

# Unclonable Polymers and Their Cryptographic Applications

Ghada Almashaqbeh<sup>1</sup>, Ran Canetti<sup>2</sup>, Yaniv Erlich<sup>3</sup>, Jonathan Gershoni<sup>4</sup>,  
Tal Malkin<sup>5</sup>, Itsik Pe'er<sup>5</sup>, Anna Roitburd-Berman<sup>4</sup> and Eran Tromer<sup>4,5</sup>

<sup>1</sup> University of Connecticut, [ghada@uconn.edu](mailto:ghada@uconn.edu)

<sup>2</sup> Boston University, [canetti@bu.edu](mailto:canetti@bu.edu)

<sup>3</sup> Eleven Therapeutics and IDC Herzliya, [erlichya@gmail.com](mailto:erlichya@gmail.com)

<sup>4</sup> Tel Aviv University, [gershoni@tauex.tau.ac.il](mailto:gershoni@tauex.tau.ac.il), [roitburda@gmail.com](mailto:roitburda@gmail.com)

<sup>5</sup> Columbia University, [{tal,itsik,tromer}@cs.columbia.edu](mailto:{tal,itsik,tromer}@cs.columbia.edu)

**Abstract.** We propose a mechanism for generating and manipulating protein polymers to obtain a new type of *consumable storage* that exhibits intriguing cryptographic “self-destruct” properties, assuming the hardness of certain polymer-sequencing problems.

To demonstrate the cryptographic potential of this technology, we first develop a formalism that captures (in a minimalistic way) the functionality and security properties provided by the technology. Next, using this technology, we construct and prove security of two cryptographic applications that are currently obtainable only via trusted hardware that implements logical circuitry (either classical or quantum). The first application is a password-controlled *secure vault* where the stored data is irrecoverably erased once a threshold of unsuccessful access attempts is reached. The second is (a somewhat relaxed version of) *one-time programs*, namely a device that allows evaluating a secret function only a limited number of times before self-destructing, where each evaluation is made on a fresh user-chosen input.

Finally, while our constructions, modeling, and analysis are designed to capture the proposed polymer-based technology, they are sufficiently general to be of potential independent interest.

## 1 Introduction

Imagine we could cryptographically create *k-time programs*, i.e., programs that can be run only some bounded number of times, and inherently self-destruct after the  $k$ -th invocation. This would open the door to a plethora of groundbreaking applications: For instance, we would be able to use even low-entropy passwords for offline data storage, because  $k$ -time programs could lock out a brute-force-search adversary after a few attempts; today this is possible only via interaction or trusted electronics.

Alternatively, we could release a sensitive and proprietary program (such as a well-trained ML model) and be guaranteed that the program can be used only a limited number of times, thus potentially preventing over-use, mission-creep, or reverse engineering. Such programs can also be viewed as a commitment to a potentially exponential number of values, with a guarantee that only few of these values are ever opened.

Indeed,  $k$ -time programs, first proposed by Goldwasser, Kalai, and Rothblum [42] are extremely powerful. What does it take to make this concept a reality? Obviously, we cannot hope to do that with pure software or classical information alone, since these are inherently cloneable. In fact, software-only  $k$ -time programs do not exist even if the program can use quantum gates [16]. In [42] it is shown that “one-out-of-two” memory gadgets, which guarantee that exactly one out of two pieces of data encoded in the gadget will be retrievable, along with circuit garbling techniques [66], suffice for building  $k$ -time programs for *any* functionality.

However, how do we obtain such memory gadgets? While Goldwasser et al. suggest a number of general directions, we are not aware of actual implementations of one-out-of-two memory gadgets other than generically tamper-proofing an entire computational component.

Can alternative technologies be explored? Also, what can be done if we only can obtain some weaker forms of such memory gadgets, that provide only limited retrievability to naive users, along with limited resilience to adversarial attacks?

More generally, where can we look for such technologies, and how can we co-develop the new technology together with the cryptographic modeling and algorithmics that will complement the technology to obtain full-fledged  $k$ -time programs, based only on minimal and better-understood assumptions on the physical gadgets, rather than by dint of complex defensive engineering?

## 1.1 Contributions

This work describes a cross-disciplinary effort to provide some answers to these questions, using ideas based on the current technological capabilities and limitations in synthesizing and identifying *random proteins*. We begin with a brief overview of the relevant biochemical technology and our ideas for using this technology for bounded-retrieval information storage. We then describe our algorithmic and analytical work towards constructing  $k$ -time programs and related applications, along with rigorous security analysis based on well-defined assumptions on the adversarial capabilities—both biochemical and computational.

**Biochemical background.** Advances in biotechnology have allowed the custom-tailored synthesis of biological polymers for the purpose of data storage. Most effort has focused on DNA molecules, which can be synthesized as to encode digital information in their sequence of bases. DNA can be readily cloned and read with excellent fidelity, both by nature and by existing technology [14, 24, 32, 45]. Even minute amounts of DNA can be reliably cloned—and then read—an effectively unbounded number of times, making it an excellent storage medium—too good, alas, for our goal, since it is unclear how to bound the number of times a DNA-based storage can be read.

Consider, though, a different biological polymer: proteins. These chains of amino acids can likewise represent digital information, and can be synthesized via standard (albeit more involved) lab procedures. However, reading (“sequencing”) the amino acid sequence in a protein appears much more difficult: the best known lab procedure for sequencing general proteins is mass spectrometry, which requires a macroscopic pure sample, free of substantial pollution. The sequencing process then destroys the sample—the protein is chopped into small fragments which are accelerated in a detector.

Furthermore, we have no way to clone a protein that is given in a small amount. Indeed, Francis Crick’s central dogma of molecular biology states: “*once ‘information’ has passed into protein it cannot get out again. [Information] transfer from protein to protein, or from protein to nucleic acid is impossible*” [25]. Over billions of years of evolution, no known biological system has ever violated this rule, despite the reproductive or immunological benefits this could have bestowed. Moreover, in the 63 years since that bold hypothesis (or, alternatively, challenge) was put forth, it has also stymied human ingenuity, in spite of the enormous usefulness to science and medicine that such ability would provide.

This makes proteins terrible as a general-purpose data storage medium: they cannot be read unless presented in just the right form, and they self-destruct after few reads. However, cryptography is the art of making computational lemonade out of hard lemons. Can we leverage the time-tested hardness of sequencing small amounts of proteins for useful functionality? We see a couple of approaches, leading to different functionality and applications.

**Biochemical “conditionally retrievable memory”.** As a first attempt, we consider a protein-based “conditionally retrievable memory”, that stores information in a way so that retrieving the information requires knowledge of some key, and furthermore, once someone attempts to retrieve the information “too many times” with wrong keys, the information becomes irrevocably corrupted. A first attempt at implementing such a system may proceed as follows: The sender encodes the payload information into a *payload* protein, and the key into a *header* protein, which are connected into a single protein (the concrete encoding and procedures is discussed in Section 2). The process actually creates a macroscopic amount of such payload-header pairs, and mixes these pairs with a large quantity of *decoys* which are similarly structured but encode random keys and payloads. The resulting sample is then put in a vial, serving the role of (biological) memory.

Recovering the information from the vial can be done via a *pull-down* procedure, i.e., a chemical reaction of the sample with an antibody that attaches to a specific portion of the protein. Given the key, one can choose the correct antibody and use it to isolate the information-bearing proteins from the added ones. Then, the information can be read via mass spectrometry.

In addition, *any* meaningful attempt to obtain information from the vial would necessarily employ some sort of pull-down on some portion of the sample in the vial, and then employ mass spectrometry on the purified portion of the sample. (Indeed, performing mass spectrometry on the vial without *pull-down* will return results that are polluted by the decoys.) Furthermore, since each application of the spectrometry process needs, and then irrevocably consumes, some fixed sample mass, an adversary is effectively limited to trying some bounded number  $n$  of guesses for the key, where  $n$  depends on the initial mass of the sample in the vial and the grade of the specific spectrometer used.

**Partially retrievable memory.** The above scheme appears to be easily adaptable to the case of storing multiple key-payload pairs in the same vial, along with the random noise proteins. This variant has the intriguing feature that even a user that knows all keys can only obtain  $n$  payloads from the vial, where  $n$  is the number of pull-down-plus-mass-spectrometry operations that can be applied to the given sample.

**Challenges.** While the above ideas seem promising, they still leave a lot to be desired as far as a cryptographic scheme is concerned: First, we would need a more precise model that adequately captures the capabilities required from honest users of the system, as well as bounds on the feasible capabilities of potential adversaries—taking into account that adversaries might have access to significantly more high-end bio-engineering and computational tools than honest users. Next, we would need to develop algorithmic techniques that combine bio-engineering steps and computational steps to provide adequate functionality and security properties. Finally, we would need to provide security analysis that rigorously asserts the security properties within the devised model. We describe these steps next.

**Formal modeling: Consumable tokens.** The full biochemical schemes we propose involve multiple steps and are thus difficult to reason about formally. We thus distill the requisite functionality and security properties into a relatively simple idealized definition of a *consumable token* in Section 4. In a nutshell, an  $(1, n, v)$ -time token is created with  $2v$  values: keys  $k_1, \dots, k_v$  and messages  $m_1, \dots, m_v$ , taken from domains  $K$  and  $M$ , respectively. Honest users can query a token only once, with key  $k'$ . If  $k' = k_i$  for some  $i$ , then the user obtains  $m_i$ , else the user obtains  $\perp$ . Adversaries can query a token  $n$  times, each with a new key  $k'$ . Whenever any of the keys equals  $k_j$ , the adversary obtains  $m_j$ . We assume that the size of  $M$ ,  $K$  and  $v$  are fixed, independent of any security parameter.

**Constructing consumable tokens.** Our biochemical procedures provide a candidate construction for consumable tokens, but with weak parameters. They can only store a few messages, of short length, under short keys, with non-negligible completeness and soundness errors. This is in addition to the power gap between an honest recipient and an adversarial one; the former can perform *one* data retrieval attempt, while the latter might be able to perform up to  $n$  queries, for some small integer  $n$ .

Thus, employing our protein-based consumable tokens in any of the applications discussed above is not straightforward. It requires several (conventional and new) techniques to mitigate these challenges. Amplifying completeness is handled by sending several vials, instead of one, all encoding the same message. Storing long messages is handled by fragmenting a long message into several shorter ones, each of which is stored under a different header in a separate vial. The rest are more involved and were impacted by the application itself.

**Bounded query, point function obfuscation for low-entropy passwords.** Password-protected secure vaults, or digital lockers, allow encrypting a message under a low entropy password. This can be envisioned as a point function with multi-bit output where the password is the point and the message is the output. With our consumable tokens, one can store the message inside a vial with the password being mapped to a token key (or header) that is used to retrieve the message. The guarantee is that an honest recipient, who knows the password, will be able to retrieve the message using one query. While an adversary can try up to  $n$  guesses after which the token will be consumed.

However, having a non-negligible soundness error complicates the matter. We cannot use the conventional technique of sharing the message among several vials, and thus reducing the error exponentially. This is due to the fact that we have one password mapped to the keys of these tokens, so revealing the key of any of these tokens would give away the password. We thus devise a chaining technique, which effectively forces the adversary to operate on the tokens sequentially. In Section 5, we start with formalizing an ideal functionality for bounded-query point function obfuscation, and then detail our consumable token and chaining based construction, along with formal security proofs.

**$(1, n)$ -time programs.** Next we use  $(1, n, v)$ -consumable tokens to construct  $(1, n)$ -time programs, namely a system that, given a description of a program  $\pi$ , generates some digital rendering  $\hat{\pi}$  of  $\pi$ , and a number of consumable tokens, that (a) allows a user to obtain  $\pi(x)$  on any value  $x$  of the user’s choice, and (b) even an adversary cannot obtain more information from the combination of  $\hat{\pi}$  and the physical tokens, on top of  $\pi(x_1), \dots, \pi(x_n)$  for  $n$  adversarially chosen values  $x_1, \dots, x_n$ .

In the case of  $n = 1$  (i.e., when even an adversary can obtain only a single message out of each token),  $(1, 1)$ -time programs can be constructed by garbling the program  $\pi$  and then implementing one-out-of-two

oblivious transfer for each input wire using a  $(1, 1, 2)$ -consumable token with  $K = M = \{0, 1\}^k$  [42]. However, constructing  $(1, n')$ -time programs from  $(1, n, v)$ -consumable tokens with  $n > 1$  turns out to be a significantly more challenging problem, even when  $v$  is large and even when  $n'$  is allowed to be significantly larger than  $n$  (i.e., even when the bound that the construction is asked to impose on the number of  $x_i$ 's for which the adversary obtains  $\pi(x_i)$  is significantly larger than the number of messages that the adversary can obtain from each token): A first challenge is that plain circuit garbling provides no security as soon as it is evaluated on more than a single input (in fact, as soon as the adversary learns both labels of some wire). Moreover, even if one were to use a “perfect multi-input garbling scheme” (or, in other words VBB obfuscation [10]), naive use of consumable tokens would allow an adversary to evaluate the function on an exponential number of inputs.

Our construction combines the use of general program obfuscation (specifically, Indistinguishability Obfuscation [10, 50]) together with special-purpose encoding techniques that guarantee zero degradation in the number of values that an adversary may obtain—namely  $(1, n)$ -time programs using our consumable tokens.

Specifically, our construction obfuscates the circuit, and uses consumable tokens to store random secret strings each of which represents an input in the circuit input domain. Without the correct strings, the obfuscated circuit will output  $\perp$ . Beside amplifying soundness error (luckily it is based on secret sharing for this case), our construction employs an innovative technique to address a limitation imposed by the concrete construction of consumable tokens. That is, a token can store a limited number of messages (or random strings), thus allowing to encode only a subset of the circuit inputs rather than the full input space. We use linear error correcting codes to map inputs to codewords, which are in turn used to retrieve random strings from several tokens.

We show a number of flavors of this construction, starting with a simple one that uses idealized (specifically VBB) obfuscation, followed by a more involved variant that uses only indistinguishability obfuscation  $i\mathcal{O}$ . We also discuss how reusable garbled circuits [41] can be used to limit the use of  $i\mathcal{O}$  to a smaller and simpler circuits.

**Protection from malicious encapsulators.** Our constructions provide varying degrees of protection for an honest evaluator in face of potentially ill-structured programs. The  $(1, n)$ -point function obfuscation application carries the guarantee that an adversary can only obfuscate (or encapsulate) valid point functions with the range and domain specified. This is due to the fact that we use consumable tokens each of which is storing one secret message  $m$  (from a fixed domain) under a single token key (from a fixed space). The use of a wrong key (i.e., one that is not derived correctly from the password that an honest evaluator knows) will return  $\perp$ . The general  $(1, n)$ -program application only guarantees that the evaluator is given *some* fixed program, but without guarantees regarding the nature of the program. Such guarantees need to be provided in other means. A potential direction is to provide a generic non-interactive zero knowledge proof that the encapsulated program along with the input labels belong to a given functionality or circuit class.

**The analytical model.** We base our formalism and analysis within the UC security framework [19]. This appears to be a natural choice in a work that models and argues about schemes that straddle two quite different models of computation, and in particular attempt at arguing security against attacks that combine bioengineering capabilities as well as computational components. Specifically, when quantifying security we use separate security parameters: one for the bioengineering components and one for the computational ones. Furthermore, while most of the present analysis pertains to the computational components, we envision using the UC theorem to argue about composite adversaries and in particular construct composite simulators that have both bioengineering and computational components.

## 1.2 Related Work

Katz et al. [52] initiated the study of tamper-proof hardware tokens to achieve UC security for MPC protocols in the plain model. Several follow up works explored this direction, e.g., [23, 47, 48], with a foundational study in [44]. In general, two types of tokens are used: stateful [28] and stateless (or resettable) [8, 27]; the latter is considered a weaker and more practical assumption than the former. In another line of work, Goldwasser et al. [42] employed one-time memory devices to build one-time programs as mentioned before. They assume that such memory devices exist without showing any

concrete instantiation. Our work instead provides an instantiation for a weaker version of memory devices— $(1, n)$ -time memory devices—and uses them to build  $(1, n)$ -time programs. Other works relied on tamper-proof smart cards to construct functionalities such as anonymous authentication and practical MPC protocols [46, 54]. They assume that such cards withstand reverse-engineering or side-channel attacks. Our work, on the other hand, proposes an alternative that relies on deeper, more inherent physical phenomena that have withstood the test of nature and ingenuity. We show that even a weak level of security and functionality, far below the natural smart-card trust assumption, suffices for useful cryptographic functionalities.

Quantum computing offer an unclonability feature that poses the question of whether it can offer a solution for bounded program execution. This possibility was ruled out by Broadbent et al [16] who proved that one-time programs, even in the quantum model, cannot be constructed without one-time memory devices. To circumvent this impossibility, Roehsner et al. [61] introduced a relaxed notion—probabilistic one-time programs—allowing for some error in the output, and showed a construction in the quantum model without requiring hardware tokens. Secure software leasing (SSL) [5] emerged as a weaker alternative for quantum copy-protection [1]. SSL deals with software piracy for quantum unlearnable circuits; during the lease period the user can run the program over any input, but not after the lease expires. Our work bounds the number of executions a user obtains regardless of the time period and can be used for learnable functions.

Another line of research explored basing cryptography on physical assumptions. For example, noisy channels [26] and tamper-evident seals [57] were used to implement oblivious transfer and bit commitments. Others built cryptographic protocols for physical problems: [38] introduced zero knowledge proof system for nuclear warhead verification and [34] presented a unified framework for such proof systems with applications to DNA profiling and neutron radiography. This has been extended in [35] to build secure physical computation in which parties have private physical inputs and they want to compute a function over these inputs. Notably, [35] uses *disposable circuits*; these are hardware tokens that can be destroyed (by the opposing party) after performing a computation. In comparison to all these works, our consumable tokens are weaker as they are used for storing short messages rather than performing a computation.

Physical unclonable functions (PUFs) [60] are hardware devices used as sources of randomness, that cannot be cloned. PUFs found several applications, such as secure storage [30], key management [51], oblivious transfer [62], and memory leakage-resilient encryption schemes [7]. The works [17] and [59] proposed models for using trusted and malicious PUFs, respectively, in the UC setting. Our tokens share the unclonability feature with PUFs, but they add the bounded query property and the ability to control the output of a data retrieval query.

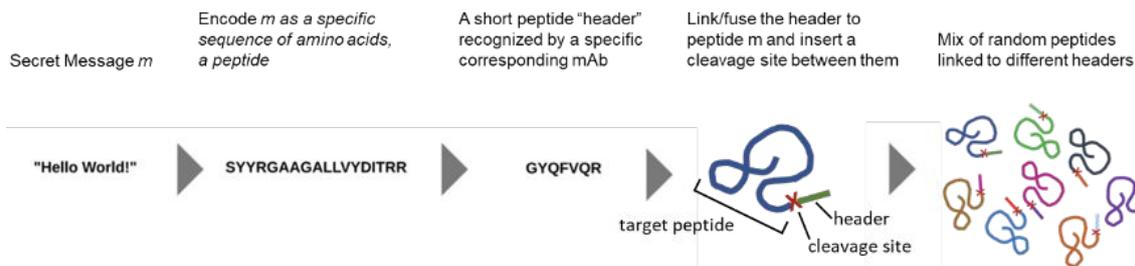
Lastly, a few works investigated the use of DNA in building cryptographic primitives and storage devices. For example, a DNA-based encryption scheme was proposed in [67], while [31] focused on bio-data storage that deteriorates with time by utilizing engineered sequences of DNA and RNA, without any further cryptographic applications. Both works do not provide any formal modeling or security analysis. To the best of our knowledge, we are the first to use unclonable biological polymers—proteins—to build advanced cryptographic applications with formal treatment. Apart from storage, a more ambitious view was posed by Adleman [3] back in the 1990s, who investigated the concept of *molecular computers*. They showed how biochemical interactions can solve a combinatorial problem over a small graph encoded in molecules of DNA [2]. This leaves an open question of whether one can extend that to proteins and build stronger tokens that can securely execute a full computation.

## 2 Unclonable Polymer-based Data Storage

In this section, we present an overview of the protein-based data storage construction that we use to build consumable tokens. We focus on the specifications and guarantees this construction provides rather than detailed explanation of the biology behind them. The detailed explanation, and a more complete version of this section, can be found in Appendix A.<sup>6</sup>

**Protein-based data storage and retrieval.** Advances in biotechnology have allowed the custom-tailored synthesis of biological polymers for the purpose of data storage. Much of the effort in this new

<sup>6</sup>It should be noted that we are working on a sister paper showing the details of this biological construction; will delve into the technical details of the biochemical realization and empirically analyze it under the framework established in this paper.



**Fig. 1.** General scheme for peptide-based data storage.

field has focused on the use of DNA, generating an arsenal of molecular protocols to store and retrieve information [14, 24, 32, 45]. With this growing application, we became interested in the cryptographic attributes this new hardware offers. Specifically, we propose the use of proteins, in particular short amino-acid polymers or peptides, as a data storage material. Curiously, the most fundamental characteristics of proteins; they cannot be directly cloned nor can they replicate or be amplified, and that "data retrieval" is typically self-destructive, might be considered as limitations from a regular data storage point of view. However, these exact traits can confer powerful features to instantiate cryptographic primitives and protocols.

Accordingly, for storage, the digital message is encoded into the primary configuration of the peptide/protein, i.e., the sequence of the 20 natural amino acids of the protein material, the "peptide-payload". To retrieve the message, the order of the amino-acids of a protein is determined, after which this sequence is decoded to reconstruct the original message. Given that our primary goal is to design a biological machinery to securely realize cryptographic primitives, we extend this basic paradigm to support data secrecy. Our proposal is based on a number of features of proteins and peptides: (i) unique peptides can be designed to comprise any string of amino acids and be physically produced with precision and at high fidelity, (ii) a peptide sample whose amino acid sequence is not known is unclonable and cannot be replicated or amplified, (iii) sequencing the peptide results in its consumption.<sup>7</sup>

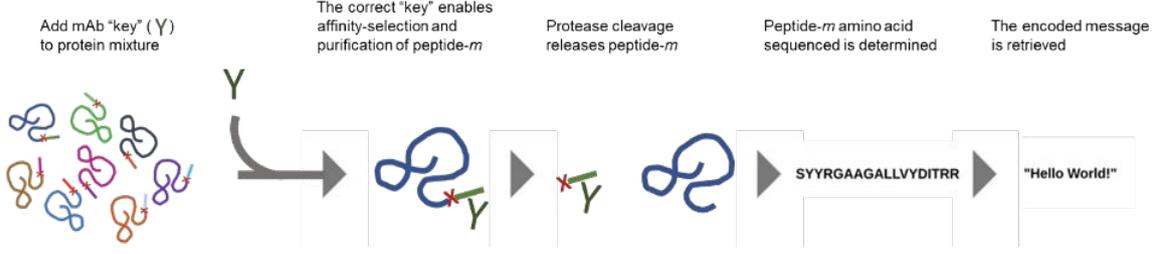
As illustrated in Figure 1, the peptide message, *peptide-m*, is conjugated to a short (< 10 amino acids) peptide tag, a tag that is recognized specifically by a predetermined monoclonal antibody (mAb). Thus, the peptide tag, designated "header", corresponds to its specific mAb. Next, *peptide-m* is mixed with a vast variety of decoy peptide messages, all of which are peptide permutations of composition and length, each conjugated to a collection of alternative header sequences. The sender shares the secret header with the recipient, i.e., the peptide sequence of the header (this is digital data), which reveals to the recipient the identity of the correct unique mAb to be used to recover *peptide-m*. Then he sends a vial of the protein mix (a physical component).

For data retrieval, as shown in Figure 2, the only possible way to decode the message is to first single out and purify *peptide-m*. This can be achieved by employing the unique mAb that specifically recognizes the unique header attached to *peptide-m*. Note that all decoy peptides and the target *peptide-m* are of the same general length, mass, and composition, but differing in sequence. Thus, effective purification of the desired protein from the decoys, without the matching mAb, is impossible through standard biochemical/biophysical methodologies. This achieves message secrecy in the sense that without the matching mAb,  $m$  cannot be retrieved.

**Biochemical properties.** Protein-based data storage enjoys several properties that we exploit in our cryptographic applications. These include the following (this is a high level description, more details on the biochemical features that supports these properties can be found in Appendix A):

- *Unclonability.* Proteins are unclonable biological polymers, meaning that given an amount of proteins one cannot replicate it to obtain a larger amount.
- *Destructive data retrieval.* Modern biology is only capable of reading protein sequences indirectly, destructively, and at lower throughput compared to DNA. The main practical strategy for reading

<sup>7</sup>Although we talk about one message in these protocols, several messages can be stored in one sample by having several *peptide-ms* instead of one, each of which is conjugated with a unique header and mixed with the decoys.



**Fig. 2.** Message retrieval.

proteins is mass spectrometry (MS) or versions thereof [9, 39]. This machinery imposes several conditions on the protein sample to allow retrieving the digital data. First, the sample must contain a sufficient amount of the target protein, and second, the sample must be pure enough. Once a vial is purified and read using MS, the structure of the protein is destructed due to fragmentation.

- *Adversarial interactions.* The only known way to retrieve any information about the data stored in a vial is by pulling-down the target protein using the key (or mAb), and then sequencing this protein using MS. Thus, an adversary, who does not know the correct mAb, can only guess a candidate mAb and check if sequencing will output  $m$ . Also, when obtaining several (independent) tokens, the adversary will operate on these tokens separately, since purification and sequencing are still needed to obtain the stored data.
- *Bounded query.* The previous properties imply that a protein-based data storage allows for a finite number of data retrieval attempts after which the vial is consumed, i.e., each data retrieval attempt destroys a portion of the biological material. In our model, we account for that fact that an adversary could be more powerful than an honest recipient, e.g., she owns more advanced MS that operates at lower thresholds. This implies that the vial will allow the adversary to perform multiple data retrieval attempts, denoted as  $n$ , but an honest recipient will perform only one.
- *Message and key (header) sizes.* Proteins can store relatively short messages using short headers. In Appendix A, we show how to use fragmentation to store a long message using several vials instead of one, such that the header will be the concatenation of all headers used in these vials. Nonetheless, in our applications, we use consumable tokens to store cryptographic keys rather than very long messages.
- *Completeness and soundness errors.* Due to laboratory experimental (human and machine) errors, the protein-based data storage may have non-negligible completeness and soundness errors. The former means that despite the use of the correct mAb, the target message may not be successfully retrieved. While the latter means that despite the use of an incorrect mAb, an adversary may manage to recover  $m$ . In other words, these incorrect mAb may have similar features to the correct one (what we call close keys). We amplify the completeness error on the biology side (by sending several vials all encoding the same message),<sup>8</sup> while we amplify the soundness error as part of the cryptographic constructions that we build in later sections.

### 3 Preliminaries

In this section, we recall preliminary notions needed in this work. This include the virtual blackbox and indistinguishability obfuscation, and injective noninteractive commitments. For background on universally composable (UC) security, we refer the reader to [18] for a formal treatment.

#### 3.1 Obfuscation

We recall the definitions of virtual blackbox (VBB) and indistinguishability obfuscation ( $i\mathcal{O}$ ).

**Definition 1 (VBB Obfuscation).** *A circuit class  $\{\mathcal{C}_\kappa\}$  is obfuscatable if there exists an algorithm  $\mathcal{O}$  that takes a circuit  $C \in \mathcal{C}_\kappa$  and outputs another circuit  $C'$  such that the following conditions are satisfied:*

<sup>8</sup>At the cryptography level this is still viewed as one token that allows the honest recipient to retrieve the message with all but negligible probability

- *Correctness:* For all security parameters  $\kappa \in \mathbb{N}$ , for all  $C \in \mathcal{C}_\kappa$ , and for all inputs  $x \in \{0, 1\}^*$ , we have that

$$\Pr [C'(x) = C(x) : C' \leftarrow \mathcal{O}(1^\kappa, C)] = 1$$

- *Efficiency:* There exists a polynomial  $p$  such that for all  $\kappa \in \mathbb{N}$ , and all  $C \in \mathcal{C}_\kappa$ , we have that  $|\mathcal{O}(C)| \leq p(|C|)$  and if  $C$  takes  $t$  time steps on an input  $x \in \{0, 1\}^*$ , then  $\mathcal{O}(C)$  takes at most  $p(t)$  time steps.
- *Virtual Blackbox:* For all PPT adversaries  $A$ , there exists a PPT simulator  $\mathcal{S}$  and a negligible function  $\text{negl}$  such that for all  $\kappa \in \mathbb{N}$  and all  $C \in \mathcal{C}_\kappa$ , we have

$$\left| \Pr [A(1^\kappa, \mathcal{O}(C)) = 1] - \Pr [\mathcal{S}^C(1^\kappa) = 1] \right| \leq \text{negl}(\kappa)$$

where the probabilities are taken over the randomness of  $A$  and  $\text{simu}$  (and  $\mathcal{O}$  and  $C$  if they are randomized).

**Definition 2 (Indistinguishability Obfuscation ( $i\mathcal{O}$ )).** A uniform machine  $i\mathcal{O}$  is called an indistinguishability obfuscator for a circuit class  $\{\mathcal{C}_\kappa\}$  if the following conditions are satisfied:

- *Correctness:* For all security parameters  $\kappa \in \mathbb{N}$ , for all  $C \in \mathcal{C}_\kappa$ , and for all inputs  $x \in \{0, 1\}^*$ , we have that

$$\Pr [C'(x) = C(x) : C' \leftarrow i\mathcal{O}(1^\kappa, C)] = 1$$

- *Indistinguishability:* For any (not necessarily uniform) PPT distinguisher  $D$ , there exists a negligible function  $\text{negl}$  such that the following hold: For all security parameters  $\kappa \in \mathbb{N}$ , for all pair of circuits  $C_0, C_1 \in \mathcal{C}_\kappa$ , we have that if  $C_0(x) = C_1(x)$  for all inputs  $x$ , then

$$\left| \Pr [D(i\mathcal{O}(1^\kappa, C_0)) = 1] - \Pr [D(i\mathcal{O}(1^\kappa, C_1)) = 1] \right| \leq \text{negl}(\kappa)$$

As noted in [13], if  $\text{negl}(\kappa)$  is bounded by  $2^{-\kappa^\epsilon}$ , for some constant  $\epsilon > 0$ , then we say that  $i\mathcal{O}$  subexponentially indistinguishable.

### 3.2 Injective Noninteractive Commitments

The work in [13] defines and constructs an injective noninteractive bit commitments scheme. For completeness, we present their definition and the construction in what follows.

An injective noninteractive bit commitment scheme is a pair of polynomials  $p(\cdot)$  and  $q(\cdot)$ , and an ensemble of injective functions  $\text{Commit}_\kappa : \{0, 1\} \times \{0, 1\}^{p(\kappa)} \rightarrow \{0, 1\}^{q(\kappa)}$ , where  $\kappa$  is the security parameter, such that for all PPT adversaries  $A$  we have:

$$\Pr \left[ A(1^\kappa, \text{Commit}_\kappa(b; r)) = b \left| \begin{array}{l} b \leftarrow \{0, 1\} \\ r \leftarrow \{0, 1\}^{p(\kappa)} \end{array} \right. \right] \leq \frac{1}{2} + \text{negl}(\kappa)$$

If  $\text{negl}(\kappa)$  can be replaced by  $2^{-\kappa^\epsilon}$ , for some constant  $\epsilon > 0$ , then the scheme is subexponentially secure.

**Construction.** Given an injective one way function  $f : \{0, 1\}^{p'(\kappa)} \rightarrow \{0, 1\}^{q'(\kappa)}$ , with  $h$  as the hard-core bit of  $f$ , one can construct an injective noninteractive commitment scheme as follows. Define  $p(\kappa) = p'(\kappa)$ ,  $q(\kappa) = q'(\kappa) + 1$ , and  $\text{Commit}_\kappa(b; r) = f(r) \parallel b \oplus h(r)$ .

## 4 The Consumable Token Functionality

We utilize the protein-based data storage to build what we call consumable tokens. A consumable token is a physical token that stores some secret messages, requires a secret key to retrieve any of these messages, and (partially) destructs after each data retrieval attempt. An honest recipient will have one data retrieval attempt, while an adversary (who could be more powerful than honest parties) may have multiple attempts. In this section, we define an ideal functionality for consumable tokens that we use in our applications.

**Notation.** We use  $[n]$  as a shorthand for  $\{1, 2, \dots, n\}$ . For time unit representation, we use the term “computational time step” to refer to the time needed to perform an operation in Turing machine-based

modeling of computations. While we use “technologically-realizable time step” to refer to the time needed to perform an operation in physical procedures, which may involve computational algorithms as well. We use  $\kappa$  to denote the security parameter which encapsulates two security parameters:  $\kappa_p$  for physical procedures and  $\kappa_c$  for computational algorithms. Thus, when we say polynomial in  $\kappa$ , this means polynomial in the  $\max\{\kappa_p, \kappa_c\}$ . Lastly, boldface letters represent vectors and PPT is a shorthand for probabilistic polynomial time.

#### 4.1 Ideal Functionality Definition

In formalizing our ideal functionality, we target an adversary class that interacts with a token only using the feasible procedure of applying token keys. Also, we adopt a deterministic approach for quantifying the closeness relation between the keys, and hence, computing the soundness error of any data retrieval attempt. In particular, each key  $k$  in the token key space has a set of close keys. Hitting any of these keys may allow retrieving the message from the token with a probability bounded by  $\gamma$  (the upper bound for the soundness error).

**Adversary class  $\mathcal{A}$ .** We require the consumable token (or any cryptographic application built using this token) to be secure against an adversary that performs data retrieval (or decode) queries using token keys. This adversary, if given multiple tokens, operate on these tokens separately. To capture the fact that class  $\mathcal{A}$  may have more power than the honest parties, an adversary  $A \in \mathcal{A}$  can perform up to  $n$  decode queries instead of only one. This adversary is adaptive in the sense that it may choose her input based on the outputs obtained from previous interactions. Furthermore, this adversary is capable of performing digital and physical procedures.

**Key affinity database.** In order to capture the relation between the keys in the token key space  $\mathcal{K}$ , we use an affinity database  $D$ . Such a database is composed of rows each of which is indexed by a key  $k \in \mathcal{K}$ . Each row, in turn, contains a set of tuples  $(k', \gamma')$  where  $k'$  is a close key to  $k$  and  $\gamma'$  is the corresponding soundness error, such that  $\gamma' \leq \gamma$ . So for a token storing message  $m$  under key  $k$ , a decode query with  $k'$  allows an adversary  $A$  to obtain  $m$  with probability  $\gamma'$ . Recall that a token can store multiple messages each of which is tied to a different key. When these keys are selected at random, any key applied by the adversary will be close to at most one of these keys. Accordingly, in our model the ideal functionality is parameterized by the affinity database  $D$ . It consults this database for each adversarial query to decide key closeness and  $\gamma'$  value (if any). Furthermore, recall that for any token the soundness error is upper bounded by  $\gamma$ . Thus, for all queries  $i \in [n]$ , we require  $\sum_i \gamma'_i \leq \gamma$ .

**Ideal functionality.** An ideal functionality for consumable tokens, denoted as  $\mathcal{F}_{CT}$ , is defined in Figure 3. As shown,  $\mathcal{F}_{CT}$  is parameterized by a security parameter  $\kappa$ , a key affinity database  $D$ , and an integer  $n$ . As noted earlier, for simplicity  $\mathcal{F}_{CT}$  allows an honest party to perform one decode query, while it allows the adversary to perform up to  $n$  queries. It is straightforward to generalize to arbitrary configurations given that the power gap between honest parties and the adversary is preserved.

As shown in the figure,  $\mathcal{F}_{CT}$  supports four interfaces. The first one, **Encode**, allows the sender  $P_1$  to create a consumable token with ID  $\text{tid}$  encoding multiple secret messages under secret keys, all chosen by  $P_1$ , and transfer the token to  $P_2$ . To capture the fact that in real life an adversary may interrupt the communication between  $P_1$  and  $P_2$ ,  $\mathcal{F}_{CT}$  asks the adversary whether to proceed. If the adversary agrees to continue,  $\mathcal{F}_{CT}$  notifies  $P_2$  about the new token, and creates a state for this token.<sup>9</sup> This state includes a counter  $j$  to track the number of decode queries performed so far, which is initialized to 0. It also includes two flags,  $\text{hflag}_1$  and  $\text{hflag}_2$ , tracking whether  $P_1$  and  $P_2$ , respectively, are honest or corrupted. These flags are set by default to 1 indicating that both parties are honest.

The second interface, **Decode**, allows  $P_2$  to query the token on a key  $k'$ . If the input key matches the  $i^{\text{th}}$  token key in  $\mathbf{k}$ , the corresponding message  $\mathbf{m}_i$  will be returned to  $P_2$ , otherwise,  $\perp$  will be returned. After the first query, where the counter  $j$  is set to 1,  $\mathcal{F}_{CT}$  stops answering all future **Decode** queries, capturing that an honest recipient gets only one retrieval query.

The third and fourth interfaces, **Corrupt-encode** and **Corrupt-decode**, are used to notify  $\mathcal{F}_{CT}$  that the environment wants to corrupt any of the involved parties. Corrupting  $P_1$  allows the adversary to encode a vector of messages  $\mathbf{m}'$  under a key vector  $\mathbf{k}'$ , both of his choice. The state of this token will indicate

<sup>9</sup>It is the responsibility of  $P_1$  to securely share  $\mathbf{k}$  with  $P_2$ .

### Functionality $\mathcal{F}_{CT}$

$\mathcal{F}_{CT}$  is parameterized by a security parameter  $\kappa$ , a key affinity database  $D$  and a positive integer  $n$ .

**Encode:** Upon receiving the command (Encode, tid,  $P_1, P_2, \mathbf{k}, \mathbf{m}, v$ ) from token creator  $P_1$ , where tid is the token ID,  $P_2$  is the token recipient,  $\mathbf{k}$  is a vector of  $v$  token keys, and  $\mathbf{m}$  is a vector of  $v$  messages, do: if a token with ID tid was created, end activation. Otherwise, do the following:

- Send (Encode, tid,  $P_1, P_2$ ) to the adversary.
- Upon receiving (OK) from the adversary, send (Encode, tid,  $P_1$ ) to  $P_2$ , and store (tid,  $P_1, P_2, \mathbf{k}, \mathbf{m}, v, j = 0, \text{hflag}_1 = 1, \text{hflag}_2 = 1$ ).

**Decode:** Upon receiving the command (Decode, tid,  $k'$ ) from  $P_2$ , if no token with ID tid exists, then end activation. Otherwise, retrieve (tid,  $P_1, P_2, \mathbf{k}, \mathbf{m}, v, j, \text{hflag}_1, \text{hflag}_2$ ) and do the following:

- If  $j > 0$ , end activation. Else, increment  $j$ , and if  $\exists i \in [v]$  s.t.  $k' = \mathbf{k}_i$ , then set  $\text{out} = \mathbf{m}_i$ , else set  $\text{out} = \perp$ .
- Send (tid, out) to  $P_2$ .

**Corrupt-encode:** Upon receiving the command (Corrupt-encode, tid,  $k', \mathbf{m}', v$ ) from the adversary, do: if a token with ID tid was created, end activation. Else, send (Encode, tid,  $P_1$ ) to  $P_2$  and store (tid,  $P_1, P_2, \mathbf{k}', \mathbf{m}', v, j = 0, \text{hflag}_1 = 0, \text{hflag}_2 = 1$ ).

**Corrupt-decode:** Upon receiving the command (Corrupt-decode, tid,  $k'$ ) from the adversary, if no token with ID tid was created, end activation. Else, retrieve (tid,  $P_1, P_2, \mathbf{k}, \mathbf{m}, v, j, \text{hflag}_1, \text{hflag}_2$ ). If  $\text{hflag}_2 = 1$  and  $j > 0$ , or  $j > n$ , then end activation, else do the following:

- If  $\exists i \in [v]$  s.t.  $k' = \mathbf{k}_i$ , then set  $\text{out} = \mathbf{m}_i$ , else set  $\text{out} = \perp$  and  $(\text{close}, \gamma', i) = \text{affinity}(D, \mathbf{k}, k')$ . If  $\text{close} = 1$ , choose  $r \stackrel{\$}{\leftarrow} [0, 1]$  and change  $\text{out} = \mathbf{m}_i$  if  $r \leq \gamma'$ .
- Store (tid,  $P_1, P_2, \mathbf{k}, \mathbf{m}, v, j + 1, \text{hflag}_1, \text{hflag}_2 = 0$ ).
- Send (tid, out) to the adversary.

**Fig. 3.** An ideal functionality for consumable tokens.

that  $P_1$  is corrupted by setting  $\text{hflag}_1 = 0$ . On the other hand, and to capture the additional power an adversary  $A \in \mathcal{A}$  has, corrupting  $P_2$  allows the adversary to perform up to  $n$  decode queries. Moreover, trying a key  $k' \neq \mathbf{k}_i$  for  $i \in [v]$ , gives the adversary  $\gamma'$  chance to obtain  $\mathbf{m}_i$  if  $k'$  is close enough to key  $\mathbf{k}_i$ .

To depict these capabilities,  $\mathcal{F}_{CT}$  tracks the number of decode queries performed so far and stops answering when this counter  $j$  reaches its maximum value  $n$ . Key closeness and soundness error are measured by invoking an algorithm called *affinity* that simply searches the database and checks if  $k'$  (the adversary's input) is listed in the close key set of any of token keys in  $\mathbf{k}$ . It outputs a flag *close*, and index  $i$ , and a soundness error value  $\gamma'$ . If  $\text{close} = 1$ , this means that  $k'$  is close to  $\mathbf{k}_i$ , and hence,  $\mathcal{F}_{CT}$  outputs  $\mathbf{m}_i$  with probability  $\gamma'$ .

As shown, we restrict the token to be in the hand of either an honest party or the adversary but not both at the same time. Therefore,  $P_2$  cannot be corrupted after the honest recipient submits a decode query. Before submitting any honest decode query, corrupting  $P_2$  is allowed, and when the environment asks for that, the value of  $\text{hflag}_2$  is set to 0.

## 4.2 A Construction for Consumable Tokens

In this section, we present a construction for consumable tokens, shown in Figure 4. It is based on the biological procedures used in storing and retrieving data using proteins discussed in Section 2. We conjecture that it securely realizes  $\mathcal{F}_{CT}$ .<sup>10</sup> In the full version, we present a mathematical (vector-based) model to abstract the biological procedures. We also show a consumable token construction (using this vector model) and formally prove its security.

<sup>10</sup>This construction is described at a high level; the biological experiments (the subject of our followup paper) will determine parameters such as required protein quantities, MS thresholds, amount of decoy proteins, etc., and falsify our conjecture.

### Protocol 1 (A Physical Construction of Consumable Tokens)

Protocol 1 is parameterized by a security parameter  $\kappa$ , the message space  $\mathcal{M}$ , the header space  $\mathcal{H}$ , and the peptide space  $\mathcal{P}$ .

**Encode<sub>phys</sub>( $\mathbf{h}, \mathbf{m}$ ):** Given a vector of  $v$  messages  $\mathbf{m} \in \mathcal{M}^v$  and a vector of  $v$  headers  $\mathbf{h} \in \mathcal{H}^v$ , do the following:

1. For  $i \in [v]$ , encode each  $\mathbf{m}_i$  as a target protein **peptide- $\mathbf{m}_i$** .
2. For each **peptide- $\mathbf{m}_i$**  and  $\mathbf{h}_i$ , synthesize a protein sequence that concatenates them with an amount that allows retrieving  $\mathbf{m}_i$  only once.
3. Mix the target proteins with a natural mixture of decoy proteins  $d_p$  selected at random from  $\mathcal{P}$ , and produce a protein vial  $S_P$ . Output  $S_P$ .

**Decode<sub>phys</sub>( $h, S_P$ ):** Given a header  $h \in \mathcal{H}$ , and a protein vial  $S_P$ , do the following:

1. Immunoprecipitate  $S_P$  with the mAb that recognizes  $h$  then wash out excess mixture.
2. Cleave the target protein and sequence it using MS. If MS identifies the peptides in this protein, then decode the message  $m$  (which will be one of the messages in  $\mathbf{m}$ ) back into its digital form, and set  $\text{out} = m$ . Otherwise, set  $\text{out} = \perp$ . Output  $\text{out}$ .

Fig. 4. A physical construction of consumable tokens.

## 5 Bounded-query Point Function Obfuscation

In this section, we introduce one of the cryptographic applications of consumable tokens: obfuscating bounded-query point functions with multibit output. We begin with motivating this application, after which we define a notion for bounded-query point function obfuscation, and a construction showing how consumable tokens can be used to realize this functionality.

**Motivation.** Program obfuscation is a powerful cryptographic concept that witnessed a large interest in the past two decades. It hides everything about a program other than what can be learned solely by running this program. A program obfuscator is a compiler that takes as input the original program, or circuit, and produces an unintelligible version that preserves functionality but hides any additional information. Program obfuscation found numerous applications, e.g., [36, 37, 55, 58]. Barak et al. [10] initiated the first rigorous study of program obfuscation laying down several security notions. Among them, we have virtual black box (VBB), which states that all what an adversary can learn from an obfuscated program can be simulated using an oracle access to the original program. The same work showed that this notion cannot be realized for general functionalities, but can be realized for restricted function classes.

Point functions are one of these classes that has been studied thoroughly [12, 20, 22, 55, 65]. A point function outputs 1 at a single target point  $x$ , and 0 at all points  $x' \neq x$ . It is useful for access control applications where providing the correct passcode grants the user an access to the system. An extended version of this function class supports a multibit output, i.e., message  $m$ , instead of a single bit. The obfuscation of this extended class is motivated by the notion of digital lockers [20]: for a message  $m$  encrypted using a low-entropy key, such as a human-generated password, the only way for an adversary to learn anything about  $m$  from its ciphertext is through an exhaustive search over the key space.

A question that arises here is whether one can strengthen this security guarantee to also prevent exhaustive search attacks. In real life access-control applications, this usually takes the form of tracking the number of login attempts and lock the user out when a maximum number is exceeded. However, this cannot be applied to digital lockers; an adversary has a copy of the ciphertext and can decrypt it for as many times as she wishes. Thus, the question becomes more about the possibility of augmenting multibit-output point function obfuscation with a bounded-query (or limited number of decryptions) capability.

We answer this question in the affirmative by instantiating a bounded-query obfuscator for point functions with multibit output using consumable tokens. We achieve that by translating the low entropy point, or password  $p$ , into the high entropy token key space, and setting the multibit output to be the message  $m$  encoded inside the token. The message  $m$  is obtained when the correct password  $p$  is queried, and only up to  $n_q$  queries can be performed ( $n_q \in \mathbb{N}$ ).

### Functionality $\mathcal{F}_{BPO}$

$\mathcal{F}_{BPO}$  is parameterized by a security parameter  $\kappa$ , a class of point functions  $\mathcal{I}_\kappa$ , and a positive integer  $n_q$ .

**Obfuscate:** Upon receiving the command  $(\text{Obfuscate}, P_2, p, m)$  from party  $P_1$  (the obfuscator), where  $P_2$  is the evaluator,  $p$  is a password, and  $m$  is the function output (so  $I_{p,m} \in \mathcal{I}_\kappa$ ), do: if this is not the first activation, then do nothing. Otherwise:

- Send  $(\text{Obfuscate}, P_1, P_2)$  to the adversary.
- Upon receiving (OK) from the adversary, store  $(p, m, j = 0, \text{hflag}_2 = 1)$  and output  $(\text{Obfuscate}, P_1)$  to  $P_2$ .

**Evaluate:** Upon receiving input  $(\text{Evaluate}, p')$  from  $P_2$ : if **Obfuscate** was not invoked yet or  $j > 0$ , then end activation. Otherwise, increment  $j$ , and if  $p = p'$ , then set  $\text{out} = m$ , else set  $\text{out} = \perp$ . Output  $\text{out}$  to  $P_2$ .

**Corrupt-obfuscate:** Upon receiving the command  $(\text{Corrupt-obfuscate}, p', m')$  from the adversary, do: If an **Obfuscate** output was generated, then end activation. Else, store  $(p', m', j = 0, \text{hflag}_1 = 0, \text{hflag}_2 = 1)$  and output  $(\text{Obfuscate}, P_1)$  to  $P_2$ .

**Corrupt-evaluate:** Upon receiving the command  $(\text{Corrupt-evaluate}, p')$  from the adversary, if no stored state exists, end activation. Else, retrieve  $(p, m, j, \text{hflag}_2)$  and do:

- If  $j = n_q$ , or  $\text{hflag}_2 = 1$  and  $j > 0$ , then end activation.
- Else, increment  $j$ , set  $\text{hflag}_2 = 0$ , and if  $p' = p$ , set  $\text{out} = m$ , else set  $\text{out} = \perp$ .
- Output  $\text{out}$  to the adversary.

**Fig. 5.** An ideal functionality for bounded-query point function obfuscation.

## 5.1 Definition

We aim to build an obfuscator for multibit-output point functions with points drawn from a low entropy distribution. For password space  $\mathcal{P}$  and message space  $\mathcal{M}$ , let  $I_{p,m} : \mathcal{P} \rightarrow \mathcal{M} \cup \{\perp\}$  be a point function that outputs  $m$  when queried on  $p$  and  $\perp$  otherwise. Let  $\mathcal{I} = \{I_{p,m} | p \in \mathcal{P}, m \in \mathcal{M}\}$  be the family of these functions. In this section, we define an ideal functionality for bounded-query point function obfuscation that allows one honest query and up to  $n_q$  function evaluations. This functionality, denoted as  $\mathcal{F}_{BPO}$ , is captured in Figure 5.

As shown in the figure,  $\mathcal{F}_{BPO}$  supports four interfaces. The first is **Obfuscate** that allows  $P_1$  to ask for obfuscating any point function  $I_{p,m}$  in the class  $\mathcal{I}$  defined earlier. If the adversary agrees to continue,  $\mathcal{F}_{BPO}$  notifies  $P_2$  about the new obfuscation request and creates a state for it. As shown, this state stores a counter to track the number of evaluate queries performed so far, which is initialized to 0. It also stores two flags,  $\text{hflag}_1$  and  $\text{hflag}_2$  introduced before, tracking whether  $P_1$  and  $P_2$ , respectively, are honest or corrupted. These flags are set by default to 1 indicating that both parties are honest. As noted,  $\mathcal{F}_{BPO}$  allows for one obfuscation request, and hence, several instantiations are needed to create multiple obfuscated functions.

The second interface, **Evaluate**, allows  $P_2$  to request evaluating the obfuscated point function over an input password  $p'$  of her choice. If this input matches the stored password  $p$ , then  $P_2$  obtains  $m$ , and  $\perp$  otherwise.  $\mathcal{F}_{BPO}$  updates the counter  $j$  to be 1, and thus, all future queries will not output anything since an honest  $P_2$  gets only one query.

The third and fourth interfaces, **Corrupt-obfuscate** and **Corrupt-evaluate**, are used to notify  $\mathcal{F}_{BPO}$  that the environment wants to corrupt any of the involved parties. Corrupting  $P_1$  allows the adversary to obfuscate any point function  $I_{p,m} \in \mathcal{I}$  of her choice. The state of this obfuscation will indicate that  $P_1$  is corrupted by setting  $\text{hflag}_1$  to 0. On the other hand, corrupting  $P_2$  allows the adversary to perform up to  $n_q$  evaluate queries over inputs of her choice. The adversary needs to invoke **Corrupt-evaluate** for each input evaluation, where after performing  $n_q$  queries,  $\mathcal{F}_{BPO}$  will stop responding. As shown, an obfuscated function can be in the hand of either an honest party or the adversary, but not both at the same time. In particular, if an honest party performs her single evaluate query, **Corrupt-evaluate** will not do anything.

Beside realizing the above ideal functionality, which captures correctness and security, we require any bounded-query point function obfuscation scheme realizing  $\mathcal{F}_{BPO}$  to satisfy the efficiency property defined below.

**Definition 3 (Efficiency of Bounded-query point function Obfuscation).** *There exists a polynomial  $q$  such that for all  $\kappa \in \mathbb{N}$ , all  $I_{p,m} \in \mathcal{I}_\kappa$ , and all inputs  $p' \in \mathcal{P}$ , if computing  $I_{p,m}(p')$  takes  $t$  computational time steps, then the command  $(\text{Evaluate}, p')$  takes  $q(t, \kappa)$  technologically-realizable time steps.*

## 5.2 Construction

A direct application of  $\mathcal{F}_{CT}$  produces a construction that suffers from two limitations. First, it obfuscates a class of point functions with multibit output that is restricted in its domain; must be in the high-entropy token key space  $\mathcal{K}$ . Second,  $\mathcal{F}_{CT}$  has a non-negligible soundness error bounded by  $\gamma$ , which will violate the security guarantees of  $\mathcal{F}_{BPO}$ . Recall that the goal is to have a construction that permits  $A$  to only perform a bounded query exhaustive search. In other words, the success probability of  $A$  in retrieving  $m$  must be only negligibly larger than the probability of guessing the correct password when performing  $n_q$  queries (e.g.,  $\frac{n_q}{|\mathcal{P}|} + \text{negl}(\kappa)$  when using a uniform password distribution). We now show our construction in stages, where to simplify the discussion, we assume a uniform password distribution in the following paragraphs.<sup>11</sup>

**First attempt.** An initial idea is to use a known soundness amplification technique in which  $m$  is shared among  $u$  tokens, accompanied with a mechanism to map a password  $p \in \mathcal{P}$  to a set of keys  $k_i \in \mathcal{K}$  for  $i \in [u]$ . This mapping can be built as, for example, a set of random oracles  $\pi_1, \dots, \pi_u$  each of which maps any password  $p \in \mathcal{P}$  to a random string of size  $\rho$  for some  $\rho \in \mathbb{N}$ . So we have  $\pi_i : \mathcal{P} \rightarrow \{0, 1\}^\rho$  and we denote the output space of each  $\pi_i$  as  $\mathcal{S}^\rho \subset \{0, 1\}^\rho$  such that  $|\mathcal{S}^\rho| = |\mathcal{P}|$ . Each random string is then used to choose a key at random from  $\mathcal{K}$ . This is modeled by having the token creator  $P_1$  use a public algorithm  $\text{KeyGen}$  that takes a random string as input and returns a token key as an output.

At a high level, with this construction an adversary  $A$  will need to retrieve all shares from all token instances in order to recover  $m$ . Taking the worst case scenario, meaning fixing the soundness error to be the maximum value  $\gamma$ , this multi-instance approach reduces the overall soundness error to  $\gamma^u$ . By setting  $u$  to be large enough, the soundness error becomes negligible. Furthermore, and given that each token instance allows  $n$  attempts to retrieve a share, and that all shares are needed to recover  $m$ ,  $A$  will have  $n_q = n$  attempts to obtain  $m$ .

However, the above analysis is flawed. The adversary  $A$  can perform what we call a *leftover attack* and utilize the relation between the keys of the  $u$  tokens (i.e., mappings of the same password) to gain a better advantage in recovering  $m$ . That is, success with any of the tokens not only reveals the message share stored in that token, but also reveals the keys of the rest of the tokens. In detail,  $A$  operates on the first token and performs up to  $n - 1$  queries (by guessing passwords and mapping them to token keys using  $\pi_1$ ). If any of these queries succeeds in retrieving  $m_1$ , then with probability at least  $1 - \gamma$ ,  $A$  knows that the key (and hence the password guess) used in this query is the correct key  $k_1$  (respectively, the password  $p$ ). Knowing  $p$ , and the public mapping function set  $\{\pi_1, \dots, \pi_u\}$  as well as  $\text{KeyGen}$ , allows  $A$  to derive the rest of the tokens keys and retrieve all shares  $m_2, \dots, m_u$ . On the other hand, if  $A$  does not succeed in retrieving  $m_1$  using the first  $n - 1$  queries, it operates on the second token by repeating the same strategy. In fact,  $A$  here has a better chance to guess the correct password/key since it will exclude all the passwords that did not succeed with the first token. If  $A$  succeeds in retrieving  $m_2$ , and thus  $p$  and  $k_1, k_3, \dots, k_u$  as mentioned previously, then it can go back to the first token and use the last query to retrieve  $m_1$ . If it didn't succeed,  $A$  applies the same strategy to the rest of the tokens with the hope of guessing the correct password.

As noted, although the probability of retrieving all shares without correctly guessing any of the token keys is  $\gamma^u$ ,  $A$  now has  $n_q = un$  queries (instead of  $n$ ) to guess the right password. Based on that, the probability of retrieving  $m$  can be computed as:<sup>12</sup>  $\Pr[m] = \frac{un}{|\mathcal{P}|} + (1 - \frac{un}{|\mathcal{P}|})\gamma^u$ . In other words,  $A$  can retrieve  $m$  by either guessing the password correctly in any of the  $un$  queries, or by being lucky and retrieving all shares from all tokens despite using incorrect keys due to the soundness error. Although,

<sup>11</sup>Later, when proving Theorem 1, we generalize that by replacing  $\frac{n_q}{|\mathcal{P}|}$  with a variable representing the probability of guessing the password using  $n_q$  queries. The value of this variable can be computed based on the underlying password distribution.

<sup>12</sup>For the  $j^{\text{th}}$  token, the size of the password space, after excluding the passwords that were already tried, is  $|\mathcal{P}| - (j - 1)n$ . For simplicity, we let  $|\mathcal{P}| - (j - 1)n \approx |\mathcal{P}|$ .

**Protocol 2 (A bounded-query obfuscation scheme for  $\mathcal{I}$ )**

For a security parameter  $\kappa$ , a number of token instances  $u$ ,  $i \in [u]$ , message  $m \in \mathcal{M}$ , password  $p \in \mathcal{P}$ , and token key space  $\mathcal{K}$ , let  $f_1, \dots, f_u$  be as defined before such that  $f_1 : \mathcal{P} \rightarrow \mathcal{K}$ , and  $f_i : \mathcal{P} \times \{0, 1\}^\kappa \rightarrow \mathcal{K}$  for  $i > 1$ ,  $P_1$  be the obfuscator,  $P_2$  be the evaluator, and  $\mathcal{F}_{CT}$  be the consumable token functionality defined in Section 4. Construct a tuple of algorithms (Obf, Eval) to obfuscate a function in  $\mathcal{I}$  as follows.

**Obf:** on input a function  $I_{p,m} \in \mathcal{I}$ ,  $P_1$  does the following:

1. Use an additive secret sharing scheme to generate random shares  $m_1, \dots, m_u$  such that  $m = \oplus_{i=1}^u m_i$ .
2. Set  $r_0 = \perp$ .
3. For  $i \in [u]$ :
  - (a) Generate a random string  $r_i \leftarrow \{0, 1\}^\kappa$ .
  - (b) Compute  $r'_i = \oplus_{j=0}^{i-1} r_j$ .
  - (c) Generate a token key  $k_i: k_i \leftarrow f_i(p, r'_i)$ .
  - (d) Generate a token  $\text{ct}_i$ , with a unique token ID  $\text{tid}_i$ , encoding  $r_i \parallel m_i$  using  $k_i$  by sending the command (Encode,  $\text{tid}_i, P_1, P_2, k_i, r_i \parallel m_i, 1$ ) to  $\mathcal{F}_{CT}$ .

**Eval:** on input an obfuscated function  $\mathfrak{o} = \{\text{ct}_1, \dots, \text{ct}_u\}$  and point  $p \in \mathcal{P}$ ,  $P_2$  does the following:

1. Set  $r_0 = \perp$ .
2. For  $i \in [u]$ :
  - (a) Compute  $r'_i = \oplus_{j=0}^{i-1} r_j$ .
  - (b) Generate a token key  $k_i: k_i \leftarrow f_i(p, r'_i)$ .
  - (c) Query token  $\text{ct}_i$  using  $k_i$  to retrieve  $r_i \parallel m_i$  by sending the command (Decode,  $\text{tid}_i, k_i$ ) to  $\mathcal{F}_{CT}$ .
3. Compute  $m = \oplus_{i=1}^u m_i$  and output  $m$ .

**Fig. 6.** A construction for a bounded-query obfuscation scheme for  $\mathcal{I}$ .

the second term has been reduced and can be set to negligible by configuring  $u$  properly, the first term increased the advantage of  $A$  way beyond  $\frac{n}{|\mathcal{P}|}$ .

**Our construction.** To address the leftover attack, we introduce a construction that chains the  $u$  tokens together so that in order to operate on the  $j^{\text{th}}$  token,  $A$  would need to retrieve all  $m_i$  for  $i < j$ . Otherwise,  $A$  will have to guess the token key from a large space (larger than  $|\mathcal{P}|$ ). This enables us to amplify the soundness error without increasing the total number of queries  $A$  obtains.

Towards building our construction, we introduce a modified way to map passwords to token keys. In particular, a function set  $f_1, \dots, f_u$  is used to generate token keys  $k_1, \dots, k_u$  such that for  $i \in [u]$  we define  $f_1 : \mathcal{P} \rightarrow \mathcal{K}$  and  $f_i : \mathcal{P} \times \{0, 1\}^\kappa \rightarrow \mathcal{K}$  when  $i > 1$ . We write  $k_i \leftarrow f_i(p, r'_i)$ , where  $r'_i = r_0 \oplus \dots \oplus r_{i-1}$  such that  $r_0 = \perp$  and  $r_i \leftarrow \{0, 1\}^\kappa$  is a random string stored in the  $i^{\text{th}}$  token. Each  $f_i$  first applies the mapping  $\pi_i$  described earlier to  $p$  and then uses the output along with the random string  $r'_i$  (for  $i > 1$ ) to generate a token key. A concrete instantiation of  $f_i$  could be composed of a random oracle that takes  $\pi_i(p) \parallel r'_i$  as input and outputs a random string of size  $\rho$ , then **KeyGen** is invoked for this random string to generate a key  $k_i$  as before.

Note that each  $f_i$ , for  $i > 1$ , may have an input space that is larger than the output space, i.e.,  $|\mathcal{P}|2^\kappa \gg |\mathcal{K}|$ . If this is the case (in particular, if  $2^\kappa \geq |\mathcal{K}|$ ), this function can be instantiated to cover the full space of  $\mathcal{K}$  and be a many-to-one mapping. That is, a password  $p \in \mathcal{P}$  can be mapped to different keys (or to all keys in  $\mathcal{K}$ ) by changing the random string  $r$  used when invoking  $f_i$ . Furthermore, correctly guessing the key  $k$  of any of the tokens (other than the first one) without the random string  $r$ , does not help the adversary in guessing the password  $p$  (the adversary still needs to guess  $r$  in order to recover the password).

Protocol 2, described in Figure 6, outlines a construction that uses the above function set, along with the consumable token ideal functionality  $\mathcal{F}_{CT}$ , to build a bounded-query obfuscator for low-entropy point functions with multibit output.

We informally argue that this construction addresses the leftover attack described previously (again, for simplicity we assume a uniform password distribution for the moment). To see this, let an adversary  $A$  follow the same strategy as before and assume that  $A$  did not obtain  $r_1 \parallel m_1$  while performing  $(n - 1)$  queries over the first token.  $A$  now moves to the second token, performs  $(n - 1)$  queries where it will succeed

in guessing the key  $k_2$  correctly with probability  $\frac{n-1}{|\mathcal{K}|}$ . This is different from the naive construction in which this probability is  $\frac{n-1}{|\mathcal{P}|-(n-1)}$  since the previously tried passwords are excluded. In our construction,  $A$ , when it does not have  $r_1$ , has the only choice of trying keys from the full key space  $\mathcal{K}$  (regardless of the password space distribution). This is due to the fact that without  $r'_2$  (where  $r'_2 = r_1$ ),  $A$  cannot compute the induced key space by  $\mathcal{P}$ , thus the only choice is to guess keys from  $\mathcal{K}$ . This probability will be negligible for a large enough  $\mathcal{K}$ .

Furthermore, even if  $A$  guesses the correct  $k_2$ , without the random string  $r'_2$  it will be infeasible to deduce the password  $p$  from  $k_2$  through  $f_2$ .  $A$  needs to feed  $f_2$  with passwords and random strings, where the latter has a space of size  $2^\kappa$ . Also, under the many-to-one construction of  $f_2, \dots, f_u$ , several (or even all) passwords could be mapped to  $k_2$  due to the random string combination, which makes the task harder for  $A$  to find out the correct password. The same argument applies to the rest of the tokens because without  $r_1$ , none of the subsequent  $r'_i$  can be computed, and the only effective strategy for  $A$  is to guess keys from the key space  $\mathcal{K}$ . So for each of these tokens, the success probability is  $\frac{n-1}{|\mathcal{K}|}$  instead of  $\frac{n-1}{|\mathcal{P}|-(i-1)(n-1)}$  as in the naive scheme (again, the latter will depend on the password distribution, but the former will always be uniform). The success probability for  $A$  to retrieve  $m$  is then approximated as:  $\Pr[m] \approx \frac{n}{|\mathcal{P}|} + \left(1 - \frac{n}{|\mathcal{P}|}\right)\gamma^u$ . That is, to retrieve  $m$ ,  $A$  either has to guess the password correctly using the first token, or get lucky with every token and retrieve the share it stores. As shown, this amplifies the soundness error (and can be set to negligible with sufficiently large  $u$ ) without increasing the number of queries  $A$  can do.<sup>13</sup>

### 5.3 Security

Theorem 1 shows that Protocol 2 in Figure 6 securely realizes  $\mathcal{F}_{BPO}$  for the function family  $\mathcal{I}$ , with an arbitrary password distribution. For simplicity, we assume that the token keys  $k_i$ , the randomness  $r_i$ , and the message  $m$  are all of an equal size, which is polynomial in the security parameter  $\kappa$ . The proof can be found in Section C.1.

**Theorem 1.** *For  $0 \leq \gamma \leq 1$ , if each of  $f_1, \dots, f_u$  is as defined above, then Protocol 2 securely realizes  $\mathcal{F}_{BPO}$  for the point function family  $\mathcal{I} = \{I_{p,m} | p \in \mathcal{P}, m \in \mathcal{M}\}$  in the  $\mathcal{F}_{CT}$ -hybrid model in the presence of any adversary  $A \in \mathcal{A}$ , with  $n_q = n$  and large enough  $u$ .*

*Remark 1.* As mentioned before,  $\kappa$  encapsulates a digital and a biological security parameters. Also,  $\mathcal{A}$  is capable of doing computational algorithms and physical procedures, so is the simulator. In the above theorem, the simulator is computational, but it relies on  $\mathcal{F}_{CT}$  whose simulator involves physical procedures. The use of UC security allows us to obtain an overall security guarantee against all physical/digital combined attacks, both in concrete and asymptotic terms.

## 6 (1, n)-time Programs

In this section, we introduce another cryptographic application of consumable tokens;  $(1, n)$ -programs. For such programs, completeness states that an honest party can run a program at most once, while soundness states that an adversary can run this program at most  $n$  times. Again, this can be generalized to allow for multiple honest queries given that the power gap between honest parties and the adversary is preserved. We begin with motivating this application, after which we present a construction showing how consumable tokens can be used to build  $(1, n)$ -programs for arbitrary functions.

*Remark 2.* One may argue that this application is a generalization of the bounded-query point function obfuscation. Thus, the previous section is not needed as one may construct a  $(1, n)$ -program for any point function. However,  $(1, n)$ -program guarantees that only some program was encapsulated, while the previous section guarantees that a valid point function has been encapsulated. Also, the construction shown in this section relies on a rather strong assumption, namely, indistinguishability obfuscation, that was not required in the previous section. Therefore, we present these applications separately.

<sup>13</sup>Similarly, to make the presentation easier, the probability is simplified here where some terms are omitted. See the full proof in Section C.1.

**Motivation.** One-time (and  $k$ -time) programs allow hiding a program and limiting the number of executions to only one (or  $k$ ). They can be used to protect proprietary software and to support temporary transfer of cryptographic abilities. Furthermore,  $k$ -time programs allow obfuscating learnable functions—functions that can be learned using a polynomial number of queries. By having  $k$  as a small constant, an adversary might not be able to learn the function, which makes obfuscating such a function meaningful.

Goldwasser et al. [42] showed a construction for one-time programs that combines garbled circuits with one-time memory devices. Goyal et al. [44] strengthened this result by employing stateful hardware tokens to support unconditional security against malicious recipients and senders. Bellare et al. [11] presented a compiler to compile any program into an adaptively secure one-time version. All these schemes assumed the existence of tamper-proof hardware tokens without any concrete instantiation. Dziembowski et al. [29] replaced one-time memory devices with one-time PRFs. Although they mentioned that no hardware tokens are needed, they impose physical restrictions such as inability to leak all bits of the PRF key, and limiting the number of read/write operations; it is unclear if these assumptions can be realized in practice. Goyal et al. [43] avoided the usage of hardware tokens by relying on a blockchain and witness encryption. In particular, the garbled circuit is posted on the blockchain and the input labels are encrypted using witness encryption, which can be decrypted later after mining several blocks given that the input is unique to guarantee at most one execution. Yet, requiring to store a garbled circuit on a blockchain is impractical.

We investigate the applicability of consumable tokens in constructing bounded execution programs. This is a natural direction given the bounded query capability of these tokens, and the fact that we build these tokens rather than assuming their existence. Nonetheless, the gap between an honest party and the adversary forces us to consider a slightly different notion; the  $(1, n)$ -program mentioned above. Thus, any application that requires the adversary to execute only on one input, like digital currencies, cannot be implemented using  $(1, n)$ -programs. However, applications that allow  $n$  adversarial queries, such as obfuscating learnable functions, can employ our scheme.

## 6.1 Definition

In this section, we define an ideal functionality for bounded-query encapsulation. This functionality, denoted as  $\mathcal{F}_{BE}$ , is captured in Figure 7. The description of the interfaces, and the goal of using the flags and the counter, are very similar to what was described in the previous section for  $\mathcal{F}_{BPO}$ . The only difference is that instead of hiding a point function,  $\mathcal{F}_{BE}$  hides an arbitrary circuit. The honest recipient can evaluate this circuit over one input, while an adversary can evaluate over up to  $n_q$  inputs. Thus, we do not repeat that here.

Beside realizing the above ideal functionality, we require any bounded-query obfuscation scheme realizing  $\mathcal{F}_{BE}$  to satisfy the efficiency property defined below.

**Definition 4 (Efficiency of Bounded-query Encapsulation).** *There exists a polynomial  $p$  such that for all  $\kappa \in \mathbb{N}$ , all  $C \in \mathcal{C}_\kappa$ , and all inputs  $x \in \{0, 1\}^*$ , if computing  $C(x)$  takes  $t$  computational time steps, then the command  $(\text{Evaluate}, \cdot, x)$  takes  $p(t, \kappa)$  technologically-realizable time steps.*

## 6.2 Construction and Security

To ease exposition, we describe our construction in an incremental way. We start with a simplified construction that handles only programs with small input space, and assumes idealized obfuscation (specifically, Virtual Black Box obfuscation [10]). Next we extend to handle programs with exponential-size domains (namely, poly-size inputs). We then replace VBB with indistinguishability obfuscation  $i\mathcal{O}$ . Finally, we briefly discuss how reusable garbling can reduce the use of  $i\mathcal{O}$ .

**First attempt—using VBB.** In this initial attempt, our goal is to lay down the basic idea behind our construction (rather than optimizing for efficiency). We use two tables  $\text{Tab}_1$  and  $\text{Tab}_2$ .  $\text{Tab}_1$  maps a program’s input space  $\mathcal{X}$  to the token message space  $\mathcal{M}$ . This table is secret and will be part of the hidden program. While  $\text{Tab}_2$  maps  $\mathcal{X}$  to the token key space  $\mathcal{K}$ , and it is public.

We use  $\text{Prog}$  to denote the program that encapsulates the intended circuit or simply function  $f$ , which we want to transform into a  $(1, n)$ -program. As shown in Figure 8,  $\text{Prog}$  is parameterized by a table

### Functionality $\mathcal{F}_{BE}$

$\mathcal{F}_{BE}$  is parameterized by a security parameter  $\kappa$ , a circuit class  $\mathcal{C}_\kappa$ , and a positive integer  $n_q$ .

**Encapsulate:** Upon receiving the command  $(\text{Encapsulate}, P_2, C)$  from party  $P_1$  (the encapsulator), where  $P_2$  is the evaluator, and  $C \in \mathcal{C}_\kappa$ , do: if this is not the first activation, then do nothing. Otherwise:

- Send  $(\text{Encapsulate}, P_1, P_2)$  to the adversary.
- Upon receiving (OK) from the adversary, store the state  $(C, j = 0, \text{hflag}_1 = 1, \text{hflag}_2 = 1)$ , and output  $(\text{Encapsulate}, P_1)$  to  $P_2$ .

**Evaluate:** Upon receiving input  $(\text{Evaluate}, x)$  from  $P_2$ , where  $x \in \{0, 1\}^*$ : if **Encapsulate** was not invoked yet or  $j > 0$ , then end activation. Otherwise, increment  $j$  and output  $(C(x))$  to  $P_2$ .

**Corrupt-encapsulate:** Upon receiving the command  $(\text{Corrupt-encapsulate}, C')$  from the adversary, do: If an **Encapsulate** output was generated, then end activation. Else, store  $(C', j = 0, \text{hflag}_1 = 0, \text{hflag}_2 = 1)$  and output  $(\text{Encapsulate}, P_1)$  to  $P_2$ .

**Corrupt-evaluate:** Upon receiving the command  $(\text{Corrupt-evaluate}, x')$  from the adversary, if no stored state exists, end activation. Else:

- Retrieve  $(C, j, \text{hflag}_1, \text{hflag}_2)$ .
- If  $\text{hflag}_2 = 1$  and  $j > 0$ , or  $j = n_q$ , then end activation, else increment  $j$ , set  $\text{hflag}_2 = 0$ , and send  $(C(x'))$  to the adversary.

**Fig. 7.** An ideal functionality for bounded-query encapsulation.

### Program $\text{Prog}_{\text{Tab}, sk, f}$

**Input:**  $m$

**Description:**

1. Parse  $m$  as  $m_0 \parallel m_1$ , and set  $y = \text{Decrypt}(sk, m_1)$
2. Check that there exists  $x \in \mathcal{X}$  such that  $\text{Tab}[x] = m_0$ . If this is not the case then output  $\perp$
3. If  $y \neq \phi^{\ell_{out}}$ , then output  $y$ , else, output  $f(x)$

**Fig. 8.** The program  $\text{Prog}_{\text{Tab}, sk, f}$

$\text{Tab} : \mathcal{X} \rightarrow \mathcal{M}$ , a secret key  $sk$ , and  $f$ . It has two paths: a trapdoor path and a regular one. The trapdoor path is activated when a hidden trigger in the input  $m$  is detected. In particular, this input may contain a ciphertext of the program output. On the other hand, if this ciphertext encrypts the special string  $\phi^{\ell_{out}}$ , where  $\phi$  is some unique value outside the range of  $f$  and  $\ell_{out}$  is the length of  $f$ 's output, the regular path is activated. It evaluates  $f$  over  $x \in \mathcal{X}$  that corresponds to the first part of  $m$ .

Protocol 3 defined in Figure 9 shows a construction for  $(1, n)$ -time program for  $\text{Prog}$  using  $\mathcal{F}_{CT}$ . For simplicity, we assume  $|\mathcal{X}| = |\mathcal{M}| = |\mathcal{K}|$ , the keys in  $\mathcal{K}$  are distinct (i.e., do not have any affinity relation), and that  $\mathcal{F}_{CT}$  has a negligible soundness error (we discuss later how to achieve that). Bounded query is achieved via the consumable token; to evaluate over input  $x$ , the obfuscated program  $bP$  requires a corresponding message  $m$  that is stored inside a token. Since the table  $\text{Tab}_2$  is secret hidden inside  $bP$ , the only way for  $P_2$  to obtain a valid  $m$  is through the consumable token. Once the token is consumed, no more evaluations can be performed. An adversary, on the other hand, and using  $\mathcal{F}_{CT}$ , will be able to obtain up to  $n$  messages corresponding to  $n$  program inputs. Thus, this adversary can run  $bP$  at most  $n$  times. See Section C.2 for an (informal) security argument of this construction.

**Our construction—extending program domain and replacing VBB with  $i\mathcal{O}$ .** The concrete construction of a consumable token may impose limitations on the number of keys and messages that can be stored in a single token. Thus, a token may not be able to cover the full domain  $\mathcal{X}$  of the program  $\text{Prog}$ . So if a single token can store a set of message  $M \subset \mathcal{M}$  messages, we have  $|M| < |\mathcal{X}|$ . To address this issue, we modify the previous construction to use multiple tokens along with an error correcting code

**Protocol 3 (A  $(1, n)$ -time program scheme for Prog—First attempt)**

For a security parameter  $\kappa$ , message space  $\mathcal{M}$ , program input space  $\mathcal{X}$ , and token key space  $\mathcal{K}$ , such that  $|\mathcal{X}| = |\mathcal{M}| = |\mathcal{K}|$ , let  $P_1$  be the encapsulator,  $P_2$  be the evaluator,  $\mathcal{F}_{CT}$  as defined in Section 4 but with negligible soundness error, and  $\text{Tab}_1$  and  $\text{Tab}_2$  are mapping tables as defined above. Construct a tuple of algorithms  $(\text{Encap}, \text{Eval})$  for a  $(1, n)$ -time program scheme as follows.

**Encap:** on input an arbitrary function  $f$  with input space  $\mathcal{X}$  and mapping tables  $\text{Tab}_1 : \mathcal{X} \rightarrow \mathcal{M}$  and  $\text{Tab}_2 : \mathcal{X} \rightarrow \mathcal{K}$ ,  $P_1$  does the following:

1. Generate a token  $\text{ct}$ , with a unique token ID  $\text{tid}$ , encoding all messages  $m \in \mathcal{M}$  each using a unique key from  $\mathcal{K}$ . This is done by sending the command  $(\text{Encode}, \text{tid}, \mathcal{K}, \mathcal{M}, |\mathcal{M}|)$  to  $\mathcal{F}_{CT}$ .
2. Generate a random secret key  $sk \in \{0, 1\}^\kappa$ .
3. Send  $\text{ct}$ ,  $\text{Tab}_2$ , and  $bP = \text{VBB}(\text{Prog}_{\text{Tab}_1, sk, f})$  to  $P_2$ , where  $bP$  is an obfuscated version of the program  $\text{Prog}_{\text{Tab}_1, sk, f}$  described in Figure 8.

**Eval:** on input  $(1, n)\text{-Prog} = (\text{ct}, \text{Tab}_2, bP)$  and  $x \in \mathcal{X}$ ,  $P_2$  does the following:

1. Set  $k' = \text{Tab}_2[x]$ .
2. Query token  $\text{ct}$  using  $k'$  by sending the command  $(\text{Decode}, \text{tid}, k')$  to  $\mathcal{F}_{CT}$  and obtain  $m$ .
3. Output  $\text{out} = bP(m)$ .

**Fig. 9.** A construction for a  $(1, n)$ -time program scheme for  $\text{Prog}_{\text{Tab}, sk, f}$ .

$C$ . We map each  $x \in \mathcal{X}$  to a codeword of length  $\omega$ , and we use  $\omega$  tokens to represent the program input. Each symbol in a codeword indicates which key to use with each token. By configuring  $C$  properly, this technique allows us to cover the program domain without impacting the number of program executions that (an honest or a malicious)  $P_2$  can perform.

Concretely, we use a linear error correcting code  $C$  with minimum distance  $\delta$ , meaning that the Hamming distance between any two legal codewords is at least  $\delta$ . We represent each key in the set  $K \subseteq \mathcal{K}$  used in creating a token, where  $|K| = |\mathcal{M}|$ , as a tuple of index and value. So the set  $K$  is ordered lexicographically such that the first key in this ordered set is given index 0, and so on. Hence, a symbol in a codeword is the index of the token key to be used with the corresponding token. Based on this terminology, we work in a field of size  $q = |K|$  with a code alphabet  $\Sigma = \{0, \dots, q-1\}$ .

**Definition 5 (Linear Codes [4]).** Let  $\mathcal{F}_q$  be a finite field. A  $[\omega, d, \delta]_q$  linear code is a linear subspace  $C$  with dimension  $d$  of  $\mathcal{F}_q^\omega$ , such that the minimum distance between any two distinct codewords  $\mathbf{c}, \mathbf{c}' \in C$  is at least  $\delta$ . A generating matrix  $G$  of  $C$  is a  $\omega \times d$ -matrix whose rows generate the subspace  $C$ .

For any  $d \leq \omega \leq q$ , there exist a  $[\omega, d, (\omega - d + 1)]_q$  linear code: the Reed-Solomon code [56], which we use in our construction. Let  $S$  denote the set of strings to be encoded, such that each input  $x \in \mathcal{X}$  is mapped to a unique  $\mathbf{s} \in S$ . Using classic Reed-Solomon, to encode an input  $x$ , we first define its corresponding  $\mathbf{s}$ , and then we multiply  $\mathbf{s}$  by the generating matrix  $G$  to generate a codeword of size  $\omega$ . Using this approach, we can cover a domain size  $|S| = q^{d+1}$ .

Accordingly,  $P_1$  now has to generate  $\omega$  tokens, denoted as  $\text{ct}_0, \dots, \text{ct}_{\omega-1}$ , instead of one. Each of these tokens will include all keys in  $K$ . Each key  $k \in K$  will be tied to a unique message  $m$  such that  $m$  will be retrieved when a decode query using  $k$  is performed over the token. Let the messages stored in the first token be  $m_{0,0}, \dots, m_{0,q-1}$ , and in the second token be  $m_{1,0}, \dots, m_{1,q-1}$ , and so on. We generate these messages using a pseudorandom generator with some random seed  $r$ . In particular, we have  $m_{i,j} = \text{PRG}(r)[i, j]$  for all  $i \in \{0, \dots, \omega-1\}$  and  $j \in \{0, \dots, q-1\}$ ; we picture the output of the PRG as an  $\omega \times q$  matrix of substrings. Hence,  $m_{0,0}$  is the substring stored at row 0 and column 0 in this matrix, which is the first substring of the PRG output, and so on.<sup>14</sup> Thus, to create token  $\text{ct}_0$ ,  $P_1$  will pass  $K$  and  $m_{0,0}, \dots, m_{0,q-1}$  to  $\mathcal{F}_{CT}$ , while for  $\text{ct}_1$  the messages  $m_{1,0}, \dots, m_{1,q-1}$  along with  $K$  will be passed, etc.

So to execute  $\text{Prog}$  over input  $x$ ,  $P_2$  first maps  $x$  to  $\mathbf{s}$ , and then generates the codeword  $\mathbf{c}$  for  $\mathbf{s}$ . After that, she uses the keys with the indices included in  $\mathbf{c}$  to query the corresponding tokens. For example, if

<sup>14</sup>As we will see shortly,  $m_{i,j} = \text{PRG}(r)[i, j] \parallel \phi^{n(|x|+\ell_{out})}$  assuming all  $x \in \mathcal{X}$  are of the same length, but we omit that for now to ease exposure.

Program  $\text{Prog}_{G,n,sk,r,f}$

**Input:**  $m, x$

**Description:**

1. Parse  $m$  as  $m_0 \parallel \dots \parallel m_{\omega-1}$ , and parse each  $m_i$  as  $m_i^0 \parallel m_i^1$
  2. Use  $G$  to compute the codeword  $\mathbf{c}$  that corresponds to  $x$ .
  3. Check that  $m$  corresponds to a valid codeword: Let  $B = \text{PRG}(r)$ , if  $\exists B[i, \mathbf{c}[i]] \neq m_i^0$ , then output  $\perp$ .
  4. Set  $y_i = \text{Decrypt}(sk, m_i^1)$  for all  $i \in \{0, \dots, \omega - 1\}$ .
  5. If  $\exists y_i \neq \phi^{n(|x|+\ell_{out})}$ , then take the first such  $y_i$  and do the following:
    - Parse  $y_i$  as  $y_i^0 \parallel \dots \parallel y_i^{n-1}$ .
    - Parse each  $y_i^j$  as  $y_i^{j,0} \parallel y_i^{j,1}$  (for  $j \in \{0, \dots, n - 1\}$ ).
    - Output  $y_i^{j,1}$  for which  $y_i^{j,0} = x$ .
- Else, output  $f(x)$ .

**Fig. 10.** The program  $\text{Prog}_{G,n,sk,r,f}$  with linear error correcting codes.

$\mathbf{c} = \{5, 9, 15, \dots\}$ , then  $k_5$  is used to query the first token to retrieve  $m_0 = m_{0,5}$ ,  $k_9$  is used to query the second token and retrieve  $m_1 = m_{1,9}$ , etc. These messages  $m = m_0 \parallel \dots \parallel m_{\omega-1}$  will be used as input to  $\text{Prog}$  to obtain the output  $f(x)$ . This in turn means that  $\text{Prog}$  must check that  $m$  corresponds to a valid codeword in  $C$ . We also modify the trapdoor path to allow including multiple outputs instead of one. This is needed to allow the simulator to simulate for an adversary who queries the tokens out of order. It may happen that the last query is common for two (or more) codewords (in other words, just when this query takes place, the simulator will tell that the adversary got valid codewords). Having multiple outputs (each concatenated with the  $x$  value that leads to this output) permits the simulator to embed the valid outputs for the inputs corresponding to these valid codewords.

The modified version of  $\text{Prog}$  can be found in Figure 10 (with both the linear code and  $i\mathcal{O}$  instead of VBB). We also modify the description of  $\text{Prog}$  (see Figure 11). The parameters of the underlying error correcting code are configured in a way that produces a code  $C$  such that  $|C| = |\mathcal{X}|$ . As shown, the output of  $\text{Encap}$  now contains  $\omega$  tokens beside the obfuscation of  $\text{Prog}$ .  $\text{Eval}$  follows the description above.

**On preserving the number of program executions ( $1, n$ ).** An honest party can query any token once. Thus, overall, she will be able to retrieve only one codeword. An adversary, on the other hand, can query each token up to  $n$  times. We want to guarantee that the  $n\omega$  messages she obtains does not allow constructing more than  $n$  valid codewords. In other words, we want to ensure that to retrieve  $n + 1$  codewords, at least  $n\omega + 1$  distinct queries are needed.

To formalize this notion, we define what we call a *cover*; a cover of two, or more, codewords is the set of all distinct queries needed to retrieve these codewords. For example, codewords  $\mathbf{c}_1 = \{5, 4, 13, 17\}$  and  $\mathbf{c}_2 = \{5, 9, 12, 18\}$  have a cover of  $\{5, 4, 9, 12, 13, 17, 18\}$ ,<sup>15</sup> and so  $P_2$  needs 7 queries to obtain the messages that correspond to these codewords from the tokens.

**Definition 6.** A code  $[\omega, d, \delta]_q$  is  $n$ -robust if for any  $n + 1$  distinct codewords the size of the cover is at least  $n\omega + 1$ .

So the robustness factor is the number of codewords an adversary can obtain. To preserve this number to be the original  $n$  that an adversary can obtain with one token, we need to configure the parameters of  $C$  to satisfy the lower bound of the cover size defined above. We show that for Reed-Solomon codes as follows.

**Lemma 1.** For a Reed-Solomon code  $[\omega, d, \delta]_q$  to be  $n$ -robust (cf. Definition 6), we must have  $\omega - n(d - 1) - 1 \geq 0$ .

*Proof.* Recall that  $\delta$  is the minimum distance between any two codewords in the code  $C$ . Thus, worst case scenario (which is the best for an adversary) is to have  $n + 1$  codewords that differ from each other

<sup>15</sup>Note that if 5 was not on the same position for both codewords then it would have been considered distinct. Different positions means that  $k_5$  will be used with different tokens, which leads to different messages  $m_{i,j}$ .

**Protocol 4 (A  $(1, n)$ -time program scheme for Prog)**

For a security parameter  $\kappa$ , message space  $\mathcal{M}$ , input space  $\mathcal{X}$ , and token key space  $\mathcal{K}$ , let  $P_1$  be the encapsulator,  $P_2$  be the evaluator,  $\mathcal{F}_{CT}$  be as defined in Section 4 but with negligible soundness error,  $[\omega, d, \delta]_q$  be a linear code  $C$  with a generating matrix  $G$  such that  $|C| = |\mathcal{X}|$ , and  $PRG : \{0, 1\}^\kappa \rightarrow \{0, 1\}^{\omega q |m|}$  be a pseudorandom generator, where  $m \in \mathcal{M}$  and  $|K| = q$  for  $K \subseteq \mathcal{K}$ . Construct a tuple of algorithms  $(\text{Encap}, \text{Eval})$  for a  $(1, n)$ -time program scheme as follows.

**Encap:** On input an arbitrary function  $f$  with input space  $\mathcal{X}$ , and a linear code  $[\omega, d, \delta]_q$  with generating matrix  $G$ ,  $P_1$  does the following:

1. Generate secret key  $sk \in \{0, 1\}^\kappa$  and a string  $r \in \{0, 1\}^\kappa$  both at random.
2. Generate messages  $m_{i,j} = PRG(r)[i, j] \parallel \phi^{n(|x|+\ell_{out})}$  for all  $i \in \{0, \dots, \omega - 1\}$  and  $j \in \{0, \dots, q - 1\}$ .
3. Generate at random token key subspace  $K \subseteq \mathcal{K}$  such that  $|K| = q$ .
4. For  $i \in \{0, \dots, \omega - 1\}$ , generate a token  $ct_i$ , with a unique token ID  $tid_i$ , encoding messages  $m_{i,0}, \dots, m_{i,q-1}$  using  $k_0 \dots k_{q-1} \in K$ . This is done by sending the command  $(\text{Encode}, tid_i, \{k_0 \dots k_{q-1}\}, \{m_{i,0}, \dots, m_{i,q-1}\}, q)$  to  $\mathcal{F}_{CT}$ .
5. Send  $ct = \{ct_0, \dots, ct_{\omega-1}\}$  and  $bP = i\mathcal{O}(\text{Prog}_{G,n,sk,r,f})$  to  $P_2$ , where  $\text{Prog}_{G,n,sk,r,f}$  is defined in Figure 10.

**Eval:** On input a  $(1, n)$ -Prog =  $(ct, bP)$  and  $x \in \mathcal{X}$ ,  $P_2$  does the following:

1. Map  $x$  to a codeword  $\mathbf{c}$ .
2. For each  $i \in \{0, \dots, \omega - 1\}$ , query token  $ct_i$  using  $k'_{c[i]}$  by sending the command  $(\text{Decode}, tid_i, k'_{c[i]})$  to  $\mathcal{F}_{CT}$  and get  $m_i$  in return.
3. Output  $\text{out} = bP(m_0 \parallel \dots \parallel m_{\omega-1}, x)$ .

**Fig. 11.** A construction for a  $(1, n)$ -time program scheme for  $\text{Prog}_{G,n,sk,r,f}$ .

in  $\delta$  symbols but the rest are identical. Thus, we have:

$$|\text{cover}| = \omega + n\delta$$

But recall that for Reed-Solomon,  $\delta = \omega - d + 1$ . Substituting this in the equation above produces:

$$|\text{cover}| = n\omega + \omega - n(d - 1)$$

To satisfy the cover lower bound stated in Definition 6, and thus have an  $n$ -robust code, we must have  $\omega - n(d - 1) \geq 1$ , which completes the proof.  $\square$

Accordingly, we have the following theorem (the proof can be found in Section C.3).

**Theorem 2.** *Assuming sup-exponentially secure  $i\mathcal{O}$  and one-way functions, the  $i\mathcal{O}$ -based construction described in Figure 11 is a  $(1, n)$ -time program in the  $\mathcal{F}_{CT}$ -hybrid model.*

*Remark 3.* It is an intriguing question whether we can obtain  $(1, n)$ -time programs without  $i\mathcal{O}$ . Since an adversary can evaluate over multiple inputs, we cannot use garbled circuits—evaluating a circuit over more than one input compromises security. A potential direction is to employ reusable garbling [41], and use our construction to build a  $(1, n)$ -time program for the circuit that encodes the inputs (which requires a secret key from the garbler). Thus,  $i\mathcal{O}$  is only needed for the encoding circuit, and our consumable token limits the number of times this circuit can be evaluated, rather than obfuscating the full program as above.

**Acknowledgements.** This material is based upon work supported by DARPA under contracts #HR001120C00, #HR00112020023, and #D17AP00027. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA. This research was supported in part by a grant from the Columbia-IBM center for Blockchain and Data Transparency, by JPMorgan Chase & Co., and by LexisNexis Risk Solutions. Any views or opinions expressed herein are solely those of the authors listed.

## References

1. Aaronson, S.: Quantum copy-protection and quantum money. In: 2009 24th Annual IEEE Conference on Computational Complexity. pp. 229–242. IEEE (2009)
2. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. *science* **266**(5187), 1021–1024 (1994)
3. Adleman, L.M.: Computing with dna. *Scientific american* **279**(2), 54–61 (1998)
4. Almashaqbeh, G., Benhamouda, F., Han, S., Jaroslawicz, D., Malkin, T., Nicita, A., Rabin, T., Shah, A., Tromer, E.: Gage mpc: Bypassing residual function leakage for non-interactive mpc. *PETS* **2021**(4), 528–548 (2021)
5. Ananth, P., Placa, R.L.L.: Secure software leasing. In: *EUROCRYPT* (2021)
6. Angel, T.E., Aryal, U.K., Hengel, S.M., Baker, E.S., Kelly, R.T., Robinson, E.W., Smith, R.D.: Mass spectrometry-based proteomics: existing capabilities and future directions. *Chemical Society reviews* **41**(10), 3912–28 (may 2012)
7. Armknecht, F., Maes, R., Sadeghi, A.R., Sunar, B., Tuyls, P.: Memory leakage-resilient encryption based on physically unclonable functions. In: *Towards Hardware-Intrinsic Security*, pp. 135–164. Springer (2010)
8. Badrinarayanan, S., Jain, A., Ostrovsky, R., Visconti, I.: Uc-secure multiparty computation from one-way functions using stateless tokens. In: *ASIACRYPT* (2019)
9. Baldwin, M.A.: Protein identification by mass spectrometry issues to be considered. *Molecular & Cellular Proteomics* **3**(1), 1–9 (2004)
10. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im) possibility of obfuscating programs. In: *CRYPTO*. pp. 1–18 (2001)
11. Bellare, M., Hoang, V.T., Rogaway, P.: Adaptively secure garbling with applications to one-time programs and secure outsourcing. In: *ASIACRYPT*. pp. 134–153. Springer (2012)
12. Bitansky, N., Canetti, R.: On strong simulation and composable point obfuscation. In: *CRYPTO*. pp. 520–537. Springer (2010)
13. Bitansky, N., Canetti, R., Garg, S., Holmgren, J., Jain, A., Lin, H., Pass, R., Telang, S., Vaikuntanathan, V.: Indistinguishability obfuscation for ram programs and succinct randomized encodings. *SIAM Journal on Computing* **47**(3), 1123–1210 (2018)
14. Blawat, M., Gaedke, K., Huetter, I., Chen, X.M., Turczyk, B., Inverso, S., Pruitt, B.W., Church, G.M.: Forward error correction for dna data storage. *Procedia Computer Science* **80**, 1011–1022 (2016)
15. Bornholt, J., Lopez, R., Carmean, D.M., Ceze, L., Seelig, G., Strauss, K.: A dna-based archival storage system. *ACM SIGOPS Operating Systems Review* **50**(2), 637–649 (2016)
16. Broadbent, A., Gutoski, G., Stebila, D.: Quantum one-time programs. In: *Annual Cryptology Conference*. pp. 344–360. Springer (2013)
17. Brzuska, C., Fischlin, M., Schröder, H., Katzenbeisser, S.: Physically uncloneable functions in the universal composition framework. In: *CRYPTO*. pp. 51–70 (2011)
18. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: *FOCS*. pp. 136–145. IEEE (2001)
19. Canetti, R.: Universally composable security. *J. ACM* **67**(5), 28:1–28:94 (2020)
20. Canetti, R., Dakdouk, R.R.: Obfuscating point functions with multibit output. In: *EUROCRYPT*. pp. 489–508. Springer (2008)
21. Canetti, R., Halevi, S., Katz, J., Lindell, Y., MacKenzie, P.: Universally composable password-based key exchange. In: *EUROCRYPT*. pp. 404–421. Springer (2005)
22. Canetti, R., Kalai, Y.T., Varia, M., Wichs, D.: On symmetric encryption and point obfuscation. In: *TCC*. pp. 52–71. Springer (2010)
23. Chandran, N., Goyal, V., Sahai, A.: New constructions for uc secure computation using tamper-proof hardware. In: *EUROCRYPT*. pp. 545–562 (2008)
24. Church, G.M., Gao, Y., Kosuri, S.: Next-generation digital information storage in dna. *Science* p. 1226355 (2012)
25. Crick, F.H.: On protein synthesis. In: *Symp Soc Exp Biol*. vol. 12, p. 8 (1958)
26. Damgård, I., Kilian, J., Salvail, L.: On the (im) possibility of basing oblivious transfer and bit commitment on weakened security assumptions. In: *EUROCRYPT*. pp. 56–73. Springer (1999)
27. Damgård, I., Scafuro, A.: Unconditionally secure and universally composable commitments from physical assumptions. In: *ASIACRYPT* (2013)
28. Döttling, N., Kraschewski, D., Müller-Quade, J.: Unconditional and composable security using a single stateful tamper-proof hardware token. In: *TCC* (2011)
29. Dziembowski, S., Kazana, T., Wichs, D.: One-time computable self-erasing functions. In: *TCC*. pp. 125–143. Springer (2011)
30. Eichhorn, I., Koeberl, P., van der Leest, V.: Logically reconfigurable pufs: Memory-based secure key storage. In: *Proceedings of the sixth ACM workshop on Scalable trusted computing*. pp. 59–64 (2011)

31. El Orche, F.E., Hollenstein, M., Houdaigoui, S., Naccache, D., Pchelina, D., Roenne, P.B., Ryan, P.Y., Weibel, J., Weil, R.: Taphonomical security:(dna) information with foreseeable lifespan. *Cryptology ePrint Archive* (2021)
32. Erlich, Y., Zielinski, D.: Dna fountain enables a robust and efficient storage architecture. *Science* **355**(6328), 950–954 (2017)
33. Feist, P., Hummon, A.B.: Proteomic challenges: sample preparation techniques for microgram-quantity protein analysis from biological samples. *International journal of molecular sciences* **16**(2), 3537–63 (feb 2015)
34. Fisch, B., Freund, D., Naor, M.: Physical zero-knowledge proofs of physical properties. In: *CRYPTO*. pp. 313–336. Springer (2014)
35. Fisch, B.A., Freund, D., Naor, M.: Secure physical computation using disposable circuits. In: *TCC*. pp. 182–198. Springer (2015)
36. Garg, S., Gentry, C., Halevi, S., Raykova, M.: Two-round secure mpc from indistinguishability obfuscation. In: *TCC*. pp. 74–94. Springer (2014)
37. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: *FOCS*. pp. 40–49. IEEE (2013)
38. Glaser, A., Barak, B., Goldston, R.J.: A zero-knowledge protocol for nuclear warhead verification. *Nature* **510**(7506), 497–502 (2014)
39. Glish, G.L., Vachet, R.W.: The basics of mass spectrometry in the twenty-first century. *Nature Reviews Drug Discovery* **2**(2), 140–150 (2003)
40. Goldman, N., Bertone, P., Chen, S., Dessimoz, C., LeProust, E.M., Sipos, B., Birney, E.: Towards practical, high-capacity, low-maintenance information storage in synthesized dna. *Nature* **494**(7435), 77 (2013)
41. Goldwasser, S., Kalai, Y., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: *ACM STOC* (2013)
42. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: One-time programs. In: *CRYPTO*. pp. 39–56. Springer (2008)
43. Goyal, R., Goyal, V.: Overcoming cryptographic impossibility results using blockchains. In: *TCC*. pp. 529–561. Springer (2017)
44. Goyal, V., Ishai, Y., Sahai, A., Venkatesan, R., Wadia, A.: Founding cryptography on tamper-proof hardware tokens. In: *TCC*. pp. 308–326 (2010)
45. Grass, R.N., Heckel, R., Puddu, M., Paunescu, D., Stark, W.J.: Robust chemical preservation of digital information on dna in silica with error-correcting codes. *Angewandte Chemie International Edition* **54**(8), 2552–2555 (2015)
46. Hazay, C., Lindell, Y.: Constructions of truly practical secure protocols using standardsmartcards. In: *ACM CCS*. pp. 491–500 (2008)
47. Hazay, C., Polychroniadou, A., Venkatasubramanian, M.: Composable security in the tamper-proof hardware model under minimal complexity. In: *TCC* (2016)
48. Hazay, C., Polychroniadou, A., Venkatasubramanian, M.: Constant round adaptively secure protocols in the tamper-proof hardware model. In: *PKC*. pp. 428–460 (2017)
49. Hervey IV, W.J., Strader, M.B., Hurst, G.B.: Comparison of digestion protocols for microgram quantities of enriched protein samples. *Journal of Proteome Research* **6**(8), 3054–3061 (2007)
50. Jain, A., Lin, H., Sahai, A.: Indistinguishability obfuscation from well-founded assumptions. In: *ACM STOC*. pp. 60–73. ACM (2021)
51. Jin, C., Xu, X., Burleson, W.P., Rührmair, U., van Dijk, M.: Playpuf: Programmable logically erasable pufs for forward and backward secure key management. *IACR Cryptol. ePrint Arch.* **2015**, 1052 (2015)
52. Katz, J.: Universally composable multi-party computation using tamper-proof hardware. In: *EUROCRYPT*. vol. 7, pp. 115–128. Springer (2007)
53. Landenmark, H.K., Forgan, D.H., Cockell, C.S.: An estimate of the total dna in the biosphere. *PLoS biology* **13**(6), e1002168 (2015)
54. Lindell, Y.: Anonymous authentication. *Journal of Privacy and Confidentiality* **2**(2) (2011)
55. Lynn, B., Prabhakaran, M., Sahai, A.: Positive results and techniques for obfuscation. In: *EUROCRYPT*. pp. 20–39. Springer (2004)
56. MacWilliams, F.J., Sloane, N.J.A.: *The theory of error correcting codes*, vol. 16. Elsevier (1977)
57. Moran, T., Naor, M.: Basing cryptographic protocols on tamper-evident seals. *TCC* **411**(10), 1283–1310 (2010)
58. Naccache, D., Shamir, A., Stern, J.P.: How to copyright a function? In: *PKC*. pp. 188–196. Springer (1999)
59. Ostrovsky, R., Scafuro, A., Visconti, I., Wadia, A.: Universally composable secure computation with (malicious) physically uncloneable functions. In: *EUROCRYPT*. pp. 702–718. Springer (2013)
60. Pappu, R., Recht, B., Taylor, J., Gershenfeld, N.: Physical one-way functions. *Science* **297**(5589), 2026–2030 (2002)
61. Roehsner, M.C., Kettlewell, J.A., Batalhão, T.B., Fitzsimons, J.F., Walther, P.: Quantum advantage for probabilistic one-time programs. *Nature communications* **9**(1), 1–8 (2018)
62. Rührmair, U.: Oblivious transfer based on physical unclonable functions. In: *International Conference on Trust and Trustworthy Computing*. pp. 430–440 (2010)

63. Savaryn, J.P., Toby, T.K., Kelleher, N.L.: A researcher’s guide to mass spectrometry-based proteomics. *PROTEOMICS* **16**(18), 2435–2443 (sep 2016)
64. Stein, S.: Mass Spectral Reference Libraries: An Ever-Expanding Resource for Chemical Identification. *Analytical Chemistry* **84**(17), 7274–7282 (2012). <https://doi.org/10.1021/ac301205z>
65. Wee, H.: On obfuscating point functions. In: *ACM STOC*. pp. 523–532 (2005)
66. Yao, A.C.C.: How to generate and exchange secrets. In: *FOCS*. pp. 162–167 (1986)
67. Zhang, Y., Fu, L.H.B.: Research on dna cryptography. In: *Applied cryptography and network security*. vol. 357, pp. 10–5772. InTech: Rijeka, Croatia (2012)

## A Unclonable Polymer-based Data Storage

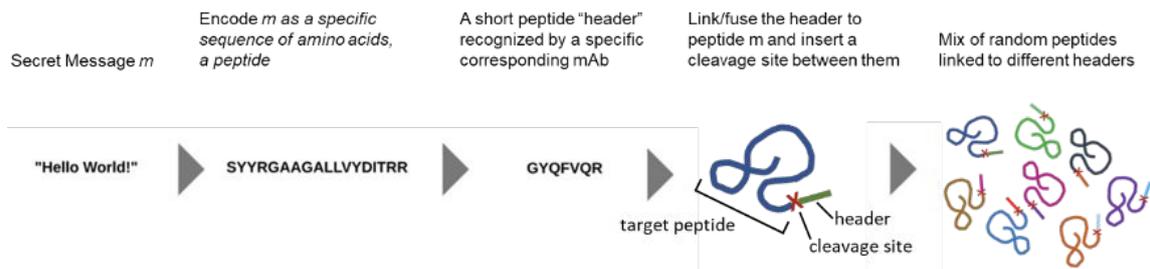
Biological polymers are large molecules made of recurring building blocks, nucleotides and amino acids for DNA/RNA and protein, respectively, and play important roles in all living organisms. For example, we pass on genetic information to our children by our genes that are made of DNA, a long polymer built of four types of nucleotides. Our cells transcribe the DNA into RNA copies, which are short-living polymers that can be thought of as a set of digital instructions. Finally, RNA is translated into proteins, which make virtually all of the molecular machines in our body; from enzymes that metabolize food to the strong fibers that comprise our hair. In over 3.5 billion years of evolution, nature optimized the chemical structure of these biopolymers to reliably carry information. Our biosphere holds about 1037 bytes of data in the DNA of organisms [53]. This amount of data is 15 orders of magnitude bigger than all human-made digital data.

Advances in biotechnology have allowed the custom-tailored synthesis of biological polymers for the purpose of data storage. Much of the effort in this new field has focused on the use of DNA, generating an arsenal of molecular protocols to store and retrieve information [14, 15, 24, 32, 40, 45]. Possible advantages of such DNA-based data storage machinery include: high storage density, where a single milligram of DNA can carry a terabyte of data; reliability and robustness as DNA samples can be amplified to replicate the data they store; and the high durability of the DNA material [45]. The data storage protocols combine digital algorithms and physical procedures that encode the digital data into DNA strings, and then synthesize these strings to produce the DNA material. Data retrieval proceeds in the reversed direction. That is, commodity devices are used to “read” the DNA molecules as digital strings of the four letters, then these strings are decoded to obtain the digital data.

With the growing application using biological polymers for storage, we became interested in the cryptographic attributes this new hardware offers. Specifically, in this paper we propose using protein as a data storage material. Curiously, the most fundamental characteristics of proteins; the fact that they cannot be directly cloned nor can they replicate or be amplified and that “data retrieval” is typically self-destructive, might be considered as limitations from a regular data storage point of view. However, these exact traits of protein can confer powerful features to instantiate cryptographic primitives and protocols. In what follows, we describe a protein-based scheme for data storage and discuss its practical specifications. We rely on this scheme in realizing the cryptographic primitives we build, and we use its specifications in justifying correctness and security of these constructions.

### A.1 A Physical Scheme for Protein-based Data Storage

Proteins are large polymers, hundreds of amino acids long, that fold onto themselves to create stable geometric objects presenting surfaces and regions of distinct physio-chemical features such as: rigidity or flexibility as well as electrical charge, polarity or hydrophobicity. The collective effect of these traits and their 3D orientation in space result in a precise spatial conformation which is critical for fulfilment of a given protein’s function. Short amino-acid polymers (< 50 amino-acids), i.e., oligomers, are referred to as peptides and often are viewed as being too short to have elaborate conformations. This however, is not necessarily true, as is illustrated by some peptide neurotransmitters and hormones such as: enkaphalin, angiotensin II, angiotensin I, substance P, alpha-endorphin and beta-endorphin that are comprised of 5, 8, 10, 11, 16, and 31 amino acids, respectively. One must keep in mind that for the purpose of the proposed use of proteins or peptides for data storage, a functional 3D conformation is not required nor does it need to be maintained. For storage, the digital message is encoded into the primary configuration of the peptide/protein, i.e., the sequence of the 20 natural amino acids of the protein material, the



**Fig. 12. General scheme for peptide-based data storage.** A message ( $m$ ) is assigned to a specific peptide sequence. A short peptide header which is specifically recognized by a unique monoclonal antibody (mAb) is linked to the peptide message via a specific protease cleavage site. The message is then mixed with a vast collection of decoy peptides, each with a different peptide sequence and linked to different headers recognized by different mAbs. The desired specific peptide can be affinity selected using its correct corresponding mAb. The peptide can be cleaved and the sequence of peptide- $m$  can be determined, and thus, enable the message to be decoded.

"peptide-payload". To retrieve the message, the order of the amino-acids of a protein is determined, after which this sequence is decoded to reconstruct the original message.

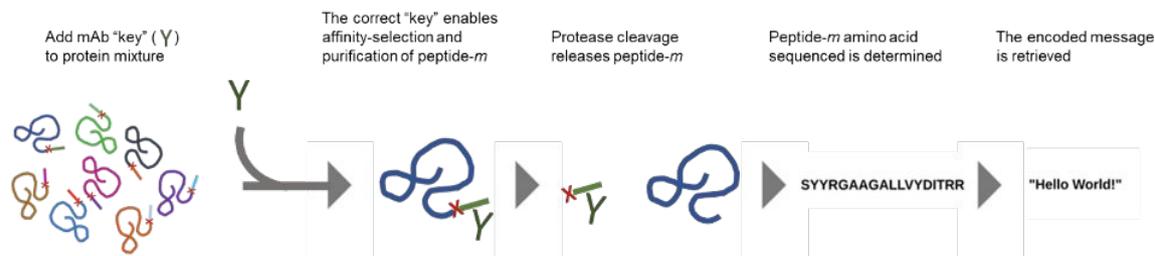
Given that our primary goal is to design a biological machinery to realize cryptographic primitives securely, we extend this basic paradigm to support data secrecy. Our proposal is based on a number of features of proteins and peptides: (i) unique peptides can be designed to comprise any string of amino acids and be physically produced with precision and at high fidelity, (ii) a peptide sample whose amino acid sequence is not known is unclonable and cannot be replicated or amplified, (iii) sequencing the peptide results in its consumption.

As illustrated in Figure 12, the peptide message, peptide- $m$ , is conjugated to a short (< 10 amino acids) peptide tag, a tag that is recognized specifically by a predetermined monoclonal antibody (mAb). Thus, the peptide tag, designated "header", corresponds to its specific mAb. Next, the peptide message, peptide- $m$ , is mixed with a vast variety of decoy peptide messages, all of which are peptide permutations of composition and length, each conjugated to a collection of alternative "header" sequences. All attempts to learn the sequence of peptide- $m$  by sequencing the mixture of peptide- $m$  along with the decoy peptides are destined to fail and will generate an incomprehensible set of sequence data for which the desired peptide message cannot be decoded. The only possible way to decode the message is to first single out and purify the data-containing peptide. This can be achieved employing the unique mAb that specifically recognizes the unique header attached to peptide- $m$ . Application of the mAb allows affinity selection of the correct peptide otherwise mixed with an overwhelming amount of decoy peptides. Without knowing which of a large collection of mAbs is to be used to affinity purify peptide- $m$ , one cannot retrieve the stored message.

We outline these processes as two protocols for data storage and retrieval, which are depicted pictorially in Figures 12 and 13, respectively. Data storage proceeds as follows:<sup>16</sup>

1. Encode the message as an unordered sequence of amino acids (peptide- $m$ ) which can be determined by protein sequencing methodologies.
2. Secretly choose a "header" peptide to be linked to peptide- $m$ . A typical length for such a peptide could be 5 to 15 amino acids and is specifically and exclusively recognized by a unique mAb.
3. Produce a single protein sequence, which concatenates the peptide- $m$  and the header peptide spaced by an amino acid motif which enables specific targeted protease cleavage, e.g., coagulation factor Xa which specifically cleaves carboxy terminus to the arginine residue in the tetra-peptide sequence IEGR or TEV (tobacco etch virus) protease which cleaves carboxy terminus to the glutamine (Q) residue of the 7 amino acid sequence ENLYFQG/S.
4. Mix the protein sequence containing peptide- $m$  and its header with a mixture of decoy peptides where each of these decoys is concatenated with a header peptide distinct and different from the secret header. The amount of peptide- $m$  will be as dictated by the data retrieval protocol to allow

<sup>16</sup>Although we talk about one message in these protocols, several messages can be stored in one vial by having several peptide- $m$ s instead of one, each of which is conjugated with a unique header and mixed with the decoys.



**Fig. 13. Message retrieval.** A vast mixture of proteins (a “vial”) containing the desired peptide-*m* linked to a header (known to the sender and the receiver) that corresponds to the secret mAb is received. The mAb is added to the mixture eliciting peptide-*m* affinity selection and purification. Protease cleavage releases peptide-*m* which can then be sequenced. Decoding the sequence reveals the stored (secret) message.

one data retrieval attempt, with limited extra amount to account for losses of material during the immunoprecipitation process.

The sender shares the secret header with the recipient (so this is digital data, i.e., the peptide sequence of the header, which reveals to the recipient the identity of the correct unique mAb to be used to recover peptide-*m*). Then he sends a vial of the protein mix (a physical component comprised of peptide-*m* mixed with a vast collection of decoy peptides). It should be noted that all decoy peptides and the target peptide-*m* are of the same general length, mass, and composition, differing in sequence. Thus, effective purification of the desired message from the decoys, without the matching mAb, is impossible through standard biochemical/biophysical methodologies.

Figure 13 illustrates the data retrieval protocol, which is the reversed process depicted in Figure 12. Basically, pre-knowledge of the correct mAb and its implementation leads to revealing the peptide-*m* sequence and the decoding of the message sent to the recipient.

## A.2 “Cloning” Proteins is a misnomer

The popular claim that a specific protein has been “cloned” is, in actuality, misleading. Proteins, in marked contrast to nucleic acid polymers (DNA and RNA), are not directly cloneable. Thus, when popular statements refer to a given “cloned protein”, in fact the correct meaning of this statement is that the DNA corresponding to the protein has been cloned, not the protein itself. This is a fundamental distinction between proteins and nucleic acids. The latter are replicated by a relatively simple process using a single strand template against which a new complementary strand is generated, nucleotide base for base through base-pairing (e.g., A to T and G to C). Proteins, on the other hand, are naturally produced via an extremely complex multi-component process in which an RNA message is read, tri-nucleotide codon after codon, by ribosomes which recruit amino acid charged t-RNA molecules to enzymatically link amino acid residues, one by one via the ribosome large subunit. To date there is no method, natural or artificial, to systematically replicate or amplify proteins or even short peptides directly. Furthermore, all production of specific proteins requires absolute pre-knowledge of the amino acid sequence being generated. Thus, storing data by protein ensures that the amount and availability of the message can be limited strictly, reliably and determined and regulated by the sender.

## A.3 Sequencing Proteins

Determining protein sequences historically has required long and tedious chemical processing entailing cleavage of proteins into shorter peptides, fragment purification and then, through processes such as Edman degradation, revealing the N-terminal residues through cyclic chemical reactions, one by one until a complete sequence is assembled. Thus, for example, the complete sequence of the bacterial enzyme beta-galactosidase (> 1000 residues) took 9 years and numerous publications (e.g., Zabin and Fowler 1970-1978). A revolution in the ability to determine protein sequences occurred when their genes (DNA) could be cloned and subjected to nucleic acid sequencing. In reality, therefore, almost all protein sequences known to date were artificially determined first by translating their cDNA (complementary DNA) equivalents.

A major breakthrough in direct reading of proteins has been by the implementation of mass spectroscopy (MS) or versions thereof [9, 40]. This strategy relies on physical fragmenting the proteins into a mixture of short peptides and generating a histogram of their observed molecular weights. Then, the spikes of this histogram are matched with weights of known peptides (amino acid combinations) registered in a database [64]. This process is inherently destructive due to fragmentation. It is further resistant to natural scaling attempts, because an increase in the length of the peptides means exponential increase in their combinatorial variety. This in turn makes the constant range of available weight spectra exponentially more dense and harder to be identified.

The MS analyses are praised for their precision and sensitivity. This however must be clarified, as the exceptional performance of MS is very much dependent on the successful assignment of an “unknown” protein to pre-known and deciphered proteins in a database for comparison. Sequencing proteins *de novo*, i.e., from first principles and not through comparisons with known baseline data is possible, however is restricted to shorter peptides and with some limitations regarding their amino acid compositions. That being said, our research has revealed various parameters and conditions required for effective *de novo* sequencing, thus, allowing us to assert that revelation of the sequence of **peptide-m** is feasible, provided that **peptide-m** can be relatively purified.

**Peptides as molecular machines for cryptographic applications.** We have established the fundamental conditions required for using peptides as physical payloads in cryptographic applications. The following are terms and conditions that need to be met for effective use of peptides as cryptographic payloads and have been determined through experimentation:

- composition of the peptide (use of select and preferred amino acids).
- peptide length (within the capacity for reliable MS *de novo* sequencing).
- association with discrete peptide headers that can be exploited for highly precise and selective message purification by defined keys (i.e., mAbs that bind their corresponding headers with exquisite specificity).
- a modular platform to produce vast collections of effective decoy payloads.

In order to read a protein sequence successfully, MS needs a biological material at the recommended amounts, i.e., above a minimum threshold of sample volume and protein concentration. This concentration allows MS to identify the peptides in the mixture through standard protocols of *de novo* sequencing. However, an order of magnitude dilution or sampling from the mixture is considered guaranteed to fail. MS protocols have been continuously challenged by limited availability of samples, and have been optimized to facilitate detection under such conditions [6, 63]. Enriched microscale approaches allow handling of samples at low-microgram and high-nanogram quantities [33, 49]. State of the art noise models are based on Poisson and multinomial distributions of the discrete ion signal [18], quantifying the expected increase in error rates upon any attempt for further reduction of the number of molecules in the experiment. All these methodologies confirm that going below the required material thresholds cause failures in sequencing proteins, and hence, prevent retrieving the stored data successfully. Moreover, the protocols referred to in these studies generally rely on an existing database of protein sequences for comparison of MS spectra. The proposed application demands sequencing *de novo* of unknown random sequences not regularly present in the existing databases. In fact, the **peptide-m** sequence can be dictated a priori to be exclusive and not existent in the MS protein database.

#### A.4 Adversarial Interaction with a Peptide-based Data Storage

We assume that the adversary is knowledgeable and surpasses the sender’s technological capacity and is knowledgeable of the existence of the complete repertoire of mAbs (or the full key space) available to the sender. The condition that ensures security is that the identity of the bona fide correct Key is absolutely and unforgivingly requires its use first and only once. Any error is unacceptable as a false attempt promises consumption of the vial and its content and thus loss of any chance to retrieve the message. The problem thus becomes a matter of reducing the chance of guessing the correct key always and repeatedly to totally impractical.

The promise of success in retrieving the secret message stored in a vial is by purifying it using the matching mAb, and then sequence the purified mix using MS. Other techniques such as attempts

to purify the protein without the use of any mAb or diluting the sample and repeated iterations of purification and sequencing, are not viable due to the following reasons. Purification strategies, such as high-performance liquid chromatography (HPLC) or gel electrophoresis, are usually based on a combination of separation methods that exploit the physiochemical characteristics of peptides, including size, charge and hydrophobicity. These strategies can only separate out relatively large fractions of the mixture, as opposed to immunoprecipitation that can pull down proteins at less than 10<sup>6</sup> concentration. Furthermore, these separation assays are rendered useless by incorporating decoy proteins most of which are chemically and physically similar to the target **peptide-m** in terms of their polarity, gel retardation, and other basic properties. In addition, partitioning the protein mix in a vial without the use of antibodies with the hope of isolating the target **peptide-m** is not effective. Any partitioning of the sample may push the amount of **peptide-m** below MS detectability threshold.

Consequently, for an adversary who seizes the protein vial, the only way to reveal anything about the secret message is by applying the pre-determined correct mAb and then sequence the purified sample. This adversary may perform a brute search attack, meaning that he guesses the secret header, apply a matching mAb to purify the sample and sequence it.

In our model, we consider an adversary that could be more powerful than the honest recipient. That is, he may use a more advanced MS; one that operates at lower quantity and purity thresholds than the honest recipient's MS. This permits this powerful adversary to perform more data retrieval attempts than what the honest recipient gets. In particular, we have that an honest recipient will perform one data retrieval attempt using the correct mAb, while an adversary could be capable of performing up to  $n$  attempts, for a small integer  $n$ . Moreover, this adversary could be an adaptive one, meaning that he may choose his next header guess based on the outcome of previous trials.

## A.5 Bounded Query

In theory, an adversary with endless resources and capability should ultimately be able retrieve  $m$ . Obviously, not on the first simple attempt, that would only be possible if the adversary was exceptionally lucky, or if the identity of the correct key was known to him, which we assume is not. The latter is the only real advantage the honest user has over an adversary, the pre-set knowledge of the correct mAb to use. Diluting the contents of the vial, or sub-dividing it so to be able to test multiple rounds, multiple attempts each time with a different key will give the adversary the chance to eventually discover and apply the correct key. However, two conflicting parameters must be considered:

- (i) the technical capability of the adversary to successfully assay ever more sub-divisions of the vial with different keys with each attempt.
- (ii) the number of available keys being so great as to make (i) impractical.

Thus, one must structure and design the vial to ensure that the power in theory of the adversary in (i) is overcome practically, by a reasonable number of keys determined in (ii). These conditions can be met by limiting the volume/amount of  $m$  in the vial and banking on the fact that with each incorrect attempt of the adversary the absolute amount of  $m$  is gradually consumed to the point that despite all technical prowess of the adversary, the concentration/amount of  $m$  is rendered way below the required threshold of sensitivity making the measure of  $m$  impossible.

Keys in the current model are mAbs which are famous for exquisite specificity to bind and react with their corresponding header. Thus, in theory, each attempt of the adversary using an incorrect key should leave the bona fide  $m$  untouched. Practically, there are always non-specific losses in every attempt to pull-down the correct  $m$  when using the wrong key, however our model does not rely on the marginal losses due to technical processing of the sample. Rather, all decoy peptides are associated with a collection of different headers recognized by different mAbs or keys. In addition, the bona fide **peptide-m** displays the correct header, plus a collection of the alternative headers of the decoys. The correct header is associated with the genuine **peptide-m** exclusively, however, **peptide-m** shares multiple additional headers of the decoys. Thus, every attempt to pull-down the correct **peptide-m** using an incorrect key will pull down the **peptide-m** along with a diversity of decoys. Due to the similarity of composition and size of the decoys and the **peptide-m**, MS sequencing generates a heap of incomprehensible fragments so complex making deciphering of  $m$  impossible. Also, due to the destructive sequencing using MC, the vial will be consumed after a limited number of attempts, establishing what we call the bounded query feature of peptide-based data storage.

## A.6 Experimental Errors

Two classes of errors can complicate the model we propose: the first is what we call completeness errors. That is, despite proficiency of the honest user and the pre-knowledge of the correct key, processing the vial is technically faulty and the sequence of `peptide-m` is not generated. The corrective of this error is to simply send multiple vials to be processed multiple times to eventually produce a robust and coherent sequence. The second type of error is that the adversary succeeds in retrieving information about the secret message  $m$  despite using an incorrect mAb. This could be due to the use of some (incorrect) mAb that could be of physical features close to the correct mAb. Coupled with any inaccuracies in the biological procedures, this may lead to retrieving  $m$  with some (non-negligible) probability. We amplify the soundness error as part of our cryptographic constructions as will be shown later.

## B Vector-based Model and Construction of Consumable Tokens

This section introduces a mathematical abstraction of protein-based data storage. Such an abstraction is needed to formally prove how the biological construction realizes our formal notion (i.e., ideal functionality) of consumable tokens presented in Section 4.1.

### B.1 Model

Our mathematical abstraction involve modeling the secret headers, the protein vial encoding the secret message(s), and all procedures needed to store and retrieve data from a protein vial. We use vectors of protein quantities to capture protein vials, hence the name vector-based model, and so all biological procedures are pictured as vector operations.

**Token payloads.** A token can be used to store a vector of  $v$  secret messages  $\mathbf{m}$  for an arbitrary  $v \in \mathbb{N}^+$ ,<sup>17</sup> such that each  $\mathbf{m}_i \in \mathcal{M}$  and  $\mathcal{M}$  is the space of all messages. Inside a token, the  $i^{th}$  message (or payload) will be encoded and synthesized as a protein, which we call a target protein `protein-mi`. Some applications may require storing only one payload per token, while others may require a large number of payloads to be stored.

It was discussed in Appendix A that our biology-based construction supports short payloads. To store a long message, we fragment it into multiple payloads and store each one in a different vial (full details can be found in Appendix B.2 as part of our security proof). Thus, storing a message in a token may correspond to a construction of several vials depending on the message length. For simplicity, in the discussion below we assume messages of short length (so a token corresponds to one vial). Even later, and as mentioned before, when we say store a message in a token this implicitly involves the fragmentation procedure (if needed) and creating several vials instead of one.

**Token keys.** A token key is the digital description of the antigen header attached to `protein-mi`. Thus, a token may have a vector of  $v$  keys denoted at  $\mathbf{h}$  such that each  $\mathbf{h}_i \in \mathcal{H}$ , where  $\mathcal{H}$  is the space of all headers.

**Protein-based consumable tokens.** A protein-based consumable token is a vial containing decoy proteins mixed with the target proteins `protein-mi` for  $i \in \{0, \dots, v - 1\}$ . All proteins in a vial are of equal amounts, which is measured in milligrams (mg), and all have the same physical properties.

We model a protein vial as a tuple  $(\mathbf{y}, \mathbf{m}, \mathbf{h})$  consisting of a polymer amount vector  $\mathbf{y}$  that represents the amounts of all proteins in the vial, vector of payloads  $\mathbf{m}$ , and a vector of token keys  $\mathbf{h}$ . The length of  $\mathbf{y}$  is the number of headers used to construct the sample, which is  $|\mathcal{H}|$  in our case.<sup>18</sup> We picture  $\mathbf{y}$  indexed by the headers in  $\mathcal{H}$ , so for an index  $h' \in \mathcal{H}$ , the value  $\mathbf{y}[h']$  is the amount of the protein attached to  $h'$ .

<sup>17</sup>As mentioned before, although  $v$  is an arbitrary integer, it may have an upper bound imposed by the physical construction.

<sup>18</sup>Although the size of this space could be exponential, in this work we do not focus on the computational costs of the model. In practice, the physical procedures will choose a subset of the headers at random when constructing a token. So, an adversary still needs to guess the secret header from the full space  $\mathcal{H}$ . Any additional errors due to this modeling difference will be counted for in the completeness and soundness errors in our model.

**Token creation.** Creating a vial storing  $\mathbf{m}$  is abstracted by setting the component values of the polymer amount vector  $\mathbf{y}$ . These values must allow an honest recipient to retrieve one of the secret messages. In other words, to retrieve  $\mathbf{m}_i$ , pulling down the target protein  $\text{protein-}\mathbf{m}_i$ —using the antibodies that match the secret header  $\mathbf{h}_i$ —must return a protein amount sufficient for the sequencing machinery MS to detect the signal of the target protein and decode  $\mathbf{m}_i$ .

The amount of a protein in a vial depends on the binding strength of the matching antibodies and the thresholds of MS. For the antibodies binding strength we have two parameters:  $\alpha$  for the strength of the specific parts of the antibodies, and  $\alpha'$  for the strength of non-specific parts of these antibodies, where  $\alpha > \alpha'$ . These are multiplicative factors that determine the fraction of a protein amount to pulled down when applying antibodies. For the MS thresholds, we have  $q_{\text{ms}}$  and  $\tau_c$  representing the two conditions that a protein sample must satisfy so MS will be able to detect a protein signal. In particular, a sample for target protein  $\text{protein-}\mathbf{m}_i$  must have at least  $q_{\text{ms}}$  mg of this protein, and second, the relative amount of this protein with respect to all proteins within the sample must be larger than  $\tau_c$ .

The values of these factors may vary based on the quality of the raised antibodies and the actual MS machine. To simplify our model, we fix them to their lower bounds as realized by biology, and thus, all proteins in a vial will have equal amounts.<sup>19</sup>

As mentioned previously, in this work we restrict ourselves to a scheme that allows reading only one message  $\mathbf{m}_i$ . This means that the amount of  $\text{protein-}\mathbf{m}_i$  will drop below the required thresholds after the first pull down. Let  $q$  be the initial amount of any protein in the vial. The conditions above can be translated as follows.

A pull down must return a protein amount above the threshold  $q_{\text{ms}}$  (where  $\alpha q$  is the pulled down amount of the target protein when using the matching antibodies). So:

$$q \geq \frac{q_{\text{ms}}}{\alpha} \quad (1)$$

A pull down must yield a pure enough protein sample. Note that  $(|\mathcal{H}| - 1)\alpha'q$  is the amount pulled from the proteins in a vial, other than the target protein, using the non-specific parts of the antibodies matching the header attached to the target protein. Thus, by computing the relative amount of the target protein and the rest of the proteins in the pulled-down sample we obtain:

$$\frac{\alpha}{\alpha + (|\mathcal{H}| - 1)\alpha'} \geq \tau_c \quad (2)$$

The condition above controls the choice of the parameters, and we assume it holds for our physical construction and model.

The `GenToken` routine is used to create a token. It is parameterized by  $\mathcal{H}$ ,  $\alpha$ , and  $q_{\text{ms}}$ , and takes as inputs the token keys  $\mathbf{h}$  and the secret messages  $\mathbf{m}$ .

```

GenToken( $\mathcal{H}, \alpha, q_{\text{ms}}$ )( $\mathbf{h}, \mathbf{m}$ ) {
   $\forall h' \in \mathcal{H}$  do:
     $\mathbf{y}[h'] \leftarrow \frac{q_{\text{ms}}}{\alpha}$ 
  return ( $\mathbf{y}, \mathbf{h}, \mathbf{m}$ )
}

```

**Pulling-down proteins.** Retrieving a message  $\mathbf{m}_i$  stored in a vial starts by pulling down the target protein  $\text{protein-}\mathbf{m}_i$  using the antibodies matching the header  $\mathbf{h}_i$ . Since knowing the header implies knowing the antibodies, we abstract the pull-down procedure as applying a header to the token  $(\mathbf{y}, \mathbf{m}, \mathbf{h})$ . This procedure produces a pulled-down tuple  $(\mathbf{z}, \mathbf{m}, \mathbf{h})$  and a new state for the original token  $(\mathbf{y}', \mathbf{m}, \mathbf{h})$ , such that  $\|\mathbf{z}\|_1 \leq \|\mathbf{y}\|_1$  and  $\mathbf{y}' = \mathbf{y} - \mathbf{z}$ .

The protein amounts in  $\mathbf{z}$  are controlled by the applied header  $h'$ . That is, the protein in  $\mathbf{y}$  with a header that matches  $h'$  will have the largest amount in  $\mathbf{z}$ , since this will be recognized by the specific parts of the antibodies. The rest of the proteins, on the other hand, will have much smaller amounts, since their headers will be recognized by the non-specific parts of these antibodies.

<sup>19</sup>Quantifying the exact values of these factors or their lower bounds is a hard task in practice. As part of our ongoing work on the biology construction paper, we will estimate these factors based on the biological experiments conducted in that work.

As discussed in Appendix A, some headers could be (physically) very close to  $\mathbf{h}_i$ , and thus, applying any of these headers may allow retrieving  $\mathbf{m}_i$ . This falls under what we call soundness error, where due to this closeness relation, an adversary who applies a header  $h' \neq \mathbf{h}_i$  may obtain  $\mathbf{m}_i$  with probability at maximum  $\gamma$  ( $\gamma$  is an upper bound for the soundness error) based on how close  $h'$  is to  $\mathbf{h}_i$ . We model this closeness relation using a *header affinity database*  $D$  indexed with all headers in  $\mathcal{H}$ . The row for header  $h$  contains all its close headers, each of which is tied to some soundness error value  $\gamma' < \gamma$  where the larger  $\gamma'$  the closer the header. The interface `affinity` can be used to determine if two headers are close and the value of  $\gamma'$ . Based on that, for a header  $h'$  that close to  $\mathbf{h}_i$ , the target protein `protein- $\mathbf{m}_i$`  will be pulled down with factor  $\alpha$  (instead of  $\alpha'$ ) with probability  $\gamma'$ .

The above is captured using a `PullDown` routine parameterized by  $\mathcal{H}$ ,  $\alpha$ ,  $\alpha'$ ,  $D$ , and security parameter  $\kappa$ .

```

PullDown( $\mathcal{H}, \alpha, \alpha', D, \kappa$ )( $h', (\mathbf{y}, \mathbf{m}, \mathbf{h})$ ) {
   $\forall h \in \mathcal{H}$  do:
    (close,  $\gamma'$ ) = affinity( $h, h', D$ )
    if  $h = h'$ :
       $\mathbf{y}'[h] \leftarrow (1 - \alpha)\mathbf{y}[h]$ 
    Else if close = 1 and  $(r \xleftarrow{\$} \{0, 1\}^\kappa) \leq \gamma'$ :
       $\mathbf{y}'[h] \leftarrow (1 - \alpha)\mathbf{y}[h]$ 
    Else:
       $\mathbf{y}'[h] \leftarrow (1 - \alpha')\mathbf{y}[h]$ 
   $\mathbf{z} = \mathbf{y} - \mathbf{y}'$ 
  return ( $\mathbf{z}, \mathbf{m}, \mathbf{h}$ ), ( $\mathbf{y}', \mathbf{m}, \mathbf{h}$ )
}

```

The routine above makes a simplifying assumption about the behavior of the pull-down procedure. In particular, both  $\mathbf{z}$  and  $\mathbf{y}'$  may contain lower amounts than what we model. This is due to the fact that some quantities could be lost by, e.g., being stuck on the lab equipment. For simplicity, we do not model this additive loss, which makes our model stronger in the sense that an adversary obtains more residual material after each pull down he performs. Also, in practice, the initial amount of proteins will be larger than the lower bound we require in our model to account for such losses.

**Sequencing proteins.** After performing a pull-down, the next step is to sequence  $(\mathbf{z}, \mathbf{m}, \mathbf{h})$  in order to obtain  $\mathbf{m}_i$  that the recipient wants. As mentioned before, sequencing is done by feeding the pulled-down proteins to the sequencing machinery MS. In order to retrieve  $\mathbf{m}_i$ ,  $\mathbf{z}$  must contain a sufficient amount of `protein- $\mathbf{m}_i$`  with high purity. We model the sufficient amount by requiring  $\mathbf{z}[\mathbf{h}_i] \geq q_{\text{ms}}$ , and we model purity by requiring  $\frac{\mathbf{z}[\mathbf{h}_i]}{\|\mathbf{z}\|_1} \geq \tau_c$ .

Recall that an adversary may use an advanced, highly accurate, MS that detects proteins at lower signal thresholds than those for an honest recipient. This makes the amount of the target protein sufficient for more than one retrieval operation. To model this aspect, we introduce adversarial threshold values, denoted as  $q'_{\text{ms}}$  and  $\tau_s$ , such that  $q'_{\text{ms}} < q_{\text{ms}}$  and  $\tau_s < \tau_c$ .

We define a `Sequence` routine to model the behavior of the sequencing machinery. This sequence takes as input the pulled-down tuple  $(\mathbf{z}, \mathbf{m}, \mathbf{h})$ , and returns either a message  $\mathbf{m}_i$  or  $\perp$  depending on whether the MS thresholds are satisfied.

```

Sequence( $q, \tau$ )( $\mathbf{z}, \mathbf{m}, \mathbf{h}$ ) {
  for  $i \in 0, \dots, v - 1$  do:
    if  $\mathbf{z}[\mathbf{h}_i] \geq q$  and  $\frac{\mathbf{z}[\mathbf{h}_i]}{\|\mathbf{z}\|_1} \geq \tau$ 
      return  $\mathbf{m}_i$ 
  return  $\perp$ 
}

```

Recall that the sequence operation destroys all proteins in  $\mathbf{z}$ . This makes  $(\mathbf{y}, \mathbf{m}, \mathbf{h})$  sufficient for a limited number of data retrieval queries (one for the honest party and  $n$  for the adversary). We formally analyze that as part of the security proof of the vector-based construction of consumable tokens that we introduce next.

**Protocol 5 (A Vector-based Construction of Consumable Tokens)**

Protocol 5 is parameterized by a security parameter  $\kappa$ , the header space  $\mathcal{H}$ , the message space  $\mathcal{M}$ , the parameters  $(q_{ms}, \tau_c, \alpha, \alpha')$  defined above, and the header affinity database  $D$ .

$\text{Encode}_{\text{vec}}(\mathbf{h}, \mathbf{m}, v)$ : Given a vector of  $v$  messages  $\mathbf{m} \in \mathcal{M}^v$  and a vector of  $v$  headers  $\mathbf{h} \in \mathcal{H}^v$ , do the following:

1.  $(\mathbf{y}, \mathbf{m}, \mathbf{h}) \leftarrow \text{GenToken}_{(\mathcal{H}, \alpha, q_{ms})}(\mathbf{h}, \mathbf{m})$
2. Output  $(\mathbf{y}, \mathbf{m}, \mathbf{h})$

$\text{Decode}_{\text{vec}}(h', (\mathbf{y}, \mathbf{m}, \mathbf{h}))$ : Given a header  $h' \in \mathcal{H}$  and a consumable token  $(\mathbf{y}, \mathbf{m}, \mathbf{h})$ , do the following:

1.  $((\mathbf{z}, \mathbf{m}, \mathbf{h}), (\mathbf{y}', \mathbf{m}, \mathbf{h})) \leftarrow \text{PullDown}_{(\mathcal{H}, \alpha, \alpha', D, \kappa)}(h', (\mathbf{y}, \mathbf{m}, \mathbf{h}))$
2.  $\text{out} \leftarrow \text{Sequence}_{(q_{ms}, \tau_c)}(\mathbf{z}, \mathbf{m}, \mathbf{h})$
3. Output out

**Fig. 14.** A vector-based construction of consumable tokens.

**B.2 Vector-Based Construction of Consumable Tokens**

We use our vector-based model to build a construction that realizes the ideal functionality of consumable tokens  $\mathcal{F}_{CT}$ . As mentioned before, this is the same as the physical construction introduced in Section 4.2, but uses the mathematical terms we introduced in the previous section. The construction is described in Figure 14.

Theorem 3 states that Protocol 5 securely realizes  $\mathcal{F}_{CT}$ . The security proof involves proving (1) the bounded query feature, i.e., an adversary can perform a finite number of decode queries  $n$ , (2) that fragmentation allows storing a message  $m$  of any length securely using several vials, and (3) that there is a simulator who can simulate a view for the adversary in the ideal world, by interacting with  $\mathcal{F}_{CT}$ , such that this view is indistinguishable from what the adversary observes in the real world.

**Theorem 3.** For  $n \leq \frac{\log(\alpha q'_{ms} q_{ms}^{-1})}{\log(1-\alpha')}$ , the vector-based construction in Protocol 5 securely realizes  $\mathcal{F}_{CT}$ .

*Proof.* We start with proving that an adversary  $A$  can perform a finite number of decode queries using a consumable token.

**Lemma 2.** For all  $\mathbf{m} \in \mathcal{M}^v$ , all  $\mathbf{h} \in \mathcal{H}^v$ , where  $v \in \mathbb{N}^+$ , all tuples  $(\mathbf{y}, \mathbf{m}, \mathbf{h})$  created by  $\text{GenToken}_{\mathcal{H}, \alpha, q_{ms}}$  with parameters  $\alpha, \alpha', q_{ms}, \tau_c$  as defined before, and an adversary  $A$  with sequencing thresholds  $q'_{ms}$  and  $\tau_s$  defined before, there exists an integer  $n \leq \frac{\log(\alpha q'_{ms} q_{ms}^{-1})}{\log(1-\alpha')}$  such that all  $j^{\text{th}}$  Decode queries, for  $j \geq n + 1$ , that  $A$  makes will output  $\perp$  with probability 1.

*Proof.* This follows by the specifications of the PullDown procedure. Recall that the only way to retrieve any of the stored messages is through a PullDown followed by a Sequence procedure. Each PullDown invocation over  $(\mathbf{y}, \mathbf{m}, \mathbf{h})$  reduces the amount of each protein in  $\mathbf{y}$  by at least a factor of  $\alpha'$ . That is, the  $j^{\text{th}}$  pull-down produces a residual amount of any protein- $m_i$  as  $\mathbf{y}_j[\mathbf{h}_i] \leq (1 - \alpha')\mathbf{y}_{j-1}[\mathbf{h}_i]$ . Given that the initial amount of each protein component in  $\mathbf{y}$  is  $\frac{q_{ms}}{\alpha}$ , this amount drops below  $q'_{ms}$  after  $n$  pull-down operations when  $\frac{q_{ms}}{\alpha}(1 - \alpha')^n < q'_{ms}$ , which produces:

$$n \leq \frac{\log(\alpha q'_{ms} q_{ms}^{-1})}{\log(1 - \alpha')} \tag{3}$$

This applies for any protein component in  $\mathbf{y}$  including the target proteins  $\text{protein-}m_i$  for  $i \in \{0, \dots, v-1\}$ . Given that the pulled-down amount will be destructed when invoking Sequence, and that proteins are uncolonable, any protein component in  $\mathbf{y}$  will be consumed after  $A$  performs (at maximum)  $n$  queries, which completes the proof.  $\square$

Next we prove that for any message  $m \in \mathcal{M}$ , we can construct a consumable token to encode it. If  $m$  is longer than what is allowed by biology, where the allowed length is denoted as  $\ell_{enc}$ , we fragment it and store each fragment in a separate vial.<sup>20</sup>

**Lemma 3.** *For an integer  $\ell_{enc} \in \mathbb{N}^+$  and a given message  $m \in \mathcal{M}$ , there exists a consumable token construction that encodes  $m$  using  $t \in \mathbb{N}^+$  vials.*

*Proof.* We have two cases:  $|m| \leq \ell_{enc}$ , and  $|m| > \ell_{enc}$ . When  $|m| \leq \ell_{enc}$ , this is the straightforward case where one vial is enough to encode  $m$ . When  $|m| > \ell_{enc}$ , we fragment  $m$  into  $m_1 \parallel \dots \parallel m_t$  such that each  $|m_j| = \ell_{enc}$  (it could be the case that  $|m_t| < \ell_{enc}$ ) and  $t = \lceil \frac{|m|}{\ell_{enc}} \rceil$ . Each  $m_j$  will be encoded in a separate vial using a randomly selected header  $h_j$ . Thus, a consumable token in this case is a set of  $t$  vials in which the message  $m$  is encoded under the token key  $h = h_1 \parallel \dots \parallel h_t$ .

The security of this construction relies on the security of the single-vial token instance. That is, for the single-vial token instance, the probability that an adversary  $A$  retrieves a message  $m$  stored in the token is:

$$\Pr[m] \leq \frac{n}{|\mathcal{H}|} + \left(1 - \frac{n}{|\mathcal{H}|}\right)\gamma$$

In other words, either  $A$  gets lucky and guesses the correct header  $h$  using the  $n$  queries he can perform, or he does not guess the right header but tries a close enough one that allows him to obtain  $m$  with probability  $\leq \gamma$ . If  $A$  can retrieve any message fragment with probability better than that, then we can use  $A$  to construct an adversary  $A'$  to attack the single-vial token instance. Thus, this multi-vial token scheme is a secure consumable token for long messages, which completes the proof.  $\square$

Lastly, we describe a simulator  $\mathcal{S}$  that resides in the ideal world and interacting with  $\mathcal{F}_{CT}$ . We show that  $\mathcal{S}$  will generate a view such that the environment cannot distinguish from the real world view. For simplicity, we consider  $A$  as a proxy for the environment and focus on its view.

*The simulator.* We have two cases based on which party is corrupted:

- *Token creator  $P_1$  is corrupted:* This is a simple case.  $\mathcal{S}$  will receive the message and header vectors  $\mathbf{m}$  and  $\mathbf{h}$  (each of which contains  $v$  components) that  $A$  sends when invoking  $\text{Encode}_{vec}$ .  $\mathcal{S}$  generates a token ID  $\text{tid}$  and then sends the command  $(\text{Corrupt-encode}, \text{tid}, P_1, P_2, \mathbf{h}, \mathbf{m}, v)$  to  $\mathcal{F}_{CT}$ , where in  $\mathcal{F}_{CT}$ 's terminology  $\mathbf{h}$  is  $\mathbf{k}$ .  $\mathcal{F}_{CT}$  will inform  $P_2$  (if  $A$  allows that) that a token has been created. An honest  $P_2$  sends a decode query  $(\text{Decode}, \text{tid}, k')$  where in Protocol 5's terminology  $k'$  is  $h'$ .  $\mathcal{S}$  will forward that to  $\mathcal{F}_{CT}$  and outputs whatever  $\mathcal{F}_{CT}$  outputs. It is easy to see that the ideal world view is indistinguishable from the real world view.
- *Token recipient  $P_2$  is corrupted:* This is also simple.  $\mathcal{S}$  will receive a retrieval query  $\text{Decode}_{vec}$  from  $A$  to operate on the token using the header  $h'$ .  $\mathcal{S}$  will send the command  $(\text{Corrupt-decode}, \text{tid}, h')$  to  $\mathcal{F}_{CT}$  and outputs to  $A$  whatever  $\mathcal{F}_{CT}$  outputs. Based on the biological construction of consumable tokens, applying a header  $h' \neq \mathbf{h}$  for all  $i \in [v]$  will cause MS to output  $\perp$  (the pulled-down protein sample is a random subset of the proteins in a token). MS will output  $\mathbf{m}_i$  only when any of  $\mathbf{h}_i$  is applied and the threshold of MS are satisfied (this happens in the first  $n$  pull-down and sequence invocations).  $\mathcal{F}_{CT}$  has a similar behavior. Furthermore, after the  $n^{\text{th}}$  query,  $\mathcal{F}_{CT}$  will always output  $\perp$ . By Lemma 2,  $\text{Decode}_{vec}$  will also output  $\perp$  after the  $n^{\text{th}}$  decode query. By Lemma 3, this applies also to the multi-vial token construction (since each vial will have the same behavior). Thus, the ideal world view is indistinguishable from the real world one.

This completes the proof of Theorem 3.  $\square$

<sup>20</sup>Similar analogy applies to a vector of secret messages. This vector will be decomposed into sub-vectors of message fragments, i.e., the first sub-vector will contain the first fragment of each message in  $\mathbf{m}$ , and so on. Then, each sub-vector will be stored in a separate vial.

## C Proofs

### C.1 Proof of Theorem 1

**Theorem 4 (Theorem 1 restated).** For  $0 \leq \gamma \leq 1$ , if each of  $f_1, \dots, f_u$  is as defined in Section 5.2, then Protocol 2 securely realizes for the point function family  $\mathcal{I} = \{I_{p,m} | p \in \mathcal{P}, m \in \mathcal{M}\}$  in the  $\mathcal{F}_{CT}$ -hybrid model in the presence of any adversary  $A \in \mathcal{A}$ , with  $n_q = n$  and large enough  $u$ .

*Proof.* We start with proving that our construction is efficient in the sense of Definition 6.1, then we show how Protocol 2 securely realizes in the  $\mathcal{F}_{CT}$ -hybrid model.

**Efficiency.** Protocol 2 consists of regular computational operations, all of which takes polynomial time, and interactions with  $\mathcal{F}_{CT}$ . We need to show that  $\mathcal{F}_{CT}$  executes all commands sent to it efficiently. This clearly depends on the concrete construction used to realize  $\mathcal{F}_{CT}$ , which in our case, is the physical construction described in Section 4.2. All procedures performed within any of the  $u$  Encode and Decode queries that Protocol 2 issues, can be conducted in labs with appropriate equipment within technologically realizable time steps. The number of these steps is polynomial in the sense that computationally-bounded parties are able to receive the output, whether it is  $m$  or  $\perp$ , within polynomial time (in the security parameter  $\kappa$ ).

Next, we define an ideal model adversary  $\mathcal{S}$  that interacts with the environment and simulates a copy of the real-life adversary  $A \in \mathcal{A}$ . For simplicity, we let  $A$  be a proxy for the environment, and hence, the task of  $\mathcal{S}$  is to simulate a view for  $A$  in the ideal world (when interacting with ) that is computationally indistinguishable from the real world view (when executing Protocol 2), where both are in the  $\mathcal{F}_{CT}$ -hybrid model.

We analyze four cases; corrupted  $P_1$  ( $A$  will be controlling the obfuscator), corrupted  $P_2$  ( $A$  will be controlling the evaluator), corrupted  $P_1$  and  $P_2$  ( $A$  will be controlling both parties), and honest  $P_1$  and  $P_2$  ( $A$  will not be controlling any of them). Furthermore, we present our analysis for the case of a static adversary class  $\mathcal{A}$ . That is, the environment will choose which party to corrupt at the onset of the protocol. In Remark 5, we show how this analysis can be extended to deal with adaptive adversaries who can corrupt parties at any point of time during the protocol execution.

**Case 1 - Corrupted  $P_1$ .** To obfuscate a point function, a corrupted  $P_1$  will submit Corrupt-encode queries to  $\mathcal{F}_{CT}$  (see step 3(d), under Obf in Protocol 2). Based on the number and content of these queries, any of the following cases may occur: Submitting invalid or fewer than  $u$  Corrupt-encode queries, exactly  $u$  queries, or more than  $u$  queries. We show how to handle each of these cases separately.

1.a Invalid or fewer than  $u$  queries:  $A$ , through the corrupted  $P_1$ , may not submit any Corrupt-encode query, or submit fewer than  $u$  queries (recall that  $u$  tokens need to be created when obfuscating any function  $I_{p,m}$ ), or even submit invalid ones (e.g., invalid format). All these cases represent an invalid obfuscation request. Thus,  $\mathcal{S}$  will not send anything and no obfuscation state will be created. Later on, if  $P_2$  requests an evaluation over some password, will do nothing. This is equivalent to what happens in the real world,  $P_2$  may not receive any tokens, or receive fewer than  $u$  tokens. In both cases,  $P_2$  cannot evaluate the obfuscated  $I_{p,m}$ .

1.b Exactly  $u$  queries: Here,  $P_1$  submits  $u$  valid Corrupt-encode queries to  $\mathcal{F}_{CT}$ , each of which contains some message share  $m_i$ , some token key  $k_i$ , and some randomness  $r_i$ .  $\mathcal{S}$  uses this information to generate a Corrupt-obfuscate query for as follows. Recall that  $\mathcal{S}$  is simulating  $\mathcal{F}_{CT}$  locally, as well as the random oracles representing the functions  $f_1, \dots, f_u$ . Thus, it gets to see the full content of all queries that  $A$  submits to  $\mathcal{F}_{CT}$  and each  $f_i$ . This allows  $\mathcal{S}$  to easily define the point function  $I_{p,m}$  (if any) submitted by  $A$ . In detail,  $\mathcal{S}$  records the password and random string that  $A$  sends to each  $f_i$ , which we denote as  $(p', r')$ , along with the key  $k'$  that  $\mathcal{S}$  returned to  $A$ . Later on,  $\mathcal{S}$  can search this record to find the password  $p$  that corresponds to the keys encapsulated in the Corrupt-encode queries. It also can compute  $m$  using the shares found in these queries as  $m = \oplus_{i=1}^u m_i$ . At the end,  $\mathcal{S}$  submits the query (Corrupt-obfuscate, sid,  $P_1, P_2, I_{p,m}$ ) to , and it creates a local state recording the token instances information including their tid,  $k_i, r_i$ , and  $m_i$ .

Two edge cases may occur here that  $\mathcal{S}$  has to check:

- $A$  may not submit a well-defined point function  $I_{p,m} \in \mathcal{I}$ , meaning that  $A$  may not use keys generated properly using the same password. This can be verified by checking the record that  $\mathcal{S}$  maintains for each  $f_i$ . If this is the case, then the obfuscation request is considered for circuit  $C = \perp$  and  $\mathcal{S}$  submits the query (`Corrupt-obfuscate`, `sid`,  $P_1, P_2, C$ ) to  $\cdot$ . This is equivalent to what happens in the real world, an invalid point function cannot be obfuscated, so we have  $C = \perp$ .
- $A$  may not permit transferring all  $u$  tokens to  $P_2$  despite submitting valid  $u$  `Corrupt-encode` queries. Recall that  $\mathcal{F}_{CT}$  asks the permission of  $A$  to send the tokens to  $P_2$ . Hence,  $\mathcal{S}$  can track the number of tokens that were allowed to be sent. If at least one token is not sent, then no obfuscation state will be created. This is also equivalent to what happens in the real world, if the adversary does not allow transferring all tokens, then  $P_2$  does not have the obfuscated function which require  $u$  token instances.

For evaluating the obfuscated function, honest  $P_2$  will submit the command (`Evaluate`, `sid`,  $x$ ) to  $\cdot$ .  $P_2$  will receive nothing if no obfuscation state has been created yet,  $\perp$  if the obfuscated circuit  $C = \perp$ , or  $m$  if an obfuscation state for  $C \neq \perp$  and  $x$  is the valid point (otherwise, it will obtain  $\perp$ ).

It is easy to see that the adversary view is identical to his view in the ideal world.

1.c More than  $u$  queries: A corrupted  $P_1$  may submit more than  $u$  `Corrupt-encode` queries and  $A$  may allow sending more than  $u$  tokens to  $P_2$ . Recall that  $\mathcal{F}_{CT}$  creates a token and also transfers it to  $P_2$  (if  $A$  allows that) as part of the `Corrupt-encode` query implementation. Furthermore, recall that  $P_2$ , according to Protocol 2, will wait only for the first  $u$  tokens and use them to evaluate the obfuscated function, i.e., it will ignore the rest of the tokens. Thus,  $\mathcal{S}$  will use the first (correctly-formatted)  $u$  `Corrupt-encode` queries to simulate the view for  $A$  in a similar way as discussed above.

**Case 2 - Corrupted  $P_2$ .** Honest  $P_1$  submits (`Obfuscate`, `sid`,  $P_1, P_2, I_{p,m}$ ) to  $\cdot$ , where  $\mathcal{S}$  does not know the function  $I_{p,m}$ . If  $A$  does not allow sending the obfuscated function to  $P_2$ , then  $\cdot$  will not create any obfuscation state, and all `Evaluate` queries will output nothing. Thus,  $\mathcal{S}$ , who is simulating  $\mathcal{F}_{CT}$  will not notify  $P_2$  about any token instances creation. This is equivalent to what happens in the real world, when  $A$  does not allow sending the obfuscated function,  $P_2$  will not receive any tokens.

On the other hand, if  $A$  allows sending the obfuscated function to  $P_2$ , then  $\mathcal{S}$  will notify  $P_2$  that  $u$  tokens have been created by  $P_1$ . Recall that through the `Corrupt-decode` interface,  $\mathcal{F}_{CT}$  permits corrupted  $P_2$  (or  $A$ ) to query each token  $n$  times. In total,  $A$  can submit  $nu$  queries. Any incorrectly-formatted query (one that does not follow the correct format or does not provide the required inputs) will be ignored, and hence, will not be counted. In order to simulate the protocol view for  $A$ ,  $\mathcal{S}$  needs to reply to each `Corrupt-decode` in a way that accounts for the possibility that  $A$  might be able to recover  $m$ .  $\mathcal{S}$  needs to extract the passwords (if any) that correspond to the `Corrupt-decode` queries that  $A$  submits, and then use them to generate up to  $n_q$  `Evaluate` commands for  $\cdot$  and see if any of them will output  $m$  (we show later in Lemma 4 that  $n_q = n$ ).

In order to do so,  $\mathcal{S}$  will prepare a local state consisting of dummy tokens generated as follows:

- $\mathcal{S}$  chooses a password  $p' \in \mathcal{P}$  at random,<sup>21</sup> some random message shares  $m'_1, \dots, m'_u$ , and some random strings  $r_1, \dots, r_{u-1}$ .
- $\mathcal{S}$  maps  $p'$  to a set of keys using  $f_1, \dots, f_u$  and the random strings  $r_1, \dots, r_{u-1}$  as done in Protocol 2.
- $\mathcal{S}$  stores this information in an array  $T$  of size  $u$  representing the  $u$  token instances.

$\mathcal{S}$  will use the array  $T$  to reply to  $A$ 's `Corrupt-decode` queries. To simplify the presentation, we discuss the simple case in which  $A$  starts with querying the first token. Then, we show how the simulation extends to any arbitrary order of `Corrupt-decode` queries.

2.a Simple case. The simulation for this case proceeds as follows:

- $A$  submits the  $j^{\text{th}}$  query (`Corrupt-decode`, `tid`<sub>1</sub>,  $k'_{j,1}$ ) for the first token.
- $\mathcal{S}$  extracts the password  $p'_j$  that corresponds to the input key (if any) by searching the random oracle record as before, i.e., set  $p'_j = p'$  such that  $f_1(p') = k'_{j,1}$ .
- If no password is found in the oracle records, then  $\mathcal{S}$  leaves the token array  $T$  as is.
- Else,  $\mathcal{S}$  submits the command (`Corrupt-evaluate`, `sid`,  $p'_j$ ) to  $\cdot$

<sup>21</sup>Similar to [21], we assume for convenience that the password space  $\mathcal{P}$  is super-polynomial in size. In particular,  $p'$  will be different from all passwords input to the protocol with all but negligible probability.

- If it receives  $m$  back, this means that the password  $p'_j$  is correct.  $\mathcal{S}$  then replaces the dummy message share  $m_u$  for the last token entry in  $T$  with one that allows constructing  $m$ . That is, it sets  $m_u = m \oplus_{i=1}^{u-1} m'_i$ .  $\mathcal{S}$  also replaces all dummy token keys in  $T$  with correct ones generated using the correct password  $p'_j$  (combined with the same random strings  $r_1, \dots, r_{u-1}$  that  $\mathcal{S}$  used when constructing  $T$ ).
  - If  $\mathcal{S}$  returns  $\perp$  (meaning that  $p'_j$  is incorrect),  $\mathcal{S}$  leaves the token array  $T$  as is.
- $\mathcal{S}$  uses  $T$  to reply to any **Corrupt-decode** query as described in the definition of  $\mathcal{F}_{CT}$  (if  $A$  inputs a valid key, reply with  $m_i$ , else if it is a close key based on the affinity database  $D$ , then return  $m_i$  with probability  $\gamma'$ . Otherwise, reply with  $\perp$ ).

Accordingly, if  $A$  does not know (or successfully guess) the correct password,  $\mathcal{S}$  will reply with dummy message shares. By Lemma 5, the probability that  $A$ , who does not know the dummy password used by  $\mathcal{S}$ , retrieves all dummy shares of  $m'$  (and in this case the environment will be able to distinguish the real world from the ideal world) is negligible for large enough  $u$ . On the other hand, if it happens that  $A$  knows (or successfully guessed) the correct password,  $A$  will retrieve the message  $m$  successfully as in the real world since  $T$  will contain the correct information that corresponds to  $I_{p,m}$ . So, the output is the same in both worlds, and the distribution of the dummy info in  $T$  is the same as the distribution of the token info in the real world. Thus, the view of the environment (or  $A$ ) in the ideal world is indistinguishable from the one in real world.

*2.b General case.* If  $A$  does not start with the first token,  $\mathcal{S}$  uses  $T$  to respond to **Corrupt-decode** using the same technique as above. Although  $A$  can start querying any token in any order and skip the first token, the probability it hits the right key of any of these tokens is negligible even if  $A$  happened to know the correct password. This is because of the chaining construction. That is, generating the  $i^{\text{th}}$  token key for  $i > 1$  requires not only the password, but also the randomness stored in all previous tokens (where guessing this randomness succeeds with negligible probability). Also, and as will be shown by Lemma 5, the probability that  $A$  obtains all shares of  $m$  without guessing the password correctly using the first token is negligible for large enough  $u$ . Thus, it is better for  $A$  to start with the first token.

Nonetheless, assume  $A$  does not want to start with the first token.  $\mathcal{S}$  will handle this case follows:

- For an input key  $k_{i,j}$  from  $A$ ,  $\mathcal{S}$  searches the random oracle  $f_i$  record to check if there is a password  $p'_j$  that corresponds to this key.
- If such a record exists, and if  $\mathcal{S}$  outputs  $m$  for  $p'_j$ , then  $\mathcal{S}$  updates  $T$  as follows:
  - $\mathcal{S}$  repeats the same process as in the previous case to update the keys in  $T$ .
  - To update the message shares,  $\mathcal{S}$  will choose one of the tokens that its share was not retrieved by  $A$ . In other words,  $\mathcal{S}$  picks a token  $i$  that  $A$  did not succeed in retrieving its share  $m'_i$  and computes a new share value as  $m_i = m \oplus m'_i \oplus_{v=1}^u m'_v$ . By Lemma 5, the probability that  $A$  retrieves all shares of  $m$  is negligible for large enough  $u$ . Thus, with overwhelming probability  $\mathcal{S}$  will find at least one token to update its share.
- $\mathcal{S}$  continues replying to all **Corrupt-decode** queries using the array  $T$ . If  $A$  exceeds  $n$  queries for any token,  $\mathcal{S}$  will reply with  $\perp$  for all future **Corrupt-decode** queries over that token.

The indistinguishability argument for this case follows as in the simple case discussed above.

**Case 3 - Corrupted  $P_1$  and  $P_2$ .** This case is easy to handle. Since  $\mathcal{S}$  is simulating  $\mathcal{F}_{CT}$  and each  $f_i$  locally, it gets to see the content of all queries submitted by both  $P_1$  and  $P_2$ , and can respond to all **Corrupt-decode** queries submitted by  $P_2$ . As noted,  $\mathcal{S}$  is not using any dummy information to reply to any of the queries; all are taken from  $P_1$ 's queries. Thus, the ideal world view generated by  $\mathcal{S}$  is indistinguishable from the real world one.

**Case 4 - Honest  $P_1$  and  $P_2$ .** Here recall that if both parties are honest all what  $A$  sees is the request to deliver the obfuscated circuit to  $P_2$ . Nothing about the content of any of the **Obfuscate** and **Evaluate** commands, nor  $\mathcal{S}$ 's response, will be revealed to  $A$ . Thus, the task of  $\mathcal{S}$  is just to simulate this delivery request. When  $P_1$  submits the command (**Obfuscate**, sid,  $P_1, P_2, I_{p,m}$ ),  $\mathcal{S}$  simulates submitting the delivery of  $u$  tokens by asking  $A$ 's permission to send these tokens to  $P_2$ , and then notifying  $P_2$  about them if  $A$  agrees. Thus, the ideal world view generated by  $\mathcal{S}$  is indistinguishable from the real world one.

It remains to show that Protocol 2 does not amplify the number of queries  $A$  obtains, meaning that  $n_q = n$ , and that, when  $u$  is configured properly, the soundness error of Protocol 2 is negligible. These are shown in Lemmas 4 and 5 below.

**Lemma 4.** *Given the parameters listed in Theorem 1, Protocol 2 has  $n_q = n$ .*

*Proof.* Recall that the leftover attack under the construction outlined by Protocol 2 succeeds with negligible probability. Also, recall that operating on any of the  $u$  token instances does not provide  $A$  with any additional information to enhance its chances in recovering the share from any given token. Since  $A$  can perform up to  $n$  **Corrupt-decode** queries to retrieve any share, and that all shares are needed to recover  $m$ , this is equivalent to having  $n$  **Corrupt-evaluate** queries in total to retrieve  $m$ .  $\square$

**Lemma 5.** *Given the parameters listed in Theorem 1, Protocol 2 has negligible soundness error for large enough  $u$ .*

*Proof.* For simplicity, the proof considers uniform password distribution ( Remark 4 shows how to generalize for arbitrary password distributions). Also, we let each function  $f_i$  have an output space of size  $|\mathcal{S}^{\mathcal{P}}| = |\mathcal{P}|$ .<sup>22</sup>

Recall that with a single token, the probability that  $A$  obtains  $m$  without knowing the correct password is  $\frac{n}{|\mathcal{P}|} + (1 - \frac{n}{|\mathcal{P}|})\gamma$ , where  $\gamma$  is non-negligible. Our goal is to show that with chaining construction, this probability can be amplified  $\frac{n}{|\mathcal{P}|} + \text{negl}(\kappa)$  for a security parameter  $\kappa$ . We now proceed to prove that.

An adversary  $A$  will be given  $u$  tokens each stores a message share  $m_i$ , concatenated with some random string, under a key  $k_i$  as described in Protocol 2. For the case of  $u = 1$ , and as mentioned previously,  $A$  will obtain  $m$  with the following probability:

When  $u > 1$ ,  $A$  will submit decode queries each of which containing a token key guess.  $A$  has to retrieve each share  $m_i$  for  $i \in [u]$ . The chaining construction affects the success probability of  $A$  since knowing the subspace of keys that corresponds to the password space for the  $j^{\text{th}}$  token requires all random strings stored in previous tokens for  $i < j$ . The success probability of  $A$  is the probability of recovering the message  $m$ , which can be expressed as follows (where  $\text{ct}_i$  is the token storing  $m_i$ ):

$$\Pr[m] = \Pr\left[\bigcap_{i=1}^u A(\text{ct}_i) = m_i\right]$$

To simplify the discussion, we refer to the above as  $\Pr[m] = \Pr[m_1, \dots, m_u]$ , where knowing  $m_i$  implies knowing  $r_i$  since they are stored together. Note that composing multiple tokens together will not give  $A$  any extra advantage. This is due to the following. First, the keys used to generate the tokens are chosen independently at random. Second, by the definition of the adversary class  $\mathcal{A}$  and the key affinity database, all what  $A$  can do is to search the database for keys close to some key of his choice. Such information will not be helpful since keys are chosen at random. Thus, it is just like having  $A$  operating on each token separately (in any order he wishes), and each key guess  $A$  tries has an independent success probability.

For  $u = 1$ , we have:

$$\Pr[m] = \Pr[m_1] = \frac{n}{|\mathcal{P}|} + \left(1 - \frac{n}{|\mathcal{P}|}\right)\gamma$$

For  $u = 2$ , we have  $\Pr[m] = \Pr[m_1, m_2]$ , which can be computed as:<sup>23</sup>

$$\begin{aligned} \Pr[m_1, m_2] &= \frac{n}{|\mathcal{P}|} + \gamma \left(1 - \frac{n}{|\mathcal{P}|}\right) \left(\frac{n}{|\mathcal{P}|} + \gamma \left(1 - \frac{n}{|\mathcal{P}|}\right)\right) \\ &= \frac{n}{|\mathcal{P}|} + \frac{n}{|\mathcal{P}|} \left(1 - \frac{n}{|\mathcal{P}|}\right)\gamma + \left(1 - \frac{n}{|\mathcal{P}|}\right)^2 \gamma^2 \end{aligned}$$

That is, guessing  $k_1$  correctly (which happens with probability  $\frac{n}{|\mathcal{P}|}$ ) allows retrieving both shares with probability 1 ( $k_1$  allows recovering  $m_1$  and the randomness  $r_1$ , as well as deducing the password

<sup>22</sup>This can be extended to cases where the key space for tokens  $i > 1$  covers  $\mathcal{K}$ , i.e.,  $|\mathcal{S}^{\mathcal{P}}| = |\mathcal{K}|$ . This will reduce the number of token instances  $u$  needed to satisfy a certain soundness error bound.

<sup>23</sup>The exact probability of guessing the correct password for the  $i^{\text{th}}$  token should be  $\frac{n}{|\mathcal{P}| - (i-1)n}$ , but for a large password space  $|\mathcal{P}| - (i-1)n \approx |\mathcal{P}|$ .

$p$ ,<sup>24</sup> which suffice to compute  $k_2$  and retrieve  $m_2$ ). If all guesses of  $k_1$  were incorrect (which happens with probability  $1 - \frac{n}{|\mathcal{P}|}$ ), then the probability of obtaining  $m_1$  is  $\gamma$ , and the probability of obtaining  $m_2$  depends on whether  $k_2$  is guessed correctly or not.

For  $u = 3$ , we have  $\Pr[m] = \Pr[m_1, m_2, m_3]$ , which can be computed as:

$$\begin{aligned} \Pr[m_1, m_2, m_3] &= \frac{n}{|\mathcal{P}|} + \gamma \left(1 - \frac{n}{|\mathcal{P}|}\right) \left(\frac{n}{|\mathcal{P}|} + \gamma \left(1 - \frac{n}{|\mathcal{P}|}\right) \left(\frac{n}{|\mathcal{P}|} + \gamma \left(1 - \frac{n}{|\mathcal{P}|}\right)\right)\right) \\ &= \frac{n}{|\mathcal{P}|} + \gamma \left(1 - \frac{n}{|\mathcal{P}|}\right) \left(\frac{n}{|\mathcal{P}|} + \frac{n}{|\mathcal{P}|} \left(1 - \frac{n}{|\mathcal{P}|}\right) \gamma + \left(1 - \frac{n}{|\mathcal{P}|}\right)^2 \gamma^2\right) \\ &= \frac{n}{|\mathcal{P}|} + \frac{n}{|\mathcal{P}|} \left(1 - \frac{n}{|\mathcal{P}|}\right) \gamma + \frac{n}{|\mathcal{P}|} \left(1 - \frac{n}{|\mathcal{P}|}\right)^2 \gamma^2 + \left(1 - \frac{n}{|\mathcal{P}|}\right)^3 \gamma^3 \end{aligned}$$

This is done by extending the analogy of computing  $\Pr[m_1, m_2]$  described above to three shares while observing that guessing  $k_2$  correctly (and with the knowledge of  $r_1$ ) allows deducing  $p$ , and hence, computing  $k_3$  and obtaining  $m_3$  successfully.

Continuing in this manner allows deriving a formula to compute  $\Pr[m] = \Pr[m_1, \dots, m_u]$  as follows:

$$\Pr[m_1, m_2, \dots, m_u] = \frac{n}{|\mathcal{P}|} + \left(1 - \frac{n}{|\mathcal{P}|}\right)^u \gamma^u + \frac{n}{|\mathcal{P}|} \sum_{i=1}^{u-1} \left(1 - \frac{n}{|\mathcal{P}|}\right)^i \gamma^i \quad (4)$$

The last term has an upper bound of  $\frac{n}{|\mathcal{P}|} \left(1 - \frac{n}{|\mathcal{P}|}\right) \gamma$ , which for large password space will be very small since  $\frac{n}{|\mathcal{P}|}$  will be very small.<sup>25</sup> The second term can be made negligible by setting  $u$  to be large enough. This makes  $\Pr[m] = \frac{n}{|\mathcal{P}|} + \text{negl}(\kappa)$ , which completes the proof.  $\square$

This completes the proof of Theorem 1.  $\square$

*Remark 4 (On arbitrary password distribution).* The above analysis also works for arbitrary password distributions. It suffices to replace  $\frac{n}{|\mathcal{P}|}$  with the probability of guessing the correct password when performing  $n$  queries based on the underlying password distribution chosen in the protocol.

*Remark 5 (On adaptive adversaries).* The simulation strategy described above can be used to show that Protocol 2 is secure against adaptive adversaries. Note that  $P_1$  does not receive any output. Thus, corrupting this party later in the protocol will not impact the simulation. If anything, it makes the simulation easier since corrupting  $P_1$  will reveal its state which contains  $I_{p,m}$ . Similarly, corrupting  $P_2$  after performing the first (honest) Evaluate query has no impact. At that time, the tokens are consumed and all future Corrupt-decode queries made by  $A$  will output  $\perp$ . Corrupting  $P_2$  after the protocol starts but before performing any Evaluate query is the same as *Case 2 - Corrupted  $P_2$*  discussed before, with the difference that  $\mathcal{S}$  gets additional useful information, i.e., the password  $p$ , which is part of  $P_2$ 's state.

## C.2 Security Arguments for Intermediate $(1, n)$ -time Program Constructions

**VBB-based construction of  $(1, n)$ -time programs.** This is for the first intermediary construction for  $(1, n)$ -time programs shown in Section 6.2. We show a sketch of the simulator  $\mathcal{S}$  in the  $\mathcal{F}_{CT}$ -hybrid model, where we let the adversary  $A$  be a proxy for the environment. We have two cases:

- $P_1$  is corrupted: Here adversary  $A$  is controlling party  $P_1$ . The simplest case is that  $P_1$  sends a correct program  $\text{Prog}$  to  $\mathcal{S}$ , which  $\mathcal{S}$  can see, and thus can evaluate on its own, in addition to valid token encode queries that  $\mathcal{S}$  will implement. It is easy to simulate as  $\mathcal{S}$  can send an obfuscated version of the program to  $P_2$ , and can reply to the decode queries coming from  $P_2$  without even interacting with  $P_1$ . Other cases may involve  $P_1$ 's sending an invalid program  $\text{Prog}$  or invalid/incomplete token encoding query (all these cases are similar to these appearing in the security proof of the bounded-query point function obfuscation and discussed in details in Appendix C.1). All these are equivalent to obfuscating an invalid function, and thus it always outputs  $\perp$ . Replying to the token decode queries can be also easily handled since  $\mathcal{S}$  knows the full information about the token from the encode queries that  $A$  submitted.

<sup>24</sup>With  $k_1$ ,  $A$  can search the password space and find the password  $p$  that satisfies  $k_1 = f_1(p)$ .

<sup>25</sup>Recall when we set  $|\mathcal{S}^{\mathcal{P}}| = |\mathcal{K}|$ , this term will become  $\frac{n}{|\mathcal{K}|}$ , which is negligible.

- $P_2$  is corrupted: Here adversary  $A$  is controlling party  $P_2$ , and hence, it gets to query the program up to  $n$  times over inputs  $x_1, \dots, x_n$ . The simulator implements  $\mathcal{F}_{CT}$  locally and it prepares an obfuscation of program  $\text{Prog}$  with only the trapdoor path (since  $\mathcal{S}$  does not know the program that an honest  $P_1$  has submitted to) with some decryption key  $sk$  that  $\mathcal{S}$  chooses, and a mapping table  $\text{Tab}_1$  that maps the message space to the program input space. This version is denoted as  $bP'$ , which  $\mathcal{S}$  sends it to  $A$ .

When  $A$  queries the token  $\text{ct}$  over some key  $k'$ ,  $\mathcal{S}$  finds which  $x$  corresponds to  $k'$  using  $\text{Tab}_2$  (that is, it searches for  $x$  such that  $\text{Tab}_2[x] = k'$ ). Then, it queries  $\text{Prog}$  over  $x$  and gets the output back denoted as  $y$ . After that,  $\mathcal{S}$  retrieves  $m_0 = \text{Tab}_1[x]$ , encrypts  $y$  as  $m_1 = \text{Encrypt}(sk, y)$  and sends  $m = m_0 \parallel m_1$  to  $A$ . When  $A$  queries  $bP'$  over  $m$ , it obtains the correct output  $y$ .

If the key  $k'$  does not correspond to a valid input  $x$ , then  $\mathcal{S}$  replies with  $m = m_0 \parallel \phi^{\ell_{out}}$ , for some random  $m_0$  such that  $m_0 \notin \text{Tab}_1$ . This will cause the output of  $bP'$  to be  $\perp$ .

It is easy to see that the view of  $A$  in the ideal world is indistinguishable from the real world. This is mainly due to the use of VBB obfuscation, which does not allow  $A$  to distinguish a real obfuscation of  $f$  from a simulated one that contains only the trapdoor path.

**VBB-based construction of  $(1, n)$ -time programs with linear error correcting codes.** This is for the second intermediary construction for  $(1, n)$ -time programs shown in Section 6.2, in which we extend the function domain using linear codes. We show a sketch for the simulator in the  $\mathcal{F}_{CT}$ -hybrid model. We have two cases:

- $P_1$  is corrupted: This is similar to the case seen in the previous construction.
- $P_2$  is corrupted: At a high level,  $\mathcal{S}$  does not get to see the program that  $P_1$  want to obfuscate.  $\mathcal{S}$  prepares an obfuscation of  $\text{Prog}$  with the trapdoor path only, and for that it generates both the decryption key  $sk$  and the PRG seed  $r$ .  $\mathcal{S}$  will receive  $\text{Decode}$  queries from the adversary  $A$  and will reply with message payloads generated using the PRG as described before.

However, in order to obtain the program output on a given input from  $A$ ,  $\mathcal{S}$  needs to find out the codeword (if any) that  $A$  wants.  $A$  may query the tokens out of order. Thus,  $\mathcal{S}$  needs to keep track of when all  $\omega$  tokens are queried and see if a valid codeword exists in the set of queries so far. Worst case scenario, each time  $\mathcal{S}$  needs to check  $n^{\omega-1}$  combinations of queries to find a valid codeword (if any), which is a polynomial number ( $n$  is a small constant, and  $\omega = |K|$  is of a polynomial value). If a valid codeword exists,  $\mathcal{S}$  finds out the corresponding  $x$ , queries  $\text{Prog}$  to obtain the output over this  $x$ , and embeds an encryption of the output inside the reply to this last query (i.e., inside the string  $\phi^{n(|x|+\ell_{out})}$  appended to substring of the PRG output). In particular,  $\mathcal{S}$  will replace the first  $|x| + \ell_{out}$  bits with  $x$  concatenated with the output ciphertext.

It could be the case that a query allows forming more than one valid codeword. In this case,  $\mathcal{S}$  will have more than one input value  $x$  each of which has its own output value. It will query  $\text{Prog}$  as above and embed these inputs and their corresponding outputs in the string  $\phi^{n(|x|+\ell_{out})}$ .

As before, it is easy to see that the view of  $A$  in the ideal world is indistinguishable from the real world. This is mainly due to the use of VBB obfuscation, which does not allow  $A$  to distinguish a real obfuscation of  $f$  from a simulated one that contains only the trapdoor path.

### C.3 Proof of Theorem 2

**Theorem 5 (Theorem 2 restated).** *Assuming sup-exponentially secure  $i\mathcal{O}$  and one way functions, the  $i\mathcal{O}$ -based construction described above is a  $(1, n)$ -time program in the  $\mathcal{F}_{CT}$ -hybrid model.*

*Proof.* The proof is similar to the proof of the VBB-based construction in the sense that there will be two cases: corrupted  $P_1$  and corrupted  $P_2$ . The case of corrupted  $P_1$  is identical to the VBB-based construction. This is because  $\mathcal{S}$  gets to see the function submitted by  $P_1$ , and hence, can prepare an obfuscation program identical to what  $A$  sees in the real world. Since the two programs have equivalent functionality over all inputs, by the security of  $i\mathcal{O}$ ,  $A$  cannot distinguish between the simulated obfuscation and the real world one.

On the other hand, and similar to before, when  $P_2$  is corrupted and  $P_1$  is honest,  $\mathcal{S}$  hands  $P_2$  an obfuscated program that contains only the trapdoor path. The program that underlies the simulated

Simulated Program  $\text{Prog}_{G,n,sk,r,f}^{sim}$

**Input:**  $m, x$

**Description:**

1. Parse  $m$  as  $m_0 \parallel \dots \parallel m_{\omega-1}$ , and parse each  $m_i$  as  $m_i^0 \parallel m_i^1$
  2. Use  $G$  to compute the codeword  $\mathbf{c}$  that corresponds to  $x$
  3. Check that  $m$  corresponds to a valid codeword: Let  $B = \text{PRG}(r)$ , if  $\exists B[i, \mathbf{c}[i]] \neq m_i^0$ , then output  $\perp$
  4. Set  $y_i = \text{Decrypt}(sk, m_i^1)$  for all  $i \in \{0, \dots, \omega - 1\}$
  5. If  $\exists y_i \neq \phi^{n(|x|+\ell_{out})}$ , then take the first such  $y_i$  and do the following:
    - Parse  $y_i$  as  $y_i^0 \parallel \dots \parallel y_i^{n-1}$
    - Parse each  $y_i^j$  as  $y_i^{j,0} \parallel y_i^{j,1}$  (for  $j \in \{0, \dots, n - 1\}$ )
    - Output  $y_i^{j,1}$  for which  $y_i^{j,0} = x$
- Else, output  $\perp$

**Fig. 15.** The program  $\text{Prog}_{G,n,sk,r,f}^{sim}$  that  $\mathcal{S}$  obfuscates for the case of an honest  $P_1$  and corrupted  $P_2$ .

obfuscation (denoted as  $\text{Prog}^{sim}$ ) can be found in Figure 15. This program agrees with the one that  $A$  receives in the real world (denoted as  $\text{Prog}^{real}$  and can be found in Figure 10 in Section 6) over only the  $n$  inputs that  $\mathcal{S}$  receives from  $A$ . For all other inputs, the simulated program will output  $\perp$ , while the real world one will output  $f(x)$ .

The only remaining issue is to show that  $A$  cannot distinguish between the simulated and real world obfuscated programs. Towards this goal, we use similar techniques to these used in the security proof of authenticated constrained encryption found in [13]. In particular, let  $M_0$  denote the set of  $n$  messages that correspond to the inputs that both the simulated and real world obfuscated programs agree on, and  $M_1$  denote the set of the messages corresponding to the rest of the inputs on which these obfuscated programs differ. We present a series of hybrid experiments  $H_0, \dots, H_{|M_1|+1}$ , each of which is indistinguishable from the previous one.

Let  $u_j$  be the lexicographically  $j^{th}$  element of  $M_1$ . We define hybrid  $H_j$  as follows:

**Hybrid  $H_j$**  (For  $1 \leq j \leq |M_1|$ ): In the  $j^{th}$  hybrid,  $\text{Prog}$  first checks whether the input  $m \leq u_j$ , and if so, it outputs  $\perp$ . Otherwise, it behaves in an identical way to  $\text{Prog}^{real}$  that the adversary receives in the real world. The hybrid program is described in Figure 16.

Thus,  $H_0$  corresponds to the real world obfuscated program, and  $H_{|M_1|+1}$  corresponds to the ideal world obfuscated program.

In order to show that  $H_0$  and  $H_{|M_1|+1}$  are indistinguishable, we need to show that each  $H_j$  and  $H_{j+1}$  are indistinguishable. We construct a series of intermediate hybrids  $H_{j,0}, \dots, H_{j,5}$ , where  $H_{j,0}$  is the same as  $H_j$  and  $H_{j,5}$  is the same as  $H_{j+1}$ . For every  $t \in \{0, \dots, 5\}$ , we prove that  $H_{j,t}$  and  $H_{j,t+1}$  are indistinguishable, which establishes that  $H_j$  and  $H_{j+1}$  are indistinguishable.

**Hybrid  $H_{j,0}$ :** This is the same as experiment  $H_j$ .

**Hybrid  $H_{j,1}$ :** This is the same as experiment  $H_{j,0}$ , except that we modify  $\text{Prog}$  as shown in Figure 17. We move the program parameters to constants for clarity, and we use an injective non-interactive commitment scheme [13] to compute commitments  $z_i$ , for  $i \in \{0, \dots, \omega - 1\}$  as shown in the figure. If the input message  $m = u_{j+1}$ , then  $\text{Prog}$  will check whether this  $m$  is consistent with the commitment to the message corresponding to the codeword of  $u_{j+1}$ . In other words, checking that  $m$  corresponds to a valid codeword when  $m = u_{j+1}$  has been moved to a later point (beyond its original location in line 3).

**Hybrid  $H_{j,2}$ :** This is the same as experiment  $H_{j,1}$ , except that the hardwired  $z_i$  values are now computed as  $z_i = \text{Commit}(0; r'_i)$  for some random strings  $r'_i$  (again,  $i \in \{0, \dots, \omega - 1\}$ ).

$j^{\text{th}}$  Hybrid Program  $\text{Prog}_{G,n,sk,r,f}$

**Input:**  $m, x$

**Description:**

1. Parse  $m$  as  $m_0 \parallel \dots \parallel m_{\omega-1}$ , and parse each  $m_i$  as  $m_i^0 \parallel m_i^1$
  2. Use  $G$  to compute the codeword  $\mathbf{c}$  that corresponds to  $x$
  3. Check that  $m$  corresponds to a valid codeword: Let  $B = \text{PRG}(r)$ , if  $\exists B[i, \mathbf{c}[i]] \neq m_i^0$ , then output  $\perp$
  4. Set  $y_i = \text{Decrypt}(sk, m_i^1)$  for all  $i \in \{0, \dots, \omega - 1\}$
  5. If  $\exists y_i \neq \phi^{n(|x|+\ell_{out})}$ , then take the first such  $y_i$  and do the following:
    - Parse  $y_i$  as  $y_i^0 \parallel \dots \parallel y_i^{n-1}$
    - Parse each  $y_i^j$  as  $y_i^{j,0} \parallel y_i^{j,1}$  (for  $j \in \{0, \dots, n - 1\}$ )
    - Output  $y_i^{j,1}$  for which  $y_i^{j,0} = x$
- Else:
- If  $m \leq u_j$ , then output  $\perp$
  - Otherwise, output  $f(x)$

**Fig. 16.** The program  $\text{Prog}_{G,n,sk,r,f}$  for hybrid  $H_j$ .

Program  $\text{Prog}_{G,n,sk,r,f}$  for  $H_{j,1}$

**Input:**  $m, x$ .

**Constants:**  $G, sk, r, u_{j+1}$  and its codeword  $\mathbf{c}'$ ,  $z_i = \text{Commit}(0; \text{PRG}(r)[i, \mathbf{c}'[i]])$  for  $i \in \{0, \dots, w - 1\}$

**Description:**

1. Parse  $m$  as  $m_0 \parallel \dots \parallel m_{\omega-1}$ , and parse each  $m_i$  as  $m_i^0 \parallel m_i^1$ .
  2. Use  $G$  to compute the codeword  $\mathbf{c}$  that corresponds to  $x$
  3. If  $m \neq u_{i+1}$ , check that  $m$  corresponds to a valid codeword: Let  $B = \text{PRG}(r)$ , if  $\exists B[i, \mathbf{c}[i]] \neq m_i^0$ , then output  $\perp$
  4. Set  $y_i = \text{Decrypt}(sk, m_i^1)$  for all  $i \in \{0, \dots, \omega - 1\}$
  5. If:
    - $m \leq u_j$  and  $\forall y_i = \phi^{n(|x|+\ell_{out})}$ , then output  $\perp$ .
    - $m = u_{j+1}$  and  $\exists \text{Commit}(0; m_i^0) \neq z_i$ , then output  $\perp$ .
  6. Otherwise, if  $\exists y_i \neq \phi^{n(|x|+\ell_{out})}$ , then take the first such  $y_i$  and do the following:
    - Parse  $y_i$  as  $y_i^0 \parallel \dots \parallel y_i^{n-1}$
    - Parse each  $y_i^j$  as  $y_i^{j,0} \parallel y_i^{j,1}$  (for  $j \in \{0, \dots, n - 1\}$ )
    - Output  $y_i^{j,1}$  for which  $y_i^{j,0} = x$
- Else, output  $f(x)$

**Fig. 17.** The program  $\text{Prog}_{G,n,sk,r,f}$  for hybrid  $H_{j,1}$ .

**Hybrid  $H_{j,3}$ :** This is the same as experiment  $H_{j,2}$ , except that the hardcoded  $z_i$  is now computed as  $z_i = \text{Commit}(1; r'_i)$ .

**Hybrid  $H_{j,4}$ :** This is the same as experiment  $H_{j,3}$ , except that we modify  $\text{Prog}$  to output  $\perp$  when  $m = u_{i+1}$  (without checking the commitment). Denote this modified version as  $\text{Prog}'$ . An equivalent description of  $\text{Prog}'$  is to remove the second bullet in line 5 in Figure 17, and change the first bullet in that line to be  $m \leq u_{j+1}$ .

**Hybrid  $H_{j,5}$ :** This is the same as experiment  $H_{j,4}$ , except that we remove the commitments  $z_i$ .

This completes the description of the hybrid experiments. Now we argue that these hybrids are indistinguishable.

*Indistinguishability of  $H_{j,0}$  and  $H_{j,1}$ .* These are indistinguishable by the security of  $i\mathcal{O}$ . The commitments are for the same strings that the PRG produces for the codeword. Also, moving the test of the valid codeword of  $m = u_{i+1}$  to line 5, does not change the program behavior; if the codeword is inconsistent, the output will be  $\perp$  as before. Same for the check of  $y_i$ , if a  $y_i$  containing the program output over  $x$  exists, this value will be output by **Prog**. Hence, the underlying programs of  $H_{j,0}$  and  $H_{j,1}$  have an equivalent functionality, and by the security of  $i\mathcal{O}$ , an adversary  $A$  cannot distinguish between the two hybrids.

*Indistinguishability of  $H_{j,1}$  and  $H_{j,2}$ .* These are indistinguishable by the security of the PRG (i.e., the PRG output is indistinguishable from random).

*Indistinguishability of  $H_{j,2}$  and  $H_{j,3}$ .* These are indistinguishable by the hiding property of the commitment scheme.

*Indistinguishability of  $H_{j,3}$  and  $H_{j,4}$ .* These are indistinguishable by the security of  $i\mathcal{O}$ . Since we have  $z_i = \text{Commit}(1; r'_i)$ , then the condition in the third bullet in line 6 will always fail. Thus, the output will always be  $\perp$  for  $m = u_{j+1}$  in **Prog** which is identical to the behavior of **Prog'**. Thus, by the security of  $i\mathcal{O}$ , an adversary  $A$  cannot distinguish between the two hybrids.

*Indistinguishability of  $H_{j,4}$  and  $H_{j,5}$ .* These are indistinguishable by the security of  $i\mathcal{O}$ . The removal of the constants  $z_i$  will not impact functionality since they are not used in either of the programs underlying both hybrids. Similarly, when the input is  $u_{j+1}$ , in hybrid  $H_{j,4}$  the output will be  $\perp$  since the condition in the second bullet in line 5 will fail. In  $H_{j,5}$ , the output will be also  $\perp$  because of the first bullet in line 5 (i.e., if  $m \leq u_{j+1}$  then output  $\perp$ ). For the rest of all inputs, both programs behave in an identical way. Thus, by the security of  $i\mathcal{O}$ , an adversary  $A$  cannot distinguish between the two hybrids.

This completes the proof of Theorem 2. □

*Remark 6 (On amplifying the soundness error of  $\mathcal{F}_{CT}$ ).* Recall that in our construction in Section 6, we assume that  $\mathcal{F}_{CT}$  has a negligible soundness error in our constructions (this is instead of the non-negligible soundness error  $\gamma$ ). Amplifying this error to negligible can be done using conventional approaches; share each  $m$  in  $\mathbf{m}$  as  $m = \bigoplus_{j=1}^t m_j$ , and store each share in a separate token instance under an independent key. Thus, for a vector of messages  $\mathbf{m}$ , we will have  $t$  vectors of message shares, each of which is stored in a separate token. Since these are independent, and an attacker needs all shares to construct any message, the soundness error will be reduced to  $\gamma^t$ . By setting  $t$  to be large enough, this quantity becomes negligible.