Unclonable Polymers and Their Cryptographic Applications

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

[1] University of Connecticut, ghada@uconn.edu
[2] Boston University, canetti@bu.edu
[3] Eleven Therapeutics and IDC Herzliya, erlichya@gmail.com
[4] Columbia University, {tal,itsik,tromer}@cs.columbia.edu
[5] Tel Aviv University, gershoni@tauex.tau.ac.il, roitburda@gmail.com

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, along with a guarantee that only few of these values are ever opened.

Indeed, k-time programs, first proposed by Goldwasser, Kalai, and Rothblum [35] 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 [13].

In [35] 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 [55], 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 bases 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 [12, 19, 27, 38]. 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"* [20]. Over billions of years of evolution, no known biological system has ever violates 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 obtains 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 relatively simple idealized definition of a *consumable token* in Section 3. In a nutshell, an $(1, n, v)$-time token is created with $2v$ values: keys $k_1, ..., k_v$ and messages $m_1, ..., m_v$, 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 keys equals k_j, the adversary obtains m_j. We assume that 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 4, 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 π, generates some digital rendering $\hat{\pi}$ of π, 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 $\pi(x_1), ..., \pi(x_n)$ for n adversarially chosen values $x_1, ..., x_n$.

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 π and then implementing one-out-of-two oblivious transfer for each input wire using a $(1, 1, 2)$-consumable token with $K = M = \{0, 1\}^\kappa$ [35]. 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 [9]), 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 [9, 42]) 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 [34] 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 [15]. 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. [44] 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., [18, 40, 41], with a foundational study in [37]. In general, two types of tokens are used: stateful [23] and stateless (or resettable) [7, 22]; the latter is considered a weaker and more practical assumption than the former. In

another line of work, Goldwasser et al. [35] 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 [39, 45]. 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 [13] 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. [52] 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 [21] and tamper-evident seals [48] were used to implement oblivious transfer and bit commitments. Others built cryptographic protocols for physical problems: [32] introduced zero knowledge proof system for nuclear warhead verification and [28] presented a unified framework for such proof systems with applications to DNA profiling and neutron radiography. This has been extended in [29] to build secure physical computation in which parties have private physical inputs and they want to compute a function over these inputs. Notably, [29] 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) [51] are hardware devices used as sources of randomness, that cannot be cloned. PUFs found several applications, such as secure storage [25], key management [43], oblivious transfer [53], and memory leakage-resilient encryption schemes [6]. The works [14] and [50] 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 [56], while [26] 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 combinitorial 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 the full version.[6]

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 field has focused on the use of DNA, generating an arsenal of molecular protocols to store and retrieve information [12, 19, 27, 38]. 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

[6] 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.

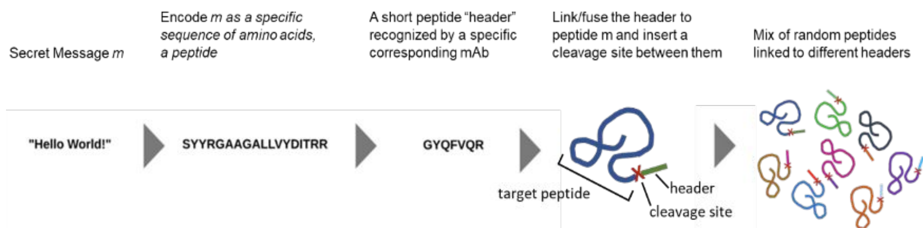


Fig. 1. General scheme for peptide-based data storage.

not known is unclonable and cannot be replicated or amplified, (iii) sequencing the peptide results in its consumption.[7]

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 the full version):

- *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 proteins is mass spectrometry (MS) or versions thereof [8,33]. This machinery imposes several conditions on

[7] 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.

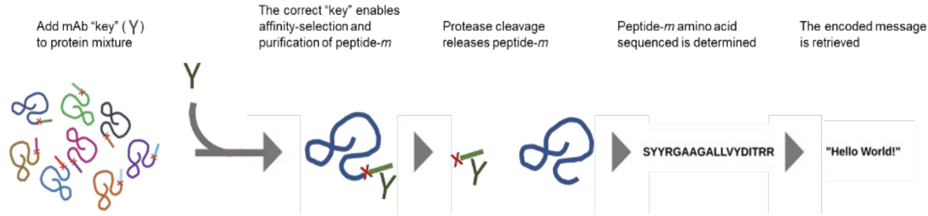


Fig. 2. Message retrieval.

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 that 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 the full version, 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),[8] while we amplify the soundness error as part of the cryptographic constructions that we build in later sections.

3 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 messagess, 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. Some preliminary notions that we use in our work can be found in the full version.

Notation. We use $[n]$ as a shorthand for $\{1, 2, \ldots, n\}$. 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 κ to denote the security parameter which is encapsulates two security parameters: κ_p for physical procedures and κ_c for computational algorithms. Thus, when we say polynomial in κ, this means polynomial in the $\mathsf{max}\{\kappa_p, \kappa_c\}$. Lastly, boldface letters represent vectors and PPT is a shorthand for probabilistic polynomial time.

3.1 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 γ (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.

[8] 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

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', γ') where k' is a close key to k and γ' 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 γ'. 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 γ' value (if any). Furthermore, recall that for any token the soundness error is upper bounded by γ. 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 κ, 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 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.[9] 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, hflag_1 and 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 ith 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 that P_1 is corrupted by setting $\mathsf{hflag}_1 = 0$. 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 γ' chance to obtain $\mathbf{m}_i$ if k' is close enough to key $\mathbf{k}_i$.

[9] 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 κ, a key affinity database D and a positive integer n.

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

Decode: Upon receiving the command $(\mathsf{Decode}, \mathsf{tid}, k')$ from P_2, if no token with ID tid exists, then end activation. Otherwise, retrieve $(\mathsf{tid}, P_1, P_2, \mathbf{k}, \mathbf{m}, v, j, \mathsf{hflag}_1, \mathsf{hflag}_2)$ 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 $\mathsf{out} = \mathbf{m}_i$, else set $\mathsf{out} = \perp$.
- Send $(\mathsf{tid}, \mathsf{out})$ to P_2.

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

Corrupt-decode: Upon receiving the command $(\mathsf{Corrupt\text{-}decode}, \mathsf{tid}, k')$ from the adversary, if no token with ID tid was created, end activation. Else, retrieve $(\mathsf{tid}, P_1, P_2, \mathbf{k}, \mathbf{m}, v, j, \mathsf{hflag}_1, \mathsf{hflag}_2)$. If $\mathsf{hflag}_2 = 1$ 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 $\mathsf{out} = \mathbf{m}_i$, else set $\mathsf{out} = \perp$ and $(\mathsf{close}, \gamma', i) = \mathsf{affinity}(D, \mathbf{k}, k')$. If $\mathsf{close} = 1$, choose $r \xleftarrow{\$} [0, 1]$ and change $\mathsf{out} = \mathbf{m}_i$ if $r \leq \gamma'$.
- Store $(\mathsf{tid}, P_1, P_2, \mathbf{k}, \mathbf{m}, v, j + 1, \mathsf{hflag}_1, \mathsf{hflag}_2 = 0)$.
- Send $(\mathsf{tid}, \mathsf{out})$ to the adversary.

Fig. 3. An ideal functionality for consumable tokens.

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 $\mathsf{affinity}$ 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 γ'. If $\mathsf{close} = 1$, this means that k' is close to $\mathbf{k}_i$, and hence, $\mathcal{F}_{CT}$ outputs $\mathbf{m}_i$ with probability γ'.

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

Protocol 1 (A Physical Construction of Consumable Tokens)

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

$\mathsf{Encode}_{\mathsf{phys}}(\mathbf{h}, \mathbf{m})$: 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 $\mathsf{peptide\text{-}m}_i$.
2. For each $\mathsf{peptide\text{-}m}_i$ 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.

$\mathsf{Decode}_{\mathsf{phys}}(h, S_P)$: Given a header $h \in \mathcal{H}$, and a protein vial S_p, do the following:

1. Immunoperciptate 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 $\mathsf{out} = m$. Otherwise, set $\mathsf{out} = \perp$. Output out.

Fig. 4. A physical construction of consumable tokens.

decode query, corrupting P_2 is allowed, and when the environment asks for that, the value of hflag_2 is set to 0.

3.2 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 $\mathcal{F}_{CT}$.[10] 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.

4 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.

[10]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.

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., [30, 31, 46, 49]. Barak et al. [9] 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 [11, 16, 17, 46, 54]. 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 [16]: 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 ($n_q \in \mathbb{N}$).

4.1 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 $I_{p,m} : \mathcal{P} \to \mathcal{M} \cup \{\perp\}$ 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.

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

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

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

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

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

Corrupt-evaluate: Upon receiving the command $(\mathsf{Corrupt\text{-}evaluate}, p')$ from the adversary, if no stored state exists, end activation. Else, retrieve $(p, m, j, \mathsf{hflag}_2)$ and do:
- If $j = n_q$, or $\mathsf{hflag}_2 = 1$ and $j > 0$, then end activation.
- Else, increment j, set $\mathsf{hflag}_2 = 0$, and if $p' = p$, set $\mathsf{out} = m$, else set $\mathsf{out} = \perp$.
- Output out to the adversary.

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

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, hflag_1 and 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 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 1 (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 $(\mathsf{Evaluate}, p')$ takes $q(t, \kappa)$ technologically-realizable time steps.*

4.2 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 γ, 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., $\frac{n_q}{|\mathcal{P}|} + \mathsf{negl}(\kappa)$ 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.[11]

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 $\pi_1, \ldots, \pi_u$ each of which maps any password $p \in \mathcal{P}$ to a random string of size ρ for some $\rho \in \mathbb{N}$. So we have $\pi_i : \mathcal{P} \to \{0,1\}^\rho$ and we denote the output space of each π_i as $\mathcal{S}^\mathcal{P} \subset \{0,1\}^\rho$ such that $|\mathcal{S}^\mathcal{P}| = |\mathcal{P}|$. 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 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

[11] 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.

scenario, meaning fixing the soundness error to be the maximum value γ, this multi-instance approach reduces the overall soundness error to γ^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 π_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 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 γ^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:[12] $\Pr[m] = \frac{un}{|\mathcal{P}|} + \left(1 - \frac{un}{|\mathcal{P}|}\right)\gamma^u$. 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, 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 jth 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 $f_1 : \mathcal{P} \to \mathcal{K}$ and $f_i : \mathcal{P} \times \{0,1\}^\kappa \to \mathcal{K}$ when $i > 1$. We write $k_i \leftarrow f_i(p, r'_i)$, where $r'_i = r_0 \oplus \cdots \oplus r_{i-1}$ such that $r_0 = \perp$ and $r_i \leftarrow \{0,1\}^\kappa$ is a random string stored in the ith token.

[12] For the jth 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 $|\mathcal{P}| - (j-1)n \approx |\mathcal{P}|$.

Each f_i first applies the mapping π_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 $\pi_i(p) \,\|\, r'_i$ as input and outputs a random string of size ρ, 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., $|\mathcal{P}|2^{\kappa} >> |\mathcal{K}|$. 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 $r_1||m_1$ 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^{κ}. Also, under the many-to-one construction of $f_2, \ldots, f_u$, 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: $\Pr[m] \approx \frac{n}{|\mathcal{P}|} + \big(1 - \frac{n}{|\mathcal{P}|}\big)\gamma^u$. 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,

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

For a security parameter κ, 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 $f_1, \ldots, f_u$ be as defined before such that $f_1 : \mathcal{P} \to \mathcal{K}$, and $f_i : \mathcal{P} \times \{0,1\}^\kappa \to \mathcal{K}$ 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 3. Construct a tuple of algorithms $(\mathsf{Obf}, \mathsf{Eval})$ 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 $m_1, \ldots, m_u$ 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 ct_i, with a unique token ID tid_i, encoding $r_i \parallel m_i$ using k_i by sending the command $(\mathsf{Encode}, \mathsf{tid}_i, P_1, P_2, k_i, r_i \parallel m_i, 1)$ to $\mathcal{F}_{CT}$.

Eval: on input an obfuscated function $\mathsf{o} = \{\mathsf{ct}_1, \ldots, \mathsf{ct}_u\}$ 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 ct_i using k_i to retrieve $r_i \parallel m_i$ by sending the command $(\mathsf{Decode}, \mathsf{tid}_i, k_i)$ 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 $\mathcal{I}$.

this amplifies the soundness error (and can be set to negligible with sufficiently large u) without increasing the number of queries A can do.[13]

4.3 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 κ. The proof can be found in the full version.

Theorem 1. *For $0 \leq \gamma \leq 1$, if each of $f_1, \ldots, f_u$ is as defined above, then Protocol 2 securely realizes $\mathcal{F}_{BPO}$ for the point function family $\mathcal{I} = \{I_{p,m} | p \in$*

[13]Similarly, to make the presentation easier, the probability is simplified here where some terms are omitted. See the full proof in the full version.

$\mathcal{P}, m \in \mathcal{M}\}$ 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, κ 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.

5 $(1, n)$-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.

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. [35] showed a construction for one-time programs that combines garbled circuits with one-time memory devices. Goyal et al. [37] strengthened this result by employing stateful hardware tokens to support unconditional security against malicious recipients and senders. Bellare et al. [10] 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. [24] 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. [36] 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.

5.1 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 2 (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 (*$\mathsf{Evaluate}, \cdot, x$*) takes $p(t, \kappa)$ technologically-realizable time steps.*

5.2 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 [9]). 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 Tab_1 and Tab_2. Tab_1 maps a program's input space $\mathcal{X}$ to the token

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

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

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

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

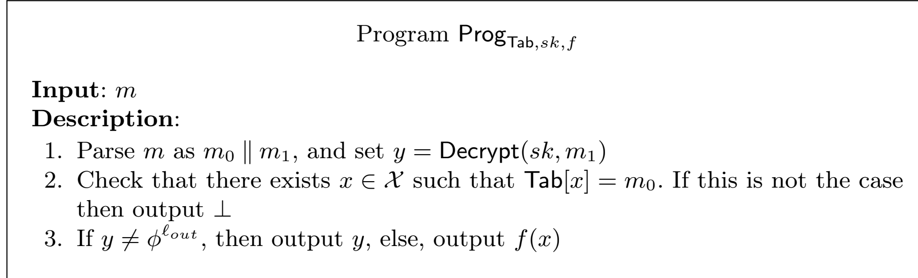
Corrupt-evaluate: Upon receiving the command $(\mathsf{Corrupt\text{-}evaluate}, x')$ from the adversary, if no stored state exists, end activation. Else:
- Retrieve $(C, j, \mathsf{hflag}_1, \mathsf{hflag}_2)$.
- If $\mathsf{hflag}_2 = 1$ and $j > 0$, or $j = n_q$, then end activation, else increment j, set $\mathsf{hflag}_2 = 0$, and send $(C(x'))$ to the adversary.

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

message space $\mathcal{M}$. This table is secret and will be part of the hidden program. While Tab_2 maps $\mathcal{X}$ to the token key space $\mathcal{K}$, and it is public.

We use 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, Prog is parametererized by a table $\mathsf{Tab} : \mathcal{X} \to \mathcal{M}$, 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 ϕ is some unique value outside the range of f and ℓ_{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 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 Tab_2 is secret hidden inside bP, the only way for P_2 to obtain a valid m is


Program $\mathsf{Prog}_{\mathsf{Tab},sk,f}$

Input: m
Description:
1. Parse m as $m_0 \parallel m_1$, and set $y = \mathsf{Decrypt}(sk, m_1)$
2. Check that there exists $x \in \mathcal{X}$ such that $\mathsf{Tab}[x] = m_0$. 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 $\mathsf{Prog}_{\mathsf{Tab},sk,f}$

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 the full version 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 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 C. We map each $x \in \mathcal{X}$ to a codeword of length ω, and we use ω 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 δ, meaning that the Hamming distance between any two legal codewords is at least δ. We represent each key in the set $K \subseteq \mathcal{K}$ used in creating a token, where $|K| = |M|$, 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 $\Sigma = \{0, \ldots, q-1\}$.

Definition 3 (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 δ. A generating matrix G of C is a $\omega \times d$-matrix whose rows generates 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 [47], 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 $\mathbf{s} \in S$.

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

For a security parameter κ, 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 3 but with negligible soundness error, and Tab_1 and Tab_2 are mapping tables as defined above. Construct a tuple of algorithms $(\mathsf{Encap}, \mathsf{Eval})$ for a $(1,n)$-time program scheme as follows.

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

1. Generate a token ct, with a unique token ID tid, encoding all messages $m \in \mathcal{M}$ each using a unique key from $\mathcal{K}$. This is done by sending the command $(\mathsf{Encode}, \mathsf{tid}_i, \mathcal{K}, \mathcal{M}, |\mathcal{M}|)$ to $\mathcal{F}_{CT}$.
2. Generate a random secret key $sk \in \{0,1\}^\kappa$.
3. Send ct, Tab_2, and $bP = \mathsf{VBB}(\mathsf{Prog}_{\mathsf{Tab}_1, sk, f})$ to P_2, where bP is an obfuscated version of the program $\mathsf{Prog}_{\mathsf{Tab}_1, sk, f}$ described in Figure 8.

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

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

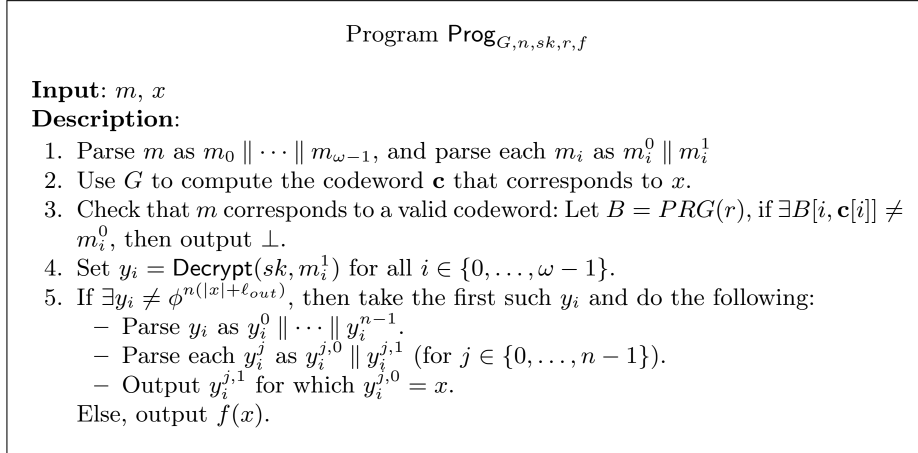
Fig. 9. A construction for a $(1,n)$-time program scheme for $\mathsf{Prog}_{\mathsf{Tab}, sk, f}$.

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 ω. Using this approach, we can cover a domain size $|S| = q^{d+1}$.

Accordingly, P_1 now has to generate ω tokens, denoted as $\mathsf{ct}_0, \ldots, \mathsf{ct}_{\omega-1}$, 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 $m_{0,0}, \ldots, m_{0,q-1}$, and in the second token be $m_{1,0}, \ldots, m_{1,q-1}$, and so on. We generate these messages using a pseudorandom generator with some random seed r. In particular, we have $m_{i,j} = PRG(r)[i,j]$ for all $i \in \{0, \ldots, \omega-1\}$ and $j \in \{0, \ldots, q-1\}$; 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.[14] Thus, to create token ct_0, P_1 will pass K and $m_{0,0}, \ldots, m_{0,q-1}$ to $\mathcal{F}_{CT}$, while for ct_1 the messages $m_{1,0}, \ldots, m_{1,q-1}$ along with K will be passed, etc.

So to execute 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 $\mathbf{c}$

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


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

Input: m, x
Description:
1. Parse m as $m_0 \parallel \cdots \parallel m_{\omega-1}$, 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 $B = PRG(r)$, if $\exists B[i, \mathbf{c}[i]] \neq m_i^0$, then output $\perp$.
4. Set $y_i = \mathsf{Decrypt}(sk, m_i^1)$ for all $i \in \{0, \ldots, \omega - 1\}$.
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 $y_i^0 \parallel \cdots \parallel y_i^{n-1}$.
   – Parse each y_i^j as $y_i^{j,0} \parallel y_i^{j,1}$ (for $j \in \{0, \ldots, n-1\}$).
   – Output $y_i^{j,1}$ for which $y_i^{j,0} = x$.

   Else, output $f(x)$.

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

to query the corresponding tokens. For example, if $\mathbf{c} = \{5, 9, 15, \ldots\}$, 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 $m = m_0 \parallel \cdots \parallel m_{\omega-1}$ will be used as input to Prog to obtain the output $f(x)$. This in turn means that 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 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 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 Encap now contains ω tokens beside the obfuscation of Prog. 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

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

For a security parameter κ, 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 3 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 psuedorandom generator, where $m \in \mathcal{M}$ and $|K| = q$ for $K \subseteq \mathcal{K}$. Construct a tuple of algorithms ($\mathsf{Encap}, \mathsf{Eval}$) 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 $(\mathsf{Encode}, \mathsf{tid}_i, \{k_0 \dots k_{q-1}\}, \{m_{i,0}, \dots, m_{i,q-1}\}, q)$ to $\mathcal{F}_{CT}$.
5. Send $\mathsf{ct} = \{\mathsf{ct}_0, \dots, \mathsf{ct}_{\omega-1}\}$ and $bP = i\mathcal{O}(\mathsf{Prog}_{G,n,sk,r,f})$ to P_2, where $\mathsf{Prog}_{G,n,sk,r,f}$ is defined in Figure 10.

Eval: On input a $(1, n)$-$\mathsf{Prog} = (\mathsf{ct}, bP)$ 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 $k'_{\mathbf{c}[i]}$ by sending the command $(\mathsf{Decode}, \mathsf{tid}_i, k'_{\mathbf{c}[i]})$ to $\mathcal{F}_{CT}$ and get m_i in return.
3. Output $\mathsf{out} = bP(m_0 \parallel \cdots \parallel m_{\omega-1}, x)$.

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

codewords. For example, codewords $\mathbf{c}_1 = \{5, 4, 13, 17\}$ and $\mathbf{c}_2 = \{5, 9, 12, 18\}$ have a cover of $\{5, 4, 9, 12, 13, 17, 18\}$,[15] and so P_2 needs 7 queries to obtain the messages that correspond to these codewords from the tokens.

Definition 4. *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 (the proof can be found in the full version).

Lemma 1. *For a Reed-Solomon code $[\omega, d, \delta]_q$ to be n-robust (cf. Definition 4), we must have $\omega - n(d-1) - 1 \geq 0$.*

[15] 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 $m_{i,j}$.

Accordingly, we have the following theorem (the proof can be found in the full version.)

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 [34], 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 grabler). 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 JP-Morgan 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. 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)
7. Badrinarayanan, S., Jain, A., Ostrovsky, R., Visconti, I.: Uc-secure multiparty computation from one-way functions using stateless tokens. In: ASIACRYPT (2019)
8. Baldwin, M.A.: Protein identification by mass spectrometry issues to be considered. Molecular & Cellular Proteomics **3**(1), 1–9 (2004)
9. 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)
10. 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)

11. Bitansky, N., Canetti, R.: On strong simulation and composable point obfuscation. In: CRYPTO. pp. 520–537. Springer (2010)
12. 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)
13. Broadbent, A., Gutoski, G., Stebila, D.: Quantum one-time programs. In: Annual Cryptology Conference. pp. 344–360. Springer (2013)
14. Brzuska, C., Fischlin, M., Schröder, H., Katzenbeisser, S.: Physically uncloneable functions in the universal composition framework. In: CRYPTO. pp. 51–70 (2011)
15. Canetti, R.: Universally composable security. J. ACM **67**(5), 28:1–28:94 (2020)
16. Canetti, R., Dakdouk, R.R.: Obfuscating point functions with multibit output. In: EUROCRYPT. pp. 489–508. Springer (2008)
17. Canetti, R., Kalai, Y.T., Varia, M., Wichs, D.: On symmetric encryption and point obfuscation. In: TCC. pp. 52–71. Springer (2010)
18. Chandran, N., Goyal, V., Sahai, A.: New constructions for uc secure computation using tamper-proof hardware. In: EUROCRYPT. pp. 545–562 (2008)
19. Church, G.M., Gao, Y., Kosuri, S.: Next-generation digital information storage in dna. Science p. 1226355 (2012)
20. Crick, F.H.: On protein synthesis. In: Symp Soc Exp Biol. vol. 12, p. 8 (1958)
21. 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)
22. Damgård, I., Scafuro, A.: Unconditionally secure and universally composable commitments from physical assumptions. In: ASIACRYPT (2013)
23. Döttling, N., Kraschewski, D., Müller-Quade, J.: Unconditional and composable security using a single stateful tamper-proof hardware token. In: TCC (2011)
24. Dziembowski, S., Kazana, T., Wichs, D.: One-time computable self-erasing functions. In: TCC. pp. 125–143. Springer (2011)
25. 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)
26. 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)
27. Erlich, Y., Zielinski, D.: Dna fountain enables a robust and efficient storage architecture. Science **355**(6328), 950–954 (2017)
28. Fisch, B., Freund, D., Naor, M.: Physical zero-knowledge proofs of physical properties. In: CRYPTO. pp. 313–336. Springer (2014)
29. Fisch, B.A., Freund, D., Naor, M.: Secure physical computation using disposable circuits. In: TCC. pp. 182–198. Springer (2015)
30. Garg, S., Gentry, C., Halevi, S., Raykova, M.: Two-round secure mpc from indistinguishability obfuscation. In: TCC. pp. 74–94. Springer (2014)
31. 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)
32. Glaser, A., Barak, B., Goldston, R.J.: A zero-knowledge protocol for nuclear warhead verification. Nature **510**(7506), 497–502 (2014)
33. 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)
34. Goldwasser, S., Kalai, Y., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: ACM STOC (2013)

35. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: One-time programs. In: CRYPTO. pp. 39–56. Springer (2008)
36. Goyal, R., Goyal, V.: Overcoming cryptographic impossibility results using blockchains. In: TCC. pp. 529–561. Springer (2017)
37. Goyal, V., Ishai, Y., Sahai, A., Venkatesan, R., Wadia, A.: Founding cryptography on tamper-proof hardware tokens. In: TCC. pp. 308–326 (2010)
38. 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)
39. Hazay, C., Lindell, Y.: Constructions of truly practical secure protocols using standardsmartcards. In: ACM CCS. pp. 491–500 (2008)
40. Hazay, C., Polychroniadou, A., Venkitasubramaniam, M.: Composable security in the tamper-proof hardware model under minimal complexity. In: TCC (2016)
41. Hazay, C., Polychroniadou, A., Venkitasubramaniam, M.: Constant round adaptively secure protocols in the tamper-proof hardware model. In: PKC. pp. 428–460 (2017)
42. Jain, A., Lin, H., Sahai, A.: Indistinguishability obfuscation from well-founded assumptions. In: ACM STOC. pp. 60–73. ACM (2021)
43. 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)
44. Katz, J.: Universally composable multi-party computation using tamper-proof hardware. In: EUROCRYPT. vol. 7, pp. 115–128. Springer (2007)
45. Lindell, Y.: Anonymous authentication. Journal of Privacy and Confidentiality **2**(2) (2011)
46. Lynn, B., Prabhakaran, M., Sahai, A.: Positive results and techniques for obfuscation. In: EUROCRYPT. pp. 20–39. Springer (2004)
47. MacWilliams, F.J., Sloane, N.J.A.: The theory of error correcting codes, vol. 16. Elsevier (1977)
48. Moran, T., Naor, M.: Basing cryptographic protocols on tamper-evident seals. TCC **411**(10), 1283–1310 (2010)
49. Naccache, D., Shamir, A., Stern, J.P.: How to copyright a function? In: PKC. pp. 188–196. Springer (1999)
50. Ostrovsky, R., Scafuro, A., Visconti, I., Wadia, A.: Universally composable secure computation with (malicious) physically uncloneable functions. In: EUROCRYPT. pp. 702–718. Springer (2013)
51. Pappu, R., Recht, B., Taylor, J., Gershenfeld, N.: Physical one-way functions. Science **297**(5589), 2026–2030 (2002)
52. 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)
53. Rührmair, U.: Oblivious transfer based on physical unclonable functions. In: International Conference on Trust and Trustworthy Computing. pp. 430–440 (2010)
54. Wee, H.: On obfuscating point functions. In: ACM STOC. pp. 523–532 (2005)
55. Yao, A.C.C.: How to generate and exchange secrets. In: FOCS. pp. 162–167 (1986)
56. 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)