}, \mathsf{VrfyC})$ for checking validity of a random point on the elliptic curve.

invalid-curve attack of Biham and Neumann against the Bluetooth protocol [BN19]. The attack works as follows: In the attacked version of the Bluetooth protocol both parties, Alice and Bob, exchange the x- and y-coordinates of two public Diffie–Hellman shares, (x_A, y_A) and (x_B, y_B). The attack sets both y-coordinates to 0 for the Diffie–Hellman values in transition, such that the parties use the point $(x_A, 0)$ resp. $(x_B, 0)$ to complete the Diffie–Hellman computation. Although not relevant for us here, let us remark that Biham and Neumann [BN19] show that these modified points have low order on another curve, such that, with probability 1/4, the parties end up with a trivial Diffie–Hellman key consisting of the neutral element.

Note that the fixed coordinate invalid curve attack above is an active network attack (called "semi-passive" in [BN19]). But we can envision it to be a programming mistake which erroneously reads the input y-coordinate as 0 or some other value, and where no validity check is carried out. The straightforward way to mitigate the attack is to check that the received point indeed lies on the curve. We can model such a verification check in our notion for confirmation codes as shown in Figure 6. The idea is, given an incoming pair (x_r, y_r) on the verifier's side, to compute the value $x_r^3 + ax_r + b$ and check that it matches the value y_r^2; alas, a false implementation may skip this check. Hence, we let the confirmation code computed by the verifier include the value $x_r^3 + ax_r + b$ as well as y_r^2.

The creator also computes the confirmation code vector, simply placing the square of the y-coordinate in both components, $\vec{c} \leftarrow (y_r^2, y_r^2)$ over $\mathbb{F}$. Note also that we are interested in checking *random* curve points, as will be used in genuine test runs in, say, an implementation of the Bluetooth key exchange protocol. Formally, we thus set the input space $\mathcal{I} = \{\perp\}$ to be empty, and since the keys are all public, tokens are irrelevant and we also set $\mathcal{T} = \{\perp\}$.

A caveat lies in the actual encoding of elliptic curve points. We assume that the object of the curve point (x_r, y_r) uses an encoding, $\texttt{enc}(x_r, y_r)$, e.g., with encoding `0x00` for the point at infinity, prefix `0x02` or `0x03` for compressed encoding, and `0x04` for uncompressed encoding. We assume that the receiver can decode such values correctly, i.e., $\texttt{dec}(\texttt{enc}(x_r, y_r)) = (x_r, y_r)$ returns the (uncompressed) x-coordinate and y-coordinate of the point. For simplicity of representation we assume here that decoding the point at infinity yields the coordinates $\infty \notin \mathbb{F}$ for both entries, that performing any field operation with this dedicated value yields ∞ again, and that $\infty = \infty$ by definition. We simply say that the operations are over $\mathbb{F}^\infty$ to include this case.

We note that correctness of the validity verification scheme holds straightforwardly, since the creator and the verification compute the same value by the curve equation, independently of the input *in*. In the proof below we assume that the curve parameters are trustworthy, e.g., if parties used fixed-curve parameters like NIST's `P-256` or Bernstein's `Curve25519` [Ber06]. In particular, we assume that the curve is given by the values a, b, the field $\mathbb{F}$, a generator P and the order q of the group generated by P. Note that we deal with subgroup membership tests in the next section and only discuss checking validity of curve points here.

Theorem 4.3. *For any elliptic curve* $(a, b, P, q, \mathbb{F})$ *the construction of the verification scheme* $\mathsf{VC} = (\mathsf{KGenC}, \mathsf{CreateC}, \mathsf{VrfyC})$ *in Figure 6 is confirmation-code unpredictable. Concretely, for any (even possibly unbounded) adversary* $\mathcal{A}$ *against unpredictability we have*

$$\mathsf{Adv}^{\mathsf{c\text{-}UP}}_{\mathsf{VC},\mathcal{A}} \leq \frac{12}{q},$$

where q *is the order of the elliptic curve.*

Proof. Consider an arbitrary adversary $\mathcal{A}$. Note that the creator key, the verification key, and the public key only contain the description of an elliptic curve of prime order q. In order for the adversary to win, it would need to have to either output a non-trivial $\vec{\delta}^*_c$ such that value $\vec{c}^*_{\text{prog}} \leftarrow \mathsf{Sub}(\vec{c}, \vec{\delta}^*_c)$ coincides with $\vec{c}$, or to output $\vec{\delta}^*_c, \delta^*_{in}$ such that $\vec{c}^*_{\text{input}} \leftarrow \mathsf{Sub}(\vec{c}_{\text{input}}, \vec{\delta}^*_c)$ equals $\vec{c}$ but for a different object $obj^* \neq obj$.

For the first case, a non-trivial $\vec{\delta}^*_c$ making $\vec{c}^*_{\text{prog}}$ match $\vec{c}$, note that the adversary chooses the modification $\vec{\delta}^*_c$ before the random curve point rP is selected. Let us first consider the case that $\mathcal{A}$ chooses $\vec{\delta}^*_c$ such that it sets the first entry X^* in the confirmation code to some predefined value; the value Y^* in the second component may also be set, or may be left equal to y_r^2, but this decision is also made before the curve point is chosen. If $X^* = \infty$ describes the point at infinity, then the probability that our random curve point ends up there, i.e., $r = 0$, is at most $1/q$. If $X^* \in \mathbb{F}$ does not match a valid curve point, then the x-coordinate of our random point cannot match X^*, such that the probability of an adversarial success is 0 in this case.

So we may next assume that $X^* \in \mathbb{F}$ belongs to a valid curve point. Recall that we must have $Y^* = y_r^2$ for a match on the confirmation codes. Hence, there are at most 6 valid curve points with the same value $Y^* = y_r^2$ in total: The equation $x^3 + ax + b = y_r^2$ has at most three solutions for x^* over the field $\mathbb{F}$, and each solution x^* can only be combined with either sign $\pm$ for the y-value to yield the same square. Since we pick a curve point rP uniformly at random, the probability of landing on one of these at most 6 points is bounded by $6/q$.

Next assume that the adversary leaves the first entry unchanged but sets the y_r^2 component to a preselected value Y^*. If Y^* is not a valid square of a curve point then the y-coordinate our value rP cannot equal Y^* after squaring. If $Y^* = \infty$ describes the point at infinity then the probability that $r = 0$ is at most $1/q$. It remains to consider Y^* which equals the square of a y-coordinate of a valid curve point. Analogously to the previous argument, the Y^*-value on the curve can account for at most 6 elliptic curve points. Hence, the probability that our random curve points induces the same value Y^* is at most $6/q$.

The other case is that the adversary tries to create a different object $obj^* \neq obj$, but where the confirmation code $\vec{c}$ created with the object obj by $\mathsf{CreateC}$ matches $\vec{c}^*_{\text{input}}$. This value, in turn, is the non-adaptively determined substitution of the confirmation code $\vec{c}_{\text{input}}$ computed by VrfyC for obj^*. With the same argument as in the first case we can bound the probability that any predetermined changes to $\vec{c}_{\text{input}}$ yield the same confirmation code $\vec{c}$ is at most $6/q$. Note that this is independent of the fact that $\vec{c}^*_{\text{input}}$ is a modification of $\vec{c}_{\text{input}}$ computed by VrfyC for obj^* instead. Hence, we can for now assume that $\vec{\delta}^*_c = (\triangleright)^*$ and $\vec{c}^*_{\text{input}} = \vec{c}_{\text{input}}$, and therefore $\vec{c}^*_{\text{input}} = \vec{c}_{\text{input}} = \vec{c}$ for a successful attack.

Next observe that $\vec{c}_{\text{input}}$ is fully determined by the input object obj^* given to VrfyC. This object, however, is determined by the original object obj with non-adaptively chosen modifications δ^*_{in}. If the modification δ^*_{in} leaves the object unchanged then this cannot constitute a valid attack. But for any non-trivial modification we can conclude once more that a match for the confirmation code of the randomly chosen curve point obj can only occur with probability at most $6/q$. Only this time we have to add the two probabilities. □

KGenC():
1 pick curve $(a, b, P, q, n, h, \mathbb{F})$
2 $ck \leftarrow vk \leftarrow \perp$
3 $pk \leftarrow (a, b, P, q, n, h, \mathbb{F})$
4 return (ck, vk, pk)

CreateC(ck, pk, in):
5 pick $r \xleftarrow{\$} \mathbb{Z}_n$
6 compute $(x_r, y_r) \leftarrow rP$ over curve
7 $obj \leftarrow \mathtt{enc}(x_r, y_r)$
8 $tok \leftarrow \perp$
9 $\vec{c} \leftarrow (x_r, -y_r)$ in $\mathbb{F}^\infty$
10 return $(obj, tok, \vec{c})$

VrfyC(vk, pk, obj, tok):
11 parse $R = (x_r, y_r) \leftarrow \mathtt{dec}(obj)$
12 $Q \leftarrow n_\ell R$
13 for $i = \ell - 1$ downto 0 do
14 $\quad Q \leftarrow 2Q$
15 $\quad$ if $i = 0$ then $\vec{c} \leftarrow (Q_x, Q_y)$
16 $\quad Q \leftarrow Q + n_i R$
17 if $Q = \mathcal{O}$ then $d \leftarrow 1$ else $d \leftarrow 0$
18 return $(d, \vec{c})$

Figure 7: Construction of verification scheme with confirmation $\mathsf{VC} = (\mathsf{KGenC}, \mathsf{CreateC}, \mathsf{VrfyC})$ for checking that a random point on the elliptic curve belongs to the prime order subgroup.

4.4 Subgroup Membership Tests for Elliptic Curves

The verification in Section 4.3 only checks that the received value is indeed a curve point. For some applications this is already sufficient, since the parties use the so-called (low-order) clearing, also called clamping for Bernstein's curve `Curve25519` [Ber06]. This clearing means to multiply the received random point R with the cofactor h of the elliptic curve, where h is such that $q = nh$ and h is co-prime to the prime order n of the subgroup. If R is a valid curve point then $Q = hR$ is definitely in the subgroup of order n. This approach is for example recommended by the German Federal Agency of Information Security [fIS18], where the check for Q to lie on the curve is nonetheless mandatory.

For other applications, checking that the point is indeed in the subgroup is recommended. For instance, NIST [BCR+18] recommends for "ECC Full Public-Key Validation" that one checks that the received value R is a curve point and that $nR = \mathcal{O}$ is the point at infinity. The two properties together ensure that R is in the right subgroup, since h is co-prime to n and nR thus cannot have a smaller order.

We only describe here the subgroup membership test, it can be easily combined with the curve-point test of the previous section. We stress once more that this subgroup membership test requires that the input value is indeed a curve point. We assume that the test $nR = \mathcal{O}$ is done by computing nR via square-and-multiply, with $n = \sum_{i=0}^{\ell} n_i 2^i$ being the binary representation of the (odd) prime n describing the subgroup's order. Note that $n_0 = 1$ such that the verifier computes $2(\frac{n-1}{2}R) + R = (n-1)R + R$ in the final iteration. Now $(n-1)R + R$ can only equal $\mathcal{O}$ if and only if $(n-1)R = -R$ for valid curve points. Hence, we let the confirmation code equal this pair of field values in the last iteration. The creator of the random curve point can easily compute the matching confirmation code by putting $(x_r, -y_r)$ for the created point $R = (x_r, y_r)$.

Theorem 4.4. *For any elliptic curve $(a, b, P, q, n, h, \mathbb{F})$ the construction of the verification scheme $\mathsf{VC} = (\mathsf{KGenC}, \mathsf{CreateC}, \mathsf{VrfyC})$ in Figure 7 is confirmation-code unpredictable. Concretely, for any (even possibly unbounded) adversary $\mathcal{A}$ against unpredictability we have*

$$\mathsf{Adv}^{\mathsf{c\text{-}UP}}_{\mathsf{VC},\mathcal{A}} \leq \frac{1}{n},$$

where n is the order of the subgroup of the elliptic curve.

Proof. The proof is similar to the one for testing for curve points in the previous section. First note that, if the adversary non-adaptively overwrites one of the two elements for $\mathbb{F}^\infty$ via $\vec{\delta}^*_c$, then the probability of hitting this element with the random point is at most $1/n$. Similarly, if the adversary overwrites the object obj adaptively via some other (valid) curve point obj^*, then we distinguish two cases: If obj^* is also a member in the subgroup then the verification will output $(n-1)obj^* = -obj^*$ as the confirmation

code, which will not match $-obj$ output by the creator. If obj^* is a curve point but does not belong to the subgroup, then $(n-1)obj^*$ cannot equal $-obj$, or else the point obj^* would indeed belong to the subgroup. □

5 Key Exchange with Verifiable Verification

We now show how confirmation codes can be checked in practice with very little overhead, using key exchange protocols as example. Key exchange not only is one of the most widely deployed types of cryptographic protocols, underlying secure communiation protocols like TLS, SSH, QUIC, Signal, and others, it has in the past also been at the heart of severe verification bugs in the past, including Apple's goto fail bug [Pou14] and the erroneous certificate validation in GnuTLS [Tea14] or cURL [GIJ+12].

Here we show how confirmation codes can be easily integrated into the *key derivation* step of key exchange protocols. This ensures that programming mistakes that lead to verification steps to be skipped (and hence confirmation codes not properly be computed) immediately lead to *functional incorrectness* via diverging keys being derived. Such diverging keys will in turn be immediately noticed upon interoperability testing: the erroneous protocol implementation will simply be unable to establish the correct session key, making the programmer aware of the verification bug before deployment.

5.1 Basic Key Exchange Syntax

For our setting, we are most concerned with *correctness* properties of key exchange protocols. It hence suffices to consider the following, simple syntax for key exchange protocols for the most part, and defer more subtle aspects related to security properties to later.

Definition 5.1 (Key exchange protocol)**.** *A* key exchange protocol $\mathsf{KE} = (\mathsf{KGen}, \mathsf{Execute})$ *consists of two efficient algorithms defined as follows.*

$\mathsf{KGen}() \xrightarrow{\$} (sk, pk)$. *This probabilistic algorithm outputs a secret key sk and a public key vk.*

$\mathsf{Execute}(sk_A, pk_A, sk_B, pk_B) \xrightarrow{\$} (trans, K_A, K_B)$ *On input the secret/public keys of two parties A and B, this probabilistic algorithm honestly executes a run of the key exchange protocol between the two parties and outputs the resulting communication transcript trans as well as the session keys established by each party, K_A resp. K_B.*

For technical reasons, we restrict our focus to what we call *canonical* key exchange protocols where (1) the protocol aborts whenever a verification step fails during the protocol execution, and (2) the users' public keys of the key exchange protocol, generated by KGen, only contain public keys of the verification sub-procedures. We consider both to be reasonable restrictions, adhered by most practical protocols.

5.2 Correctness and Faulty Verification

We next consider correctness for key exchange protocols. Classically, correctness is defined as an undisturbed run of a key exchange protocol between two parties leads to the same session key at each party with probability 1. We call this an *always correct* key exchange protocol.

For detecting accidental mistakes in implementations, we are further interested in what we call *noticeable correctness*: whether an undisturbed key exchange run leads to matching session keys with noticeable probability. We will see that, for a faulty implementation, verifiable verification should *preclude* noticeable correctness, to make sure implementation mistakes by one side of the communication lead to correctness failures (i.e., a non-functional protocol run) with high probability in an honest execution of a key exchange.

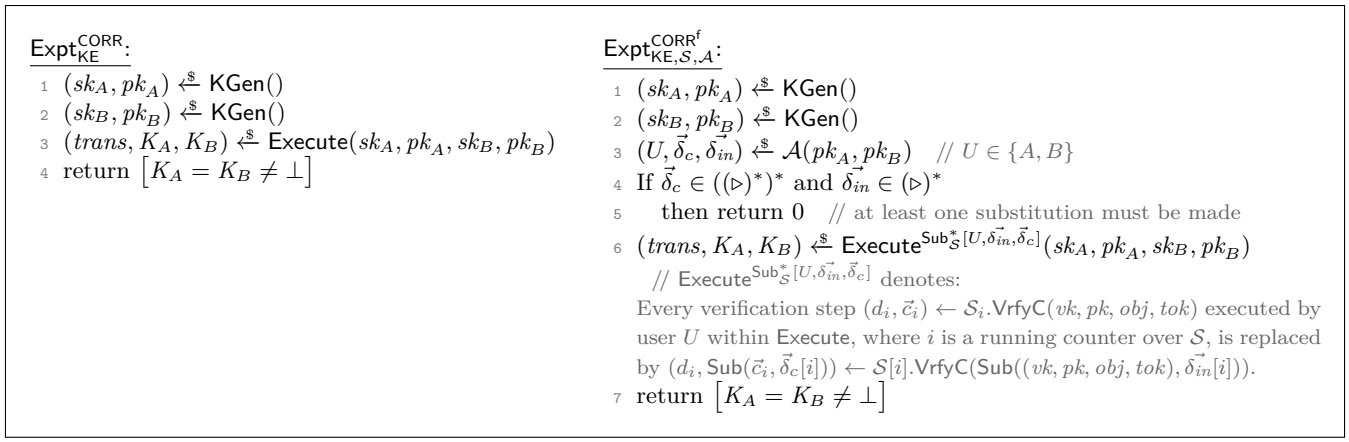

Figure 8: Correctness experiment, regular (left) and under faulty verification (right), for a key exchange protocol KE.

Definition 5.2 (Key exchange correctness)**.** *Let* KE *be a key exchange protocol and let experiment* $\mathsf{Expt}^{\mathsf{CORR}}_{\mathsf{KE}}$ *be defined as in Figure 8. We define the* correctness probability *of* KE *as*

$$\mathsf{Corr}_{\mathsf{KE}} := \Pr\left[\mathsf{Expt}^{\mathsf{CORR}}_{\mathsf{KE}} = 1\right].$$

We say that KE *is* always correct *iff* $\mathsf{Corr}_{\mathsf{KE}} = 1$.

To formally capture faulty verification steps, we now strengthen the classical correctness notion for key exchange by introducing an adversary $\mathcal{A}$ which is allowed to non-adaptively introduce errors in the execution of certain verification steps at one side of the protocol run. Concretely, this means that $\mathcal{A}$ may hard-code verification mistakes into the verification steps of a set of verification schemes $\mathcal{S}$, run within a key exchange protocol KE. As for our confirmation-code unpredictability notion (cf. Section 3), we consider both programming and input errors, modeled by modification vectors $\vec{\delta}_c$ resp. $\vec{\delta}_{in}$ in Figure 8. Correctness under faulty verification then measures whether a so-modified run of KE still yields matching keys with noticeable probability—which in practice would mean such mistakes might remain undetected.

Definition 5.3 (Key exchange correctness under faulty verification)**.** *Let* KE *be a key exchange protocol and let* $\mathcal{S}$ *be a vector of verification schemes with confirmation. Define experiment* $\mathsf{Expt}^{\mathsf{CORR}^{\mathsf{f}}}_{\mathsf{KE},\mathcal{S},\mathcal{A}}$ *for an adversary* $\mathcal{A}$ *as in Figure 8.*

We define the advantage of an adversary $\mathcal{A}$ *in achieving* correctness under faulty verification *for* KE *as*

$$\mathsf{Adv}^{\mathsf{CORR}^{\mathsf{f}}}_{\mathsf{KE},\mathcal{S},\mathcal{A}} := \Pr\left[\mathsf{Expt}^{\mathsf{CORR}^{\mathsf{f}}}_{\mathsf{KE},\mathcal{S},\mathcal{A}} = 1\right].$$

We say that KE *is* noticeably correct under faulty verification *in* $\mathcal{S}$ *iff there exists an adversary* $\mathcal{A}$ *such that* $\mathsf{Adv}^{\mathsf{CORR}^{\mathsf{f}}}_{\mathsf{KE},\mathcal{S},\mathcal{A}}$ *is non-negligible.*[6]

5.3 Detecting Verification Faults with a Generic Key Exchange Transform

We are now ready to introduce our generic transform of a key exchange protocol using classical verification schemes, into one that detects faulty verification steps using confirmation-augmented verification schemes. Our transform will be entirely oblivious to the type of verification schemes, which means it can be used to

[6]Strictly speaking we do not consider asymptotics in our definitions; we will nonetheless later give exact bounds for correctness under faulty verification which obey the common asymptotic behavior under reasonable assumptions.

$\mathsf{KE}_{\mathsf{F}}^{+c}.\mathsf{Execute}(sk_A, pk_A, sk_B, pk_B)$:

1 $(trans, K_A, K_B;\quad \vec{c}_A, \vec{c}_B) \xleftarrow{\$} \mathsf{KE}.\mathsf{Execute}^{\mathcal{S}\to\mathcal{S}^{+c}}(sk_A, pk_A, sk_B, pk_B)$
  // $\mathsf{KE}.\mathsf{Execute}^{\mathcal{S}\to\mathcal{S}^{+c}}$ denotes:
  Run $\mathsf{KE}.\mathsf{Execute}$ with scheme $\mathcal{S}[i]$ replaced by scheme $\mathcal{S}^{+c}[i]$, for $1 \le i \le |\mathcal{S}|$. Vectors $\vec{c}_A$ and $\vec{c}_B$ collect the sequences of confirmation codes $\vec{c}_i$ output by $\mathcal{S}^{+c}[i].\mathsf{CreateC}$ resp. $\mathcal{S}^{+c}[i].\mathsf{VrfyC}$ calls of user A resp. B, in order of the indices i.
2 For $U \in \{A, B\}$:
3 If $K_U \neq \bot$ then $K_U^{+c} \leftarrow \mathsf{F}(K_U, \vec{c}_U)$ // for some unambiguous encoding of $\vec{c}_U$ into a bitstring
4 Else $K_U^{+c} \leftarrow \bot$
5 return $(trans, K_A^{+c}, K_B^{+c})$

Figure 9: Generic transform $\mathsf{KE}_{\mathsf{F}}^{+c}$ of a key exchange protocol KE replacing the usage of verification schemes in vector $\mathcal{S}$ with corresponding confirmation-augmented versions in vector $\mathcal{S}^{+c}$. Key generation remains unmodified, i.e., $\mathsf{KE}_{\mathsf{F}}^{+c}.\mathsf{KGen} = \mathsf{KE}.\mathsf{KGen}$.

incorporate and check confirmation codes not only from classical signature- or MAC-based online authentication, but also from certificate checks at lower levels of the protocol execution (which were the sources of Apple's goto fail and other vulnerabilities [Pou14, Tea14, GIJ+12]), or elliptic curve parameter validation.

Formally, our transform $\mathsf{KE}_{\mathsf{F}}^{+c}$ of a key exchange protocol KE using an additional function F is given in Figure 9. Its approach is simple and modular: the original key exchange protocol KE is executed, replacing instances of some classical verification schemes $\mathcal{S}$ with confirmation-augmented versions $\mathcal{S}^{+c}$. By definition of being augmented versions (cf. Definition 2.3), this results in identical protocol executions except that, additionally, confirmation codes $\vec{c}_A$, $\vec{c}_B$ are computed by both parties involved in the protocol. Upon successful completion of the original key exchange, each party now takes their derived session key K and their collected confirmation codes $\vec{c}$ and computes the final session key as $\mathsf{F}(K, \vec{c})$. In our analysis, we will see that a simple collision-resistant PRF (e.g., HMAC [BCK96a, KBC97]) suffices for F; see Appendices A.2 and A.3 for the standard definitions of PRF security ($\mathsf{PRF\text{-}sec}$) and collision resistance (CR).

We emphasize that our transform has several desirable properties, namely, it is

- simple: beyond introducing confirmation-augmented verification schemes, it only adds a single function call F;
- non-intrusive: it does not change the operations of the original key exchange protocol; and
- transparent: it does neither modify any of the messages exchanged in the key exchange protocol nor introduce an additional message.

This makes our transform amenable to existing, deployed key exchange protocols, allowing to leverage prior analysis.

5.3.1 Noticeable Incorrectness

We now formalize and prove that our transform achieves its main goal: making certain implementation faults in verification steps visible via noticeable correctness failures.

Theorem 5.4. *Let KE be a canonical, always-correct key exchange protocol and F a collision-resistant function. Let $\mathcal{S}$, $\mathcal{S}^{+c}$ be two equal-size vectors of verification schemes such that $\mathcal{S}^{+c}[i]$, for $1 \le i \le |\mathcal{S}|$, is a* confirmation-augmented version *of $\mathcal{S}[i]$ (cf. Definition 2.3). Let all schemes in $\mathcal{S}^{+c}$ provide confirmation-value unpredictability.*

Then the key exchange protocol $\mathsf{KE}_{\mathsf{F}}^{+c}$ resulting from the generic transform in Figure 9 applied to KE is not noticeably correct under faulty verification. Concretely, we construct reductions $\mathcal{B}_1$ against the

collision resistance (CR*) of* F *and* $\mathcal{B}_{2,i}$ *against the confirmation-code unpredictability (*$\mathsf{c\text{-}UP}$*) of* $\mathcal{S}^{+c}[i]$ *(for* $1 \le i \le |\mathcal{S}|$*) such that*

$$\mathsf{Adv}^{\mathsf{CORR^f}}_{\mathsf{KE}^{+c}_{\mathsf{F}},\mathcal{S}^{+c},\mathcal{A}} \le \mathsf{Adv}^{\mathsf{CR}}_{\mathsf{F},\mathcal{B}_1} + \sum_{i=1}^{|\mathcal{S}|} \mathsf{Adv}^{\mathsf{c\text{-}UP}}_{\mathcal{S}^{+c}[i],\mathcal{B}_{2,i}}.$$

Proof. Recall that the goal of $\mathcal{A}$ in the $\mathsf{CORR^f}$ game is to make the executed key exchange sessions agree on the session keys ($K_A = K_B \neq \bot$) despite introducing some modification to the confirmation code outputs ($\vec{\delta}_c \notin ((\triangleright)^*)^*$) or the verification step inputs ($\vec{\delta_{in}} \notin (\triangleright)^*$) at user U.

Since KE is canonical, we can disregard any input modifications that make a verification step fail (i.e., output $d = 0$), as then $K_U = \bot$ and also $K^{+c}_U = \bot$ and $\mathcal{A}$ loses. Combining this with each $\mathcal{S}^{+c}[i]$ being a confirmation-augmented version of $\mathcal{S}[i]$, we have that verification outputs d_i do not change when replacing algorithms $\mathcal{S}[i].\mathsf{Vrfy}$ with $\mathcal{S}^{+c}[i].\mathsf{VrfyC}$, when $K_U \neq \bot$. As a result, we have that the outputs $(trans, K_A, K_B)$ of $\mathsf{KE.Execute}[\mathcal{S} \to \mathcal{S}^{+c}]$ equal those of $\mathsf{KE.Execute}$ (with the non-augmented schemes from $\mathcal{S}$, for the same choice of random coins). Since KE is always correct, this means that in the execution of the transformed key exchange $\mathsf{KE}^{+c}_{\mathsf{F}}$, we are guaranteed that $K_A = K_B$ in Line 1 of the transform (Figure 9), whenever $K_U \neq \bot$.

Next, we rule out collisions under F in Line 3 of $\mathsf{KE}^{+c}_{\mathsf{F}}.\mathsf{Execute}$, i.e., we let $\mathsf{Expt}^{\mathsf{CORR^f}}_{\mathsf{KE}^{+c}_{\mathsf{F}},\mathcal{S}^{+c},\mathcal{A}}$ abort whenever $K^{+c}_A = \mathsf{F}(K_A, \vec{c}_A) = \mathsf{F}(K_B, \vec{c}_B) = K^{+c}_B$ for distinct $\vec{c}_A \neq \vec{c}_B$. (Note that by the above, $K_A = K_B$.) The probability of such abort can directly be bounded by the advantage $\mathsf{Adv}^{\mathsf{CR}}_{\mathsf{F},\mathcal{B}_1}$ of a reduction $\mathcal{B}_1$ to the collision resistance of F, which simulates the $\mathsf{CORR^f}$ game for $\mathcal{A}$ and outputs $(K_A, \vec{c}_A)$, $(K_B, \vec{c}_B)$ as the collision under F if it occurs.

Finally, we rule out that $\vec{c}_A = \vec{c}_B$ in Line 1 of $\mathsf{KE}^{+c}_{\mathsf{F}}.\mathsf{Execute}$, within game $\mathsf{Expt}^{\mathsf{CORR^f}}_{\mathsf{KE}^{+c}_{\mathsf{F}},\mathcal{S}^{+c},\mathcal{A}}$, relying on the confirmation-value unpredictability of the schemes in $\mathcal{S}^{+c}$. Recall that by the definition of the $\mathsf{CORR^f}$ game, $\mathcal{A}$ must overwrite at least one confirmation-value output (via $\vec{\delta}_c$) or input (via $\vec{\delta_{in}}$) of some verification step of a scheme in $\mathcal{S}^{+c}$. Let E_i denote the event that the creation and verification steps corresponding to $\mathcal{S}^{+c}[i]$ yields equal confirmation codes for users A and B in the execution of $\mathsf{KE}^{+c}_{\mathsf{F}}.\mathsf{Execute}$, but the confirmation-value output or input of the verification step is overwritten. Note that for the overall confirmation code vectors to be equal ($\vec{c}_A = \vec{c}_B$), one of these events E_i must occur. We bound the probability of each event E_i individually by the confirmation-value unpredictability of $\mathcal{S}^{+c}[i]$, concretely by the advantage $\mathsf{Adv}^{\mathsf{c\text{-}UP}}_{\mathcal{S}^{+c}[i],\mathcal{B}_{2,i}}$ of a reduction $\mathcal{B}_{2,i}$.

To this end, $\mathcal{B}_{2,i}$ simulates the $\mathsf{CORR^f}$ game for $\mathcal{A}$ as follows. It obtains the public key pk for $\mathcal{S}^{+c}[i]$ in the $\mathsf{c\text{-}UP}$ game and uses this to compute pk_A and pk_B as per the definition of KE, sampling all other keys itself. (Recall that since KE is canonical, pk_A and pk_B only depend on the public key for $\mathcal{S}[i]$. Further, key generation for the augmented scheme $\mathcal{S}^{+c}[i]$ is the same as for $\mathcal{S}[i]$.) It then invokes $\mathcal{A}$ on pk_A, pk_B to obtain $(U, \vec{\delta}_c, \vec{\delta_{in}})$. If no overwriting values in $\vec{\delta}_c$, $\vec{\delta_{in}}$ corresponding to scheme $\mathcal{S}^{+c}[i]$ are set, i.e., event E_i is not triggered, $\mathcal{B}_{2,i}$ aborts. Otherwise, $\mathcal{B}_{2,i}$ extracts the overwriting values from $\vec{\delta}_c$, $\vec{\delta_{in}}$ for $\mathcal{S}^{+c}[i]$ into values $\vec{\delta}^*_c$, δ^*_{in}, and sets in to the input value for $\mathcal{S}^{+c}[i].\mathsf{CreateC}$ in the key exchange execution.

We now observe that the event E_i corresponds to the confirmation code $\vec{c}$ output by $\mathsf{CreateC}$ equaling that output by the VrfyC step executed by user U, despite $\vec{\delta}_c$, $\vec{\delta_{in}}$ overwriting verification input or output. This translates to, in the $\mathsf{c\text{-}UP}$ game, $\vec{c}$ being equal to $\vec{c}^*_{\text{prog}}$ (despite overwriting parts of the verification output) resp. to $\vec{c}^*_{\text{input}}$ (despite overwriting parts of the verification input), and $\mathcal{B}_{2,i}$ winning in the $\mathsf{c\text{-}UP}$ game. Hence, $\Pr[E_i] \le \mathsf{Adv}^{\mathsf{c\text{-}UP}}_{\mathcal{S}^{+c}[i],\mathcal{B}_{2,i}}$. Summing over all schemes in $\mathcal{S}$ yields the theorem bound's term.

At this point, we have established that the user confirmation codes in $\mathsf{KE}^{+c}_{\mathsf{F}}.\mathsf{Execute}$ must be distinct ($\vec{c}_A \neq \vec{c}_B$) which, having ruled out collisions under F, means that also their derived session keys are distinct, i.e., $K^{+c}_A \neq K^{+c}_B$. Hence now $\mathsf{Expt}^{\mathsf{CORR^f}}_{\mathsf{KE}^{+c}_{\mathsf{F}},\mathcal{S}^{+c},\mathcal{A}}$ always outputs 0, establishing the claim. □

5.3.2 Maintaining Security

It remains to argue that our generic transform maintains regular key exchange security, in the form of key indistinguishability under active attackers, if verification is implemented correctly.[7] The concrete flavor of key exchange security (e.g., whether to capture forward security, explicit authentication, extended version of key compromise, etc.) is highly dependent on the respective setting, and we hence refrain from narrowing down a specific formal security model. Instead, we provide a general argument for security within the fundamental game-based definition of key exchange security by Bellare and Rogaway [BR94] which forms the base of all modern key exchange security notions.

In the Bellare–Rogaway key exchange security model, the adversary is given the power to initiate many key exchange sessions among a large set of users, and arbitrarily interfere with the exchanged messages through tampering, dropping, or rerouting. It is further allowed to corrupt the long-term secret keys of users as well as to reveal the session keys established in sessions of its choice. In brief, key indistinguishability (IND security) then demands that such adversary is unable to distinguish a real session key from a uniformly random key in a challenge session (targeted via some $\mathcal{O}_{\mathsf{Test}}$ oracle query), conditioned that this session is not trivially compromised ("fresh") through reveal or corruption queries, and denoting its advantage against a protocol KE by $\mathsf{Adv}^{\mathsf{IND}}_{\mathsf{KE},\mathcal{A}}$.

It is within this basic security model that we provide our following security result. Interestingly, for technical reasons related to computing confirmation codes in our reduction, we are only able to show security is maintained generically when confirmation codes can be publicly computed. This covers a large class of practical protocols, especially for asymmetric and public verification schemes, including the signature and/or parameter checks in signed Diffie–Hellman and SIGMA [Kra03], checks for elliptic curve points in Bluetooth protocols [BT521], and TLS 1.3 [Res18] when treating handshake encryption modularly [DDGJ22]. Key exchange protocols with confirmation codes that cannot be publicly computed would require an individual treatment; we leave establishing a general result on handling non-public confirmation codes as an open question.

Theorem 5.5. *Let* KE *be a key exchange protocol satisfying Bellare–Rogaway key indistinguishability (*IND*) and* F *be a PRF. Let* $\mathcal{S}$, $\mathcal{S}^{+c}$ *be two equal-size vectors of asymmetric or public verification schemes such that* $\mathcal{S}^{+c}[i]$*, for* $1 \leq i \leq |\mathcal{S}|$*, is a* confirmation-augmented version *of* $\mathcal{S}[i]$ *(cf. Definition 2.3). Let* $\mathsf{KE}^{+c}_{\mathsf{F}}$ *be the key exchange protocol resulting from the transform in Figure 9 applied to* KE*. Assume the confirmation codes derived in each session can be publicly computed from the session's transcript and public keys.*

Then $\mathsf{KE}^{+c}_{\mathsf{F}}$ *also satisfies* IND *security. Concretely, we construct reductions* $\mathcal{B}_1$ *against the key indistinguishability (*IND*) of* KE *and* $\mathcal{B}_2$ *against the PRF security (*$\mathsf{PRF\text{-}sec}$*) of* F *such that*

$$\mathsf{Adv}^{\mathsf{IND}}_{\mathsf{KE}^{+c}_{\mathsf{F}},\mathcal{A}} \leq \mathsf{Adv}^{\mathsf{IND}}_{\mathsf{KE},\mathcal{B}_1} + \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{F},\mathcal{B}_2}.$$

Proof. We begin with the "real world" case of the IND game for $\mathsf{KE}^{+c}_{\mathsf{F}}$, i.e., the case where $\mathcal{A}$ receives the real session key K^{+c} computed in the tested session. Via one game hope, we bound $\mathcal{A}$'s advantage in distinguishing the "real" from the "random" case of the IND game.

In a first step, we modify this game to replace the session key K output by the underlying KE run in the test session (i.e., the session key derived in Line 1 of the transform, Figure 9) by a uniformly random key. We bound $\mathcal{A}$'s change in noticing this change by the IND security of KE: In a reduction $\mathcal{B}_1$, we relay all queries that $\mathcal{A}$ makes to the IND game for KE. Whenever $\mathcal{A}$ reveals a session key of $\mathsf{KE}^{+c}_{\mathsf{F}}$, $\mathcal{B}_1$ reveals the underlying key K in KE, derives the confirmation codes $\vec{c}$ for the session (which by assumption it can compute based on the public session transcript and user public keys) and returns $K^{+c} = \mathsf{F}(K, \vec{c})$. For the

[7] Note that detecting faulty verification is captured by triggering noticeable correctness errors, as shown in Theorem 5.4. Here, we ask that the confirmation codes and key derivation step via function F introduced by our transform do not infringe with security.

tested session, $\mathcal{B}_1$ likewise tests the underlying key in KE and performs the same computations. In case the IND game for KE returns the real key, this simulates the original game, whereas with a random key returned, $\mathcal{B}_1$ simulates the modified game, hence bounding this step by $\mathsf{Adv}^{\mathsf{IND}}_{\mathsf{KE},\mathcal{B}_1}$.

In the second step, we can now leverage that the key K obtained from KE in the test session is uniformly random and independent of other values in the execution. We use this to bound the advantage of $\mathcal{A}$ between this game and the "random world" of the IND game for $\mathsf{KE}^{+c}_{\mathsf{F}}$, where $\mathcal{A}$ receives a random key in response to its $\mathcal{O}_{\mathsf{Test}}$ query. We do so via a reduction $\mathcal{B}_2$ to the PRF security of F. Reduction $\mathcal{B}_2$ simulates the key exchange game for $\mathcal{A}$ as before, except that it replaces the F evaluation step in Line 3 of the transform in the test session by a call to its PRF oracle, using the (publicly computable) confirmation codes of the test session as label inputs to its oracle. Note that the now-random session key K of KE in the test session is only used within F, hence $\mathcal{B}_2$ can outsource sampling it to the PRF game. In case the PRF oracle responds with the real function evaluation, $\mathcal{B}_2$ simulates the previous game, otherwise, it simulates the random world of the IND game for $\mathsf{KE}^{+c}_{\mathsf{F}}$, resulting in the second bound, $\mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{F},\mathcal{B}_2}$. □

6 Conclusion

Our work takes a first step towards formally tying security to basic functionality in cryptographic protocols, in which, due to the brittleness of cryptographic implementations, critical errors are easily introduced and often remain unnoticed for long, like Apple's goto fail bug. Focusing on verification schemes like signatures, MACs, or parameter validation in elliptic curve groups, we introduced the concept of verifiable verification through confirmation codes that are output by verification algorithms, beyond the verification decision itself. When these confirmation codes achieve a non-adaptive notion of unpredictability, capturing a benign implementer accidentally skipping some verification steps, then they can be used to help implementers detect these implementation errors through basic functionality breaking on the protocol level. We exemplify this in the context of secure connections, providing and formally proving a simple transform for key exchange protocols. As concrete examples for verification schemes, we augment the $\mathsf{RSA\text{-}PSS}$ signature scheme, HMAC message authentication code, and elliptic curve point and subgroup membership validation with practical confirmation codes that essentially come "for free" as result of the actual verification steps.

Interesting next steps include exploring practical confirmation codes in other cryptographic settings performing verification. Natural candidates include, e.g., authenticated encryption (using integrity check codes to mask decrypted messages, scrambling the latter if verification is flawed), verifiable secret sharing (binding codes into reconstruction), and FO-based KEMs (confirming re-encryption steps are executed). For MAC schemes, implicitly verifying a MAC tag through only recomputing but not sending it appears to be a compelling approach for a confirmation code that does not come with an explicit verification decision (bit). On the protocol level, confirmation-code augmented primitives may be deployed, e.g., in code signing (requiring confirmation codes as input during installation), secure messaging (rejecting messages if codes do not match), entity authentication (using codes as challenge within challenge-response protocols), or even blockchain protocols (including codes in blocks to ensure miners verified prior blocks). Ultimately, the idea of tying security to basic functionality is not restricted to verification. Introducing it to other cryptographic settings could further help making implementing cryptography less fragile.

Acknowledgments

We thank Paul Rösler for extensive discussions about this work. We thank the anonymous reviewers for valuable comments. Felix Günther was supported in part by Research Fellowship grant GU 1859/1-1 of the German Research Foundation (DFG). This work has been co-funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297.

References

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 5–17, Denver, CO, USA, October 12–16, 2015. ACM Press. (Cited on page 3.)

[AHMP23] Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. Caveat implementor! Key recovery attacks on MEGA. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 190–218, Lyon, France, April 23–27, 2023. Springer, Heidelberg, Germany. (Cited on page 3.)

[AMPS18] Martin R. Albrecht, Jake Massimo, Kenneth G. Paterson, and Juraj Somorovsky. Prime and prejudice: Primality testing under adversarial conditions. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 281–298, Toronto, ON, Canada, October 15–19, 2018. ACM Press. (Cited on page 3.)

[AV96] Ross J. Anderson and Serge Vaudenay. Minding your p's and q's. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology – ASIACRYPT'96*, volume 1163 of *Lecture Notes in Computer Science*, pages 26–35, Kyongju, Korea, November 3–7, 1996. Springer, Heidelberg, Germany. (Cited on page 3.)

[BCK96a] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany. (Cited on pages 15 and 23.)

[BCK96b] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *37th Annual Symposium on Foundations of Computer Science*, pages 514–523, Burlington, Vermont, October 14–16, 1996. IEEE Computer Society Press. (Cited on page 16.)

[BCR+18] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, and Richard Davis. Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography, 4 2018. NIST Special Publication 800-56A, Revision 3. (Cited on page 20.)

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany. (Cited on page 4.)

[Bel06] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany. (Cited on pages 15 and 17.)

[Bel15] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision resistance. *Journal of Cryptology*, 28(4):844–878, October 2015. (Cited on pages 15 and 17.)

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228, New York, NY, USA, April 24–26, 2006. Springer, Heidelberg, Germany. (Cited on pages 18 and 20.)

[BFK+13] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press. (Cited on page 3.)

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *23rd Annual ACM Symposium on Theory of Computing*, pages 21–31, New Orleans, LA, USA, May 6–8, 1991. ACM Press. (Cited on page 6.)

[BFP22] Cecilia Boschini, Dario Fiore, and Elena Pagnin. Progressive and efficient verification for digital signatures. In *ACNS*, volume 13269 of *Lecture Notes in Computer Science*, pages 440–458. Springer, 2022. (Cited on page 7.)

[BH23] Nina Bindel and Britta Hale. A note on hybrid signature schemes. Cryptology ePrint Archive, Paper 2023/423, 2023. https://eprint.iacr.org/2023/423. (Cited on page 7.)

[BN19] Eli Biham and Lior Neumann. Breaking the bluetooth pairing - the fixed coordinate invalid curve attack. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019: 26th Annual International Workshop on Selected Areas in Cryptography*, volume 11959 of *Lecture Notes in Computer Science*, pages 250–273, Waterloo, ON, Canada, August 12–16, 2019. Springer, Heidelberg, Germany. (Cited on pages 3, 11, and 18.)

[BR94] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Heidelberg, Germany. (Cited on page 25.)

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In Ueli M. Maurer, editor, *Advances in Cryptology – EUROCRYPT'96*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416, Saragossa, Spain, May 12–16, 1996. Springer, Heidelberg, Germany. (Cited on page 13.)

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525, Santa Barbara, CA, USA, August 17–21, 1997. Springer, Heidelberg, Germany. (Cited on page 4.)

[BT521] Bluetooth core specification. Bluetooth Special Interest Group (SIG), July 2021. Ver. 5.3. (Cited on page 25.)

[Dam90] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany. (Cited on page 15.)

[DDGJ22] Hannah Davis, Denis Diemert, Felix Günther, and Tibor Jager. On the concrete security of TLS 1.3 PSK mode. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 876–906, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany. (Cited on page 25.)

[Fis03] Marc Fischlin. Progressive verification: The case of message authentication: (extended abstract). In Thomas Johansson and Subhamoy Maitra, editors, *Progress in Cryptology - INDOCRYPT 2003: 4th International Conference in Cryptology in India*, volume 2904 of *Lecture Notes in Computer Science*, pages 416–429, New Delhi, India, December 8–10, 2003. Springer, Heidelberg, Germany. (Cited on page 7.)

[Fis04] Marc Fischlin. Fast verification of hash chains. In Tatsuaki Okamoto, editor, *Topics in Cryptology – CT-RSA 2004*, volume 2964 of *Lecture Notes in Computer Science*, pages 339–352, San Francisco, CA, USA, February 23–27, 2004. Springer, Heidelberg, Germany. (Cited on page 7.)

[fIS18] Federal Office for Information Security. Elliptic curve cryptography, 6 2018. Technical Guideline BSI TR-03111, Version 2.10. (Cited on page 20.)

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany. (Cited on page 6.)

[GIJ+12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 38–49, Raleigh, NC, USA, October 16–18, 2012. ACM Press. (Cited on pages 3, 11, 21, and 23.)

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th Annual ACM Symposium on Theory of Computing*, pages 113–122, Victoria, BC, Canada, May 17–20, 2008. ACM Press. (Cited on page 6.)

[GPR14] Peter Gaži, Krzysztof Pietrzak, and Michal Rybár. The exact PRF-security of NMAC and HMAC. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 113–130, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany. (Cited on page 17.)

[Hen19] Nadia Heninger. Biased Nonce Sense: Lattice attacks against weak ECDSA signatures in the wild. Talk at the Workshop on Attacks in Cryptography 2 (WAC2), Crypto 2019. https://crypto.iacr.org/2019/affevents/wac/medias/Heninger-BiasedNonceSense.pdf, August 2019. Accessed April 27th, 2023. (Cited on page 3.)

[HNM+23] Sven Hebrok, Simon Nachtigall, Marcel Maehren, Nurullah Erinola, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. We really need to talk about session tickets: A large-scale analysis of cryptographic dangers with TLS session tickets. In *USENIX 2023*, 2023. (Cited on page 3.)

[JK03] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003. Obsoleted by RFC 8017. (Cited on pages 13 and 14.)

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151. (Cited on pages 15 and 23.)

[Kra03] Hugo Krawczyk. SIGMA: The "SIGn-and-MAc" approach to authenticated Diffie-Hellman and its use in the IKE protocols. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 400–425, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany. (Cited on pages 17 and 25.)

[Lan14a] Adam Langley. Early ChangeCipherSpec attack. https://www.imperialviolet.org/2014/06/05/earlyccs.html, June 2014. Accessed May 4th, 2023. (Cited on page 7.)

[Lan14b] Adam Langley. POODLE attacks on SSLv3. https://www.imperialviolet.org/2014/10/14/poodle.html, October 2014. Accessed April 29th, 2023. (Cited on page 3.)

[LKK19] Duc Viet Le, Mahimna Kelkar, and Aniket Kate. Flexible signatures: Making authentication suitable for real-time environments. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *ESORICS 2019: 24th European Symposium on Research in Computer Security, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 173–193, Luxembourg, September 23–27, 2019. Springer, Heidelberg, Germany. (Cited on page 7.)

[LL97] Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 249–263, Santa Barbara, CA, USA, August 17–21, 1997. Springer, Heidelberg, Germany. (Cited on page 3.)

[MC20] Nicky Mouha and Christopher Celi. Extending NIST's CAVP testing of cryptographic hash function implementations. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, volume 12006 of *Lecture Notes in Computer Science*, pages 129–145, San Francisco, CA, USA, February 24–28, 2020. Springer, Heidelberg, Germany. (Cited on page 6.)

[Mer90] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany. (Cited on page 15.)

[Nec97] George C. Necula. Proof-carrying code. In *POPL*, pages 106–119. ACM Press, 1997. (Cited on page 6.)

[NL97] George C. Necula and Peter Lee. Research on proof-carrying code for untrusted-code security. In *1997 IEEE Symposium on Security and Privacy*, page 204, Oakland, CA, USA, 1997. IEEE Computer Society Press. (Cited on page 6.)

[oST08] National Institute of Standards and Technology. The keyed-hash message authentication code (HMAC). Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 198-1, U.S. Department of Commerce, Washington, D.C., 2008. (Cited on page 15.)

[oST23a] National Institute of Standards and Technology. Cryptographic algorithm validation program. https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program, 2023. Accessed May 4th, 2023. (Cited on pages 3 and 6.)

[oST23b] National Institute of Standards and Technology. Digital signature standard (DSS). Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 186-5, U.S. Department of Commerce, Washington, D.C., 2023. (Cited on page 13.)

[Pou14] Kevin Poulsen. Behind iPhone's critical security bug, a single bad 'goto'. https://www.wired.com/2014/02/gotofail/, February 2014. Accessed April 27th, 2023. (Cited on pages 3, 11, 21, and 23.)

[Res18] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018. (Cited on pages 13 and 25.)

[Tea14] Synopsys Editorial Team. Understanding the gnutls certificate verification bug. https://www.synopsys.com/blogs/software-security/understanding-gnutls-certificate-verification-bug/, February 2014. Accessed April 27th, 2023. (Cited on pages 3, 11, 21, and 23.)

[TV21] Abdul Rahman Taleb and Damien Vergnaud. Speeding-up verification of digital signatures. *J. Comput. Syst. Sci.*, 116:22–39, 2021. (Cited on page 7.)

[VAS+17] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society. (Cited on page 3.)

[vW96] Paul C. van Oorschot and Michael J. Wiener. On Diffie-Hellman key agreement with short exponents. In Ueli M. Maurer, editor, *Advances in Cryptology – EUROCRYPT'96*, volume 1070 of *Lecture Notes in Computer Science*, pages 332–343, Saragossa, Spain, May 12–16, 1996. Springer, Heidelberg, Germany. (Cited on page 3.)

A Standard Definitions

A.1 Signature and MAC Schemes

Definition A.1 (Signature scheme)**.** *A* signature scheme $\mathsf{S} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vrfy})$ *with associated spaces for messages and signatures $\mathcal{M}$ resp. $\mathcal{S}$ consists of three efficient algorithms defined as follows.*

- $\mathsf{KGen}() \xrightarrow{\$} (sk, vk)$. *This probabilistic algorithm outputs a secret signing key sk and a public verification key vk.*
- $\mathsf{Sign}(sk, m) \xrightarrow{\$} \sigma$. *On input a signing key sk and a message $m \in \mathcal{M}$, this (possibly) probabilistic algorithm outputs a signature $\sigma \in \mathcal{S}$.*
- $\mathsf{Vrfy}(vk, m, \sigma) \to d$. *On input a verification key vk, a message m, and a signature σ, this deterministic algorithm outputs a decision bit $d \in \{0, 1\}$ (where $d = 1$ indicates validity of the signature).*

Correctness. We say that a signature scheme S is *correct*, if for any $m \in \mathcal{M}$ it holds that

$$\Pr\left[d = 1 \;\middle|\; (sk, vk) \xleftarrow{\$} \mathsf{KGen}(); \sigma \xleftarrow{\$} \mathsf{Sign}(sk, m); d \leftarrow \mathsf{Vrfy}(vk, m, \sigma)\right] = 1.$$

Definition A.2 (MAC scheme)**.** *A* message authentication code (MAC) scheme $\mathsf{M} = (\mathsf{KGen}, \mathsf{Tag}, \mathsf{Vrfy})$ *with associated spaces for messages and tags $\mathcal{M}$ resp. $\mathcal{T}$ consists of three efficient algorithms defined as follows.*

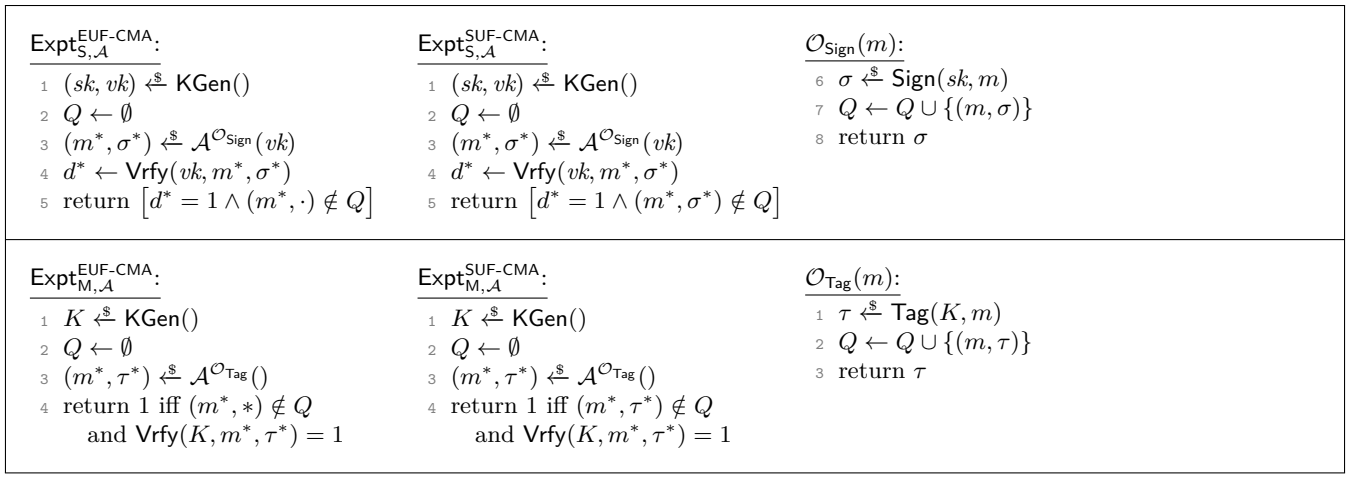

Figure 10: Security experiments for *existential and strong unforgeability under chosen-message attacks* (EUF-CMA, resp. SUF-CMA) for signature and MAC schemes. We write $(a, \cdot) \notin Q$ if $\nexists b$ s.t. $(a, b) \in Q$.

- $\mathsf{KGen}() \xrightarrow{\$} K$. *This probabilistic algorithm outputs a secret MAC key* K.
- $\mathsf{Tag}(K, m) \xrightarrow{\$} \tau$. *On input a MAC key* K *and a message* $m \in \mathcal{M}$*, this (possibly) probabilistic algorithm outputs a tag* $\tau \in \mathcal{T}$.
- $\mathsf{Vrfy}(K, m, \tau) \to d$. *On input a MAC key* K*, a message* m*, and a tag* τ*, this deterministic algorithm outputs a decision bit* $d \in \{0, 1\}$ *(where* $d = 1$ *indicates validity of the MAC).*

Correctness. We say that a MAC scheme M is *correct*, if for any $m \in \mathcal{M}$ it holds that

$$\Pr\left[d = 1 \mid K \xleftarrow{\$} \mathsf{KGen}(); \tau \xleftarrow{\$} \mathsf{Tag}(K, m); d \leftarrow \mathsf{Vrfy}(K, m, \tau)\right] = 1.$$

Definition A.3 (Existential and strong unforgeability of signature schemes)**.** *Let* X *be a signature resp. MAC scheme and let experiments* $\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{X,\mathcal{A}}$ *and* $\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{X,\mathcal{A}}$ *for an adversary* $\mathcal{A}$ *be defined as in Figure 10, for signatures resp. MAC schemes.*

We define the advantage of an adversary $\mathcal{A}$ *against the* existential (resp. strong) unforgeability under chosen-message attacks *(*EUF-CMA*, resp.* SUF-CMA*) of* X *as*

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{X,\mathcal{A}} := \Pr\left[\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{X,\mathcal{A}} = 1\right], \quad \textit{resp.} \quad \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{X,\mathcal{A}} := \Pr\left[\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{X,\mathcal{A}} = 1\right].$$

A.2 Pseudorandom Functions

Definition A.4 (Pseudorandom function (PRF))**.** *Let* $\mathsf{F}\colon X \times Y \to Z$ *be a function. We define the advantage of an adversary* $\mathcal{A}$ *against the PRF security (*PRF-sec*) of* F *as*

$$\mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{F},\mathcal{A}} := \Pr\left[\mathcal{A}^{\mathsf{F}(K,\cdot)} = 1\right] - \Pr\left[\mathcal{A}^{\mathsf{R}(\cdot)} = 1\right],$$

where $K \xleftarrow{\$} X$ *and* R *is randomly sampled from all functions mapping* Y *to* Z.

A.3 Collision Resistance

Definition A.5 (Collision resistance)**.** *Let* $\mathsf{F}\colon X \to Y$ *be a function. We define the advantage of an adversary* $\mathcal{A}$ *against the collision resistance (*CR*) of* F *as*

$$\mathsf{Adv}^{\mathsf{CR}}_{\mathsf{F},\mathcal{A}} := \Pr\left[\mathsf{F}(x) = \mathsf{F}(x') \wedge x \neq x' \mid (x, x') \xleftarrow{\$} \mathcal{A}\right].$$