One-Time Programs from Commodity Hardware

Harry Eldridge[1], Aarushi Goel[2], Matthew Green[1], Abhishek Jain[1], and Maximilian Zinkus[1]

[1]Johns Hopkins University, {hme, mgreen, abhishek, zinkus}@cs.jhu.edu
[2]NTT Research, aarushi.goel@ntt-research.com

Abstract

One-time programs, originally formulated by Goldwasser et al. [33], are a powerful cryptographic primitive with compelling applications. Known solutions for one-time programs, however, require specialized secure hardware that is not widely available (or, alternatively, access to blockchains and very strong cryptographic tools).

In this work we investigate the possibility of realizing one-time programs from a recent and now more commonly available hardware functionality: the *counter lockbox*. A counter lockbox is a stateful functionality that protects an encryption key under a user-specified password, and enforces a limited number of incorrect guesses. Counter lockboxes have become widely available in consumer devices and cloud platforms.

We show that counter lockboxes can be used to realize one-time programs for general functionalities. We develop a number of techniques to reduce the number of counter lockboxes required for our constructions, that may be of independent interest.

1 Introduction

One-time programs, formulated by Goldwasser et al. [33], are a flexible and powerful cryptographic primitive with compelling applications to limited-attempt authentication, fuzzy vaults, limited-query differential-private data analysis, and even autonomous ransomware and beyond. In the standard model, one-time programs are known to be impossible to realize purely in software [33, 16]. To evade this impossibility, prior works have examined the problem of building one-time programs from secure hardware tokens [33, 38], or alternatively, using blockchains [36].

The works of [33, 38] employ tamper-proof hardware that implements *one-time memory* – a simple, stateful functionality that allows anyone to read one location, after which all other locations become inaccessible. While these results are practical and work in a variety of settings, they have mainly garnered theoretical interest. The likely cause is that one-time memory tokens have not been available as a standard feature of popular personal or cloud computing platforms. While it is possible to realize these tokens using programmable smart cards or HSMs [21, 63, 41], such development typically requires expensive equipment and considerable development effort. Moreover, the few affordable platforms that support custom programming may provide weak or limited security guarantees. If portability is not required, tamper-proof hardware tokens can also be realized through virtualization: *secure enclaves* such as Intel SGX [52] and ARM TrustZone [56] offer tamper-resilience under relatively strong adversarial assumptions such as operating system (OS) compromise. Indeed, if such an enclave platform is considered trusted, it is likely easier to implement an entire one-time functionality within the enclave. However, implicit trust in an enclave provider is unacceptable in some threat models, and the soundness of this trust regardless of threat model has been repeatedly called into question [17, 24, 55]. These execution environments also typically place limitations on end-users' ability to deploy arbitrary code [43, 6, 61].

Counter lockboxes. Recently, a new generation of device- and cloud-based secure hardware has become available to end users. This includes secure co-processors that are now built into many smartphones and tablets, including the Apple Secure Enclave Processor (SEP) [3] and Google's Titan M2 [35] co-processor. It also includes specialized Hardware Security Modules (HSMs) that have recently been deployed within the data centers of consumer cloud providers; these can be accessed remotely from consumer devices to implement services such as Apple's Cloud Key Vault [49], Android Backup [58], WhatsApp backup [48], and Signal Secure Value Recovery [50]. Notably, these systems are not aimed at enterprise customers; they are configured to protect end-user cryptographic keys, even from attacks that might be launched by the device manufacturer or cloud provider themselves. These systems are now being used across billions of devices, making them more broadly accessible to consumers than any prior secure hardware platform.

Unlike secure enclave environments such as TrustZone or SGX, these consumer-oriented hardware devices do not allow end-devices to securely execute arbitrary programs. Instead, they present a limited interface to the device's application software. Since the primary purpose of these systems is to protect encryption keys under user-selected passwords, the most common interface is a functionality akin to what we describe as a *counter lockbox*.[1] To initialize a lockbox, the application software provides a password to the hardware along with a *maximum attempt limit*. At any later point, the software can retrieve the decryption key by providing the correct password. To protect the key against guessing attacks, the hardware increments a tamper-resistant counter for each incorrect guess: when this counter exceeds the maximum attempt limit, the hardware deletes the stored key. Given that this lockbox functionality has been deployed at massive scale, it represents an attractive building block for constructing more sophisticated cryptographic protocols.

Using lockboxes to construct one-time programs. The ubiquity of this basic lockbox functionality motivates us to investigate the following question: *can such a simple functionality be used to achieve general secure computation?* In this work, we answering the question in the affirmative: given access to a sufficient number of lockboxes, we show that it is possible to realize the full power of one-time programs.

This result has important practical implications: since lockboxes are increasingly available to consumer hardware, this approach provides a "backdoor" route to constructing obfuscated software, even on hardware that does not directly support this functionality. This capability facilitates many constructive applications. For example, it can be used to build sophisticated attempt-limiting authentication functionalities. A limited-attempt fuzzy vault [44] can release cryptographic secrets when a user provides an input that satisfies some complex approximate function such as biometric matching or inexact string comparison [19]. Obfuscated software also enables privacy-preserving applications such as differentially-private statistical data analysis, where query limits must be enforced to maintain a privacy budget [27]. This functionality has a dark side as well: one-time programs allow for the creation of *autonomous ransomware* [25, 13, 46], a form of malware with no command-and-control infrastructure: in this paradigm, decryption keys are revealed only when the user provides the malware with proof of payment on a public blockchain. This last concern illustrates how carefully system designers must tread when exposing secure lockbox functionality to users and developers, since as we demonstrate in this work, even this relatively weak primitive can be leveraged into powerful secure computation. The lower bounds for this transformation also raise practical concerns: system designers may wish to know *how many* instances may be safely exposed to users before the power of these constructions can be exploited.

[1] The term *counter lockbox* was previously introduced by Apple for its SEP [3]. We use it in this work to refer to a broad class of similar functionalities.

1.1 Our Results

In this work, we show that it is possible to construct secure one-time programs (OTP) using multiple instances of the counter lockbox functionality. Our main result is a construction of OTP for general functionalities based on one-way functions that requires a *constant* number of counter lockboxes per input-bit. This asymptotically matches prior constructions of one-time programs [32] in the number of hardware tokens utilized.

Theorem 1 (Informal)**.** *Assuming the existence of one-way functions, for any functionality F, there exists a construction of one-time programs in the lockbox-hybrid model that makes $\mathcal{O}(1)$ invocations to the lockbox functionality per input bit of F.*

We present our main result with counter lockboxes that allow exactly one password attempt. In practice, lockboxes may allow more attempts. For example, lockboxes may fix the maximum number of attempts to some system-wide constant (*e.g.,* 10 attempts.) To handle such cases, we demonstrate an extension of our main construction that supports lockboxes with *any* number of password attempts. The resulting scheme requires the same number of lockboxes as before.

Reducing Number of Hardware Tokens. We observe that at the cost of stronger assumptions, it is possible to achieve an asymptotic reduction in the *total* number of counter lockboxes. In particular, by using laconic oblivious transfer (LOT) [23] with malicious receiver security, we can reduce the total number of lockboxes to be *independent* of the input size and to depend only on the security parameter.

Our transformation is *generic*, and is applicable to any OTP construction (including prior known schemes). As such, this might be of independent interest.

Theorem 2 (Informal)**.** *Assuming the existence of malicious receiver laconic oblivious transfer, for any functionality F, there exists a construction of one-time programs that makes $\mathcal{O}(\lambda)$ total invocations to the lockbox functionality (where λ is the security parameter).*

LOT schemes with malicious receiver security can be generically constructed by compiling the receiver message of existing LOT schemes with succinct arguments of knowledge (SNARKs) [54, 14] either in the random oracle model, or by relying on knowledge assumptions.

Our Approach. Our starting point is the observation from the work of Goldwasser et al [33] that garbled circuits [62] are almost like one-time programs, except the seeming need of interactive oblivious transfer (OT) to transmit the wire labels corresponding to an evaluator's input. Fortunately, a one-time memory (OTM) token naturally yields the OT functionality, which paves the way for constructing one-time programs from OTM tokens.

Unlike OTMs, however, a natural use of counter lockboxes yields a "leaky" OT functionality, where the receiver is able to learn *both* sender inputs with some constant probability (we elaborate on this in Section 2). By applying standard OT combiner techniques [53, 40], the leaky OT functionality can be transformed into secure OT. However, this results in a significant overhead in the number of lockboxes required. Specifically, this approach requires $O(\lambda)$ lockboxes *per input bit* of the functionality, as opposed to $O(1)$ OTMs required in prior works.

Towards obtaining our result in Theorem 1, we observe that $O(1)$ lockboxes per input bit are sufficient to instantiate a leaky "batch" oblivious transfer functionality, where the receiver can learn both sender inputs for an a priori bounded constant fraction of the input bits. We then devise a way to construct a secure (i.e., "non-leaky") batch-OT from leaky batch-OT via *robust garbling* – a form of garbling where security holds even if the receiver learns both labels for a constant fraction of the input wires – for special functions. The secure batch-OT can then be used together with standard garbled circuits to obtain one-time programs for general functions.

Finally, we demonstrate that using laconic OT, the task of designing OTP for general functions with arbitrary input lengths can be reduced to the task of designing an OTP for functions whose input length is a fixed polynomial in the security parameter. As a result of this reduction, we are able to "compress" the effective input size, thereby achieving a reduction in the number of required hardware tokens. As we discuss later, this transformation requires an LOT scheme that achieves simulation-based security against malicious receivers.

Real World Implications. In order to assess the practical feasibility of our one-time programs, we need to consider several cost factors – number of hardware tokens required, cost of each hardware token, time to generate the OTP, and the size of software component of the OTP.

In our first construction, the main consideration is hardware. Indeed, besides the use of lockboxes to implement leaky batch OT, the rest of our construction comprises of robust garbling for special functions – an efficient, information-theoretic gadget, and regular garbled circuits. The efficiency of state-of-the-art constructions of regular garbled circuits is well-established in prior works [57]. In Section 10.1, we evaluate the concrete number of lockboxes required to implement one-time programs in practice and observe that there is a notable (albeit, constant factor) expansion from the input length to the number of total lockboxes required due to the use of binary linear error-correcting codes in our scheme. Overall, our results show that one-time programs may be practical for small to modest-sized inputs using a number of lockboxes that may be practical on today's systems or systems that will be available in the near future. Because such one-time programs may allow for destructive applications, our concrete bounds on the number of lockboxes can provide safety guidance for system developers who expose such functionalities to application developers.

Given our current understanding of LOT schemes, our second transformation is primarily of theoretical interest at the moment. We first note that recent works [37, 1] have achieved significant improvements in concrete efficiency of LOT by allowing for linear decryption times (as opposed to poly-logarithmic decryption complexity achieved in the initial works). Our transformation only requires the laconic digest property of LOT and is not sensitive to decryption complexity. As such, it can be instantiated using the state-of-the-art LOT schemes with linear decryption complexity. However, the main efficiency bottleneck stems from the fact that our transformation requires a "non-interactive" version of LOT which is obtained by evaluating the LOT sender algorithm *inside a garbled circuit*. For current LOT schemes, this translates to evaluating *public-key* operations inside a garbled circuit for every receiver input bit, which to our current understanding, is quite expensive. Our work, therefore, motivates the design of new LOT schemes (with potentially linear decryption times) with "garbling friendly" sender algorithms.

2 Technical Overview

We now describe our main ideas for constructing a one-time program using counter lockboxes. We first describe a basic construction that relies on a fairly large number of lockboxes with only one attempt allowed (denoted $A = 1$). This approach requires $O(\lambda)$ lockboxes *per bit of input* to the one-time program for security parameter λ. This construction serves as a technical warm-up and highlights the main challenges in building OTPs from counter lockboxes as opposed to one-time memory (OTM) tokens used by Goldwasser et al. [33].

We then describe our key ideas towards constructing OTPs with many fewer lockboxes, even *constant* per input bit. This asymptotically matches prior constructions based on OTM tokens. Finally, we discuss two extensions. First, we describe a generic method using laconic oblivious transfer [23] (LOT) to reduce the *total* number of lockboxes to be independent of the input size, and to depend only on λ. Second, we describe how our constructions can be extended to support counter lockboxes that allow multiple password attempts.

Initial ideas. Goldwasser et al. [33] proposed a construction of one-time programs using one-time memory (OTM) tokens. Their construction relies on the observation that garbled circuits are almost like one-time programs, except that the sender needs to interact with the receiver (via oblivious transfer) to securely hand over input wire labels for the garbled circuit corresponding to the receiver's input. This interaction can be replaced with OTMs for each input wire: the sender can embed both the 0-label and the 1-label for each wire inside an OTM, and send all the OTMs together with the garbled circuit in *one shot*. The security of OTM ensures that the receiver learns at most one label from each OTM, which it can then use to evaluate the garbled circuit.

While the above idea is intuitive, the security proof requires a bit more care due to the fact that the adversary can choose its input in an *adaptive* fashion and query the OTM tokens in an arbitrary order. In particular, the proof of security requires garbling schemes with adaptive security. Efficient solutions for such garbling schemes are known in the random oracle model [9].

In this work, we build OTPs using a different kind of hardware token, the *counter lockbox*. A natural approach is to emulate the OTM functionality using counter lockboxes. However, an immediate challenge arises. Recall that a counter lockbox protects a secret value with a pre-configured password and limited attempts; if the number of incorrect attempts reaches the threshold, the secret value is irrevocably deleted. A natural idea is to store the two wire labels for each input bit in two separate lockboxes and devise a mechanism that allows a receiver to unlock only one of the two lockboxes. This, however, seems to require revealing only one of the two passwords to the user, returning to the problem of emulating OTM.

2.1 Basic Protocol

Our first idea is to use the receiver's input bits as passwords to the lockboxes. Concretely, for each input wire, we can use 0 and 1 as the passwords for the lockboxes that hide the 0-label and 1-label, respectively. The two lockboxes for each wire are then shuffled so that the input-to-password mapping is not known to the receiver.

An honest receiver can simply use the same value to attempt to unlock both lockboxes associated with an input wire. This guarantees that they obtain their desired label from one lockbox and consumes the single attempt of the other. A malicious receiver may attempt to learn both labels by guessing the password for both of the lockboxes. This will give them only a $\frac{1}{2}$ chance of success: at least one label remains hidden with that probability. This idea can be leveraged to reduce the adversary's chances of learning both values: instead of embedding each label in a single lockbox, we "distribute" each label across additional lockboxes.

We now discuss the baseline construction of OTP that results from using lockboxes in this manner. A reader already familiar with the garbling based OTP approach may want to skip the next two paragraphs and directly go to the analysis of this baseline construction.

Generating the OTP. Let C be a Boolean circuit with input length n. The sender first garbles C to obtain a garbled circuit $\tilde{C}$ along with n pairs of wire labels $(\mathsf{label}_0^i, \mathsf{label}_1^i)$. It then performs the following steps:

1. Sample uniform bits $b_1, \ldots, b_{2\ell}$, where ℓ counts the number of lockboxes each label is distributed across.

2. For each $j = 1$ to 2ℓ: first, create an independent lockbox L_j^i using maximum attempt counter $A = 1$ and password $P = b_j$. Receive the corresponding lockbox secret K_j.

3. Next, compute $\mathsf{CT}_0^i = \mathsf{label}_0^i \oplus \bigoplus_{\forall j, b_j = 0} K_j$ and $\mathsf{CT}_1^i = \mathsf{label}_1^i \oplus \bigoplus_{\forall j, b_j = 1} K_j$.

Finally, the sender provides the receiver with the garbled circuit $\tilde{C}$ and the tuples $(\mathsf{CT}_0^1, \mathsf{CT}_1^1), \ldots, (\mathsf{CT}_0^n, \mathsf{CT}_1^n)$ as well as references to the $2\ell \cdot n$ lockboxes.

OTP evaluation. To evaluate this program on an input $x = (x_1, \ldots, x_n)$, the receiver performs the following steps for $i = 1$ to n:

1. For $j = 1$ to 2ℓ, attempt to open the lockbox L_i with password x_i to retrieve either K_j or an error (in which case, set $K_j = 0$.)

2. Compute $\mathsf{label}_{x_i}^i = \mathsf{CT}_{x_i}^i \bigoplus_{j=1}^{\lambda} K_j$.

The receiver can now evaluate $\tilde{C}$ using the labels $\mathsf{label}_{x_1}^1, \ldots, \mathsf{label}_{x_n}^n$ to obtain a circuit output.

Analysis. It is easy to verify correctness of the above construction. What remains is to show that the protocol achieves security, *i.e.,* that a malicious receiver has a negligible chance of recovering more than one label for any input wire. The argument here is simple: to recover both $(\mathsf{label}_0^i, \mathsf{label}_1^i)$ for some wire i, the attacker must query each of 2ℓ lockboxes L_j^i using exactly the right passwords. However, since the lockboxes do not reveal the password until the attempt to open is made (at which point, the lockbox either reveals the secret or destroys it), the attacker must succeed in distinguishing between the 0 and 1 lockboxes. With an optimal guessing strategy, this happens with probability $\frac{\ell!\ell! \cdot n}{2\ell!} \approx \frac{1}{2^{O(\ell)}}$. Therefore, for λ bits of security, we need $\ell = O(\lambda)$ lockboxes per-input wire.

Limitations. While a decent baseline solution, this simple approach has several limitations. First, the number of lockboxes required grows with $O(\lambda)$, which is significantly worse than the one-time program construction of [33] that requires a constant number of hardware tokens per wire. Moreover, the above solution does not support lockboxes that allow multiple password attempts, and therefore has limited applicability for real-world use. To address these limitations, in the following sections we present techniques to reduce the number of counter lockboxes required. Later, we also describe approaches for supporting lockboxes that allow multiple password attempts.

2.2 Reducing the Number of Lockboxes

Our baseline solution can be seen as implicitly building a secure *combiner* for the OTM functionality. Indeed, the secret-sharing-based approach is also used in prior works that build secure combiners for oblivious transfer (OT) (e.g. [53, 40]). It is natural to ask whether one can obtain a reduction in the number of lockboxes by using a more efficient combiner. To the best of our knowledge, however, all existing methods require an overhead of $O(\lambda)$ – the same as our baseline solution – when each component is only secure with constant probability.

We now discuss our key insights towards reducing the number of lockboxes required for one-time programs. To streamline this discussion, we start by defining an abstract "leaky" OT primitive and show how to obtain a one-time program using this primitive. Later, we discuss how counter lockboxes can be used to instantiate such a primitive and also analyse the total number of the lockboxes required for this instantiation.

Insight I: Leaky Batch-OT. Let us assume we have access to a leaky OT functionality, where the receiver can choose to specify: (1) either a choice bit b and get sender input m_b as output, (2) or a special "leakage" option. In this case, it learns both sender inputs m_0 and m_1 with some constant probability, and only one of the these inputs with the remaining probability.

This notion can be generalized to a *leaky batch-OT* functionality, where the receiver is allowed to learn both sender inputs for an a priori bounded *constant fraction* of the OTs. Furthermore, it is easy to see that multiple copies of the leaky OT functionality – one for each input bit – can realize leaky batch-OT. We ask whether it is possible to build one-time programs using leaky batch-OT, *without paying the overhead of standard OT combiners*.

At first, this seems highly unlikely. Indeed, the standard approach to one-time programs – as discussed earlier – involves the use of garbled circuits. Using leaky batch-OT would result in leakage of *both* wire labels for several input wires. The security of standard garbled circuits, however, completely breaks down if both wire labels are leaked even for a single wire (let alone multiple wires).

Insight II: Robust Garbling. We address this challenge by using a notion of *robust* garbling – one where security of the garbled function is ensured even if the receiver learns both labels for a constant fraction of the input wires. If achievable, such a tool would be clearly helpful for our task at hand. However, while intuitively appealing, it is not immediately apparent how to formally define such a notion.

With leakage, the adversary may obtain labels for multiple different inputs – inputs differing at bit locations where both wire labels were obtained. Should the adversary then be allowed to learn multiple outputs, or only a single output? Clearly the former conflicts with the one-time nature of the required functionality, and thus we would like to enforce the latter. This raises a new question: *which* output? For example, if the function is such that each input corresponds to a different output, it is not clear how we can enforce the single-output requirement in a meaningful way. Indeed, achieving our intuitive notion of robustness seems impossible for general functions. We note that previously, Almashaqbeh et al. [2], also considered a notion of robustness in garbled circuits (and more generally in non-interactive secure multiparty computation). However, given their application, they consider a slightly weaker setting, where they are able to assume an a priori fixed output for the adversary and hence do not need to deal with the above issue of "which output to reveal".[2] Since such assumptions are not applicable to our setting, we cannot rely on their definition of robustness.

We therefore, weaken our goal and attempt to define robust garbling for a restricted class of functions that have a huge number of collisions, i.e. where inputs have a certain degree of *redundancy*. If we consider functions where multiple inputs with an overlapping subset of input bits have the same output, we could hope to achieve robustness. Even if the receiver learns multiple labels for the remaining (non-overlapping) bits, it will only learn at most one unique output.

As the following example shows, however, we need to be more careful. Consider two n-bit input strings $\mathbf{x}_1$ and $\mathbf{x}_2$ that share the same first $n/2$ bits, and another input string $\mathbf{x}_3$ that shares the same last $n/2$ bits with $\mathbf{x}_2$. Toward the above intuitive description of collisions, if $\mathbf{x}_1$ and $\mathbf{x}_2$ correspond to the same output, and $\mathbf{x}_2$ and $\mathbf{x}_3$ do as well, by transitivity $\mathbf{x}_1$ and $\mathbf{x}_3$ (that do not necessarily share a significant fraction of overlapping bits) also have the same output. Without further specification, this can escalate quickly until all inputs have the same output and we end up with a constant function.

In a pursuit to capture more interesting and non-trivial functions, we specify a class of functions that take inputs of length n, with respect to a parameter γ and try to capture the idea that there is only at most one unique non-$\perp$ output associated with any $n - \gamma$ input bits. Note that this is different from saying that inputs with the same subset of $n - \gamma$ input bits have a unique output. We say that a function is *admissible* if for any $n - \gamma$ input bits, there exists *at most one unique* combination of the remaining γ bits, such that the output of this function on the combined n-bit input is a non-$\perp$ value. Moreover, if such a unique

[2] we refer the reader to Section 2.5 for a more detailed comparison with their work.

combination of the remaining γ bits exists, then it is *easy* to find them using a deterministic procedure.[3] In this work, we consider robust garbling for such admissible functions.

OTPs from Robust Garbling. Let us now assume that we have robust garbling for this restricted class of functions. We now describe how we can leverage robust garbling to build OTPs for general functions. Let F be the intended OTP functionality. Then, consider a new functionality F' such that $F'(\mathsf{enc}(\mathbf{x})) = F(\mathbf{x})$, where F' is an admissible function amenable to robust garbling and enc is some mapping function that allows us to map inputs of F to inputs of F'. Concretely, we can use an error-correcting code (ECC) as the mapping function enc that can introduce redundancy in the mapped input to help ensure that F' satisfies the above conditions of being an amenable function.

This idea can now be used to design an OTP for F as follows: (1) The sender garbles F using a regular garbling scheme. (2) For each input wire i and bit $b \in \{0, 1\}$, it defines $F'_{i,b}$ such that on input $\mathsf{enc}(\mathbf{x})$, $F'_{i,b}$ runs the ECC decoding function dec to decode $\mathbf{x}$ and then if $\mathbf{x}[i] = b$ it outputs the b-label for the i-th wire, and otherwise it outputs $\perp$. For any ECC with distance $\gamma + 1$, there is only one "valid" codeword associated with any $n - \gamma$-bit message, hence, it is easy to see that dec (and as a result $F'_{i,b}$) is an admissible function. (3) The sender garbles each $F'_{i,b}$ using robust garbling. An important point to note is that each $F'_{i,b}$ takes the same input $\mathsf{enc}(\mathbf{x})$. (4) The sender uses this observation to concatenate input labels for each $F'_{i,b}$ and embed them inside the leaky batch-OT.

Constructing Leaky OT. We now describe our idea for constructing leaky OT (and consequently leaky batch-OT). Intuitively, our leaky oblivious transfer functionality allows the receiver to obtain *both* sender inputs with some constant probability.

Our construction of leaky OT is quite natural: in fact, we use the same approach as in the base protocol discussed earlier, where the sender prepares 2ℓ lockboxes (where ℓ is some constant) and distributes the "0" and "1" message across ℓ lockboxes. As before, in order to learn both sender inputs, the adversary must correctly guess the passwords for each of the 2ℓ associated lockboxes. The adversary then succeeds with a constant probability of $\approx \frac{1}{2^{O(\ell)}}$.

For leaky batch-OT, when considering a collection of n such leaky OTs, the probability that an adversary can successfully obtain both sender inputs for a constant fraction of the OTs is $\approx \frac{1}{2^{O(n\ell)}}$. Now, observe that if n is sufficiently large (say $n = O(\lambda)$), then the probability $\approx \frac{1}{2^{O(n\ell)}}$ is negligible in λ, even if ℓ is some constant value. While this analysis is somewhat simplified, it suffices for the purposes of this discussion. More details can be found in the technical sections.

Importantly, the above insight gives us significant improvement in the required number of lockboxes. Specifically, we now only require a constant number of lockboxes per OT (or input wire). However, as discussed before, in order to implement our idea of combining leaky batch-OT with robust garbling, the length of input to this leaky batch-OT is slightly longer than our "real" input. In particular, the input to our leaky batch-OT is an ECC encoding of the receiver's input. If we use binary linear ECCs with constant rate, then the length of this codeword is $n + \gamma$ where $\gamma = O(n)$, and we need a total of $\ell \cdot (n + \gamma)$ lockboxes, which in an amortized sense is a constant number of lockboxes per n-bits.

Handling Adaptivity. We now highlight some important subtleties regarding the security definitions of leaky batch-OT and robust garbling.

In our OTP constructions, we use robust garbling in conjunction with leaky batch-OT. Specifically, the receiver obtains labels for a robust-garbled circuit from the leaky batch-OT. From our prior discussion on leaky batch-OT, it is clear that an adversary can obtain both labels for some (e.g. γ out of n) of the input wires of this robust garbling. Moreover, recall that in above construction of leaky batch OT, given

[3] The reason why we need this deterministic procedure will be explained shortly.

the entire set of lockboxes, an adversary can query them in *any* order of its choosing. In fact, it can "adaptively" decide an order based on the outcomes of previously queried lockboxes. In other words, the adversary can be "fully adaptive". Our definition of leaky batch-OT must allow for this flexibility and our robust garbling must also support this "fully adaptive" setting.

Since the adversary can potentially learn both labels for some of the inputs, for simulation, we need a way to predict the output based only on the input bits for which the adversary gets exactly one label. This is why we require that the set of admissible functions admit a deterministic procedure to predict the only (if any) valid associated output.

Finally, we remark that since the adversary can choose to ask for the *second* label of some input wires in any order, the simulator would not know until the last query which $n - \gamma$ input bits it must consider to predict the output. However, by then it might be "too late" to correctly simulate garbling. To overcome this, we make a crucial observation about our construction of leaky batch-OT from lockboxes: recall that in our construction we have 2ℓ lockboxes associated with every index $i \in [n]$. If an adversarial receiver successfully opens the relevant lockboxes and learns *one of the sender messages* (say msg_i^b) associated with that index, it is easy to predict if the adversary will also be able to learn the *other sender message* (say msg_i^{1-b}) corresponding to that index. Indeed, if the adversary made any incorrect password attempts for any of the ℓ lockboxes associated with msg_i^{1-b}, then the simulator can predict that the adversary will never be able to learn msg_i^{1-b}. However, if no incorrect password attempts were made for those ℓ lockboxes, then the adversary can be certain that the remaining (unopened) lockboxes associated with index-i have password $1 - b$ and can always successfully open them and learn msg_i^{1-b}.

Therefore, we model our definition of leaky batch-OT to require the following: whenever the adversary makes a query for a particular index, it must specify whether it plans to query the second message for this index in the future. Moreover, since we only want to allow for some bounded leakage, the number of indices for which the adversary can make this request is bounded by a parameter γ. This observation helps ensure that the simulator of robust garbling does not need to wait until the "last query" to determine which $n - \gamma$ input bits it must consider to predict the output. Instead, this can be determined once the adversary makes at least one query for each of the n indices.

Constructing Robust Garbling. We now discuss robust garbling for a sub-class of admissible functions. As discussed earlier, such a construction for a restricted function class suffices for our use in the construction of OTP. In particular, we consider admissible functions of the form $f = (\mathbf{M}, \mathbf{u}, \mathbf{z})$, where $\mathbf{M} \in \{0,1\}^{k \times n}$, $\mathbf{u} \in \{0,1\}^k$ are public and $\mathbf{z} \in \{0,1\}^k$ is private, such that on any input $\mathbf{x} \in \{0,1\}^n$,

$$f(\mathbf{x}) = \begin{cases} \mathbf{z} & \text{if } \mathbf{u} = \mathbf{M}\mathbf{x} \\ \mathbf{z}' \xleftarrow{\$} \{0,1\}^k & \text{otherwise} \end{cases}$$

While all "invalid" inputs must lead to a $\perp$ output in admissible functions, the above function instead outputs a random $\mathbf{z}'$. We note that this is not a problem in our setting (and the above function is still admissible). This is because in our OTP construction, the value $\mathbf{z}$ will correspond to labels of the garbled circuit that garbles the actual function for which we compute the OTP. In the case that the output of the above function is a random unrelated value instead of a valid label, the receiver will be able to detect this while evaluating and demarcate this output as essentially equivalent to $\perp$. We elaborate more on this in Section 6.2.

Benhamouda et al. [12] design a non-interactive *multi-party* computation (NIMPC) protocol for such functions, but where $\mathbf{M}, \mathbf{u}, \mathbf{z}$ could be matrices and vectors in any field and where each party contributes one element of $\mathbf{x}$ as input. This NIMPC protocol can be re-imagined as a robust garbling for such functionalities, when $\mathbf{M}, \mathbf{u}, \mathbf{z}$ are matrices and vectors over the Boolean field. Previously, Almashaqbeh et al. [2] leveraged a similar observation (of combining this NIMPC protocol with a regular garbled circuit) towards designing a garbling scheme that remains robust in the presence of an adversary who gets access to both labels for a fraction of the input-wires. However, there are some important differences between our definition and theirs; see Section 2.5 for a discussion).

The NIMPC protocol in [12] is presented in two phases – (1) an *offline pre-processing phase* that outputs private messages to each party and a broadcast message to all parties, and (2) an *online phase* where each party deterministically computes and broadcasts a single message based on its input and the private message output in the pre-processing phase. We observe that when working over a Boolean field, the broadcast message of the offline phase can be viewed as a garbling of the above function. Since there are only two-possible values for each element of the input vector $\mathbf{x}$, we can compute both possible messages corresponding to each element that the parties are expected to send in the online phase, and these may essentially act as the wire labels for the garbled circuit.

More concretely, this robust garbling works as follows: (1) sample a random matrix $\mathbf{s} \xleftarrow{\$} \{0,1\}^{k\times k}$ and compute $\mathbf{s'}_i = \mathbf{s}\cdot\mathbf{M}_{\cdot,i}$ for each $i \in [n]$. (2) For input wire $i \in [n]$, the 0-label $\mathsf{label}_{i,0} \in \{0,1\}^k$ is sampled randomly and the 1-label is computed as $\mathsf{label}_{i,1} = \mathsf{label}_{i,0} \oplus \mathbf{s'}_i$. (3) The garbled function is defined as $\tilde{f} = \mathbf{z} \oplus \mathbf{s}\cdot\mathbf{u} \oplus \bigoplus_{i\in[n]} \mathsf{label}_{i,0}$. To evaluate, the receiver can simply exclusive-or all the appropriate labels with $\tilde{f}$. In Section 6.2, we show this construction satisfies the above notion of robust garbling, and that if $\mathbf{x}$ satisfies $\mathbf{u} = \mathbf{M}\mathbf{x}$, then $\mathbf{z} = \tilde{f} \oplus \bigoplus_{i\in[n]} \mathsf{label}_{i,\mathbf{x}[i]}$, otherwise, this evaluation will output a random $\mathbf{z}'$.

2.3 Reducing Lockboxes using Laconic OT

We now describe a generic method for achieving an asymptotic reduction in the total number of counter lockboxes by using laconic oblivious transfer (LOT) [23]. Recall that our previous construction requires a total of $\mathcal{O}(n)$ lockboxes for n-bit inputs. Using LOT, we can reduce the number of lockboxes to be independent of the input size and only depend on the security parameter (as determined by the LOT scheme).

An LOT scheme allows a receiver to commit to a large input $x \in \{0,1\}^n$ via a short *hash* whose size is a fixed polynomial in the security parameter. Subsequently, a sender with inputs (m_0, m_1) and an index i sends a short message to the receiver. Using this message, the receiver can recover $m_{x[i]}$ but $m_{1-x[i]}$ remains computationally hidden.[4] Moreover, the hash value can be reused by the sender to transmit different messages to the receiver, based on different choices of indices i.

At a high-level, we can use LOT to "compress" the effective input size, thereby achieving an asymptotic reduction in the number of lockboxes. More specifically, let C be a circuit with n-bit inputs. We can build a one-time program for C using the following two-step approach:

1. First, we compute an adaptively secure garbled circuit $\tilde{C}$ for C together with a set of wire labels.

2. Now let Send be the next-message sender function in an LOT scheme. Let us consider n different copies $(\mathsf{Send}_1, \ldots, \mathsf{Send}_n)$ of Send, where the i-th copy is hardwired with an index $i \in [n]$ and a pair of labels $(\mathsf{lab}_i^0, \mathsf{lab}_i^1)$. Here, lab_i^b is the b-th label corresponding to the i-th input bit computed in the first step.

   Now, consider a new circuit $\mathbf{Send}$ that computes all of the functions $\mathsf{Send}_1, \ldots, \mathsf{Send}_n$ (in parallel). The input to this circuit is the LOT receiver message H – namely, the hash of an input x (to the original circuit C). We now create a one-time program $\widetilde{\mathsf{OTP}}$ for $\mathbf{Send}$ with $\mathcal{O}(|\mathsf{H}|)$ counter lockboxes using the

[4] We emphasize that LOT is non-trivial even without privacy for receivers. While receiver privacy can be generically added [23], we do not require it for our transformation.

scheme described in the previous sub-section. The final one-time program OTP for circuit C consists of $\widetilde{\mathsf{OTP}}$ *and* the garbled circuit $\tilde{C}$ computed in the first step.

To evaluate the one-time program OTP on an input x, a receiver first computes an LOT hash H of x and evaluates $\widetilde{\mathsf{OTP}}$ on input H. Using the output values, it evaluates the garbled circuit $\tilde{C}$ and returns its output.

It is easy to verify that the above construction achieves correctness. In order to prove security, we need to be able to *extract* the input of the receiver. However, from the security of $\widetilde{\mathsf{OTP}}$, we can only hope to extract the input to $\widetilde{\mathsf{OTP}}$, namely, H, which is presumably the LOT hash of some input x. In order to extract the actual x, we therefore require an LOT scheme that achieves simulation-based security against malicious receivers.

It is well known that such an LOT scheme cannot be constructed using standard black-box simulation techniques [26]. However, if we rely on random oracles or knowledge assumptions, then such a scheme can be constructed by compiling an LOT scheme with a succinct argument of knowledge (SNARK) [54, 14]. We defer further details to Appendix 8.

2.4 Counter Lockboxes with Multiple Password Attempts

Up to this point we have only considered counter lockboxes that allow for a *single* attempt to guess the password. For some real-world instantiations of counter lockboxes e.g. [49, 50], this may not be a valid assumption. We now discuss how our construction of leaky batch-OT can be adapted to support counter lockboxes that allow for *any number* of password attempts.

A natural approach is that the sender may simply "burn" all but one attempt from each lockbox they configure. However, this may be undesirable, especially in a cloud-based lockbox setting or if the sender does not wish to track the state of each lockbox. Therefore, we also provide a subtler approach described in this section and more fully examined in Appendix 9.

Let z be the number of password attempts allowed by a counter lockbox functionality. We modify the previous construction as follows: once the sender decides that a particular lockbox should be a b-lockbox for a choice bit b, they do not simply set its password to b. Instead, they create z distinct strings $\mathsf{bin}(1)\|b, \ldots, \mathsf{bin}(z)\|b$ – each ending with bit b, where $\mathsf{bin}(i)$ denotes the binary representation of i. The sender then selects one of these z at random and sets it as the password for the counter lockbox.

For any choice bit b, an honest receiver can simply generate and try all of the z potential passwords for any lockbox. This guarantees that it can open all of the required lockboxes to reconstruct the desired label for its choice bit. On the other hand, the adversary gains no new advantage from having z attempts since there are $2z$ potential password choices for any lockbox. In particular, an adversary can do no better in determining whether a lockbox is a b-lockbox than by "committing" to some b and trying b concatenated with each possible prefix string. We can therefore achieve the same parameters for the multiple password attempt case as in the single attempt case.

2.5 Related Work

Chaum and Pederson [20] were the first to propose the use of tamper-proof hardware for cryptography purposes, and Goldreich and Ostrovsky [31] explored its application to software protection. Goldwasser, Kalai and Rothblum [33] introduced the notion of one-time programs as well as one-time memory tokens. Further improvements to their construction were investigated by Goyal et al. [38] and Bellare et al. [9]. More recently, Goyal and Goyal [36] investigated the use of blockchains to construct one-time programs.

Prior to our work, Almashaqbeh et al. [2] also leveraged the techniques from [12] to achieve a form of robustness in non-interactive secure computation using garbled circuits in a different context. There are some key differences between our work and theirs: we provide a general definition of robust garbling that accounts for the challenges involved in determining the adversary's input (and output) in our setting involving "leakage". In particular, as discussed earlier, since it is unclear how to define robust garbling for general functions, we define a class of admissible functions and robust garbling for such functions (as discussed in Section 2.2). In contrast, their definitions assume an a priori fixed input (and output) for the adversary, and are not applicable to our setting. Further, our definitions (unlike theirs) account for *fully adaptive* adversaries, which is crucial to our setting where the adversary can query the lockboxes in arbitrary order.

3 Preliminaries

We include preliminary definitions and discussion for *computational indistinguishability*, the *UC-Framework*, *adaptive projective garbling schemes*, *linear error-correcting codes* and *succinct non-interactive arguments of knowledge (SNARKs)* in the supplementary material (appendices).

3.1 One-Time Programs

One-time Programs (OTP) were introduced by [33]. At a high level, a one-time program for a function f enables a party to evaluate f on any one input of its choice. The security of a one-time program dictates that no efficient adversary should be able to learn anything about the function f, beyond what can be inferred from its output $f(x)$ on any one input x of its choice.

Similar to Goyal et al. [38], we model one-time programs as a two-party non-interactive protocol that is secure against malicious receivers. In this work, we consider one-time programs that are secure against a semi-honest sender and malicious receivers. We define the ideal functionality for a one-time program in Figure 1.

Functionality $\mathcal{F}_f^{\mathsf{OTP}}$

Create: Upon receiving $(\mathsf{create}, \mathsf{sid}, P_i, P_j, x)$ from P_i where x is a string do:

1. Send $(\mathsf{create}, \mathsf{sid}, P_i, P_j)$ to P_j.
2. Store (P_i, P_j, x).

Execute: On receiving $(\mathsf{run}, \mathsf{sid}, P_i, y)$ from party P_j, find the stored tuple (P_i, P_j, x) (if no such tuple exists, do nothing.) Send $f(x, y)$ to P_j and delete tuple (P_i, P_j, x).

Figure 1: Ideal functionality for a one-time program (OTP), parameterized with a specific function f, quoted from [38].

4 Counter Lockboxes

In this section, we formalize our notion of *counter lockboxes*. A *counter lockbox*, or just "lockbox," is a stateful abstraction for securely storing cryptographic secrets such that they are protected by a humanmemorable password. To create a new lockbox, a requester provides a password P and a maximum attempt counter A. The lockbox then generates random value K and returns K to the requester. The lockbox also stores internally A, some data with which it can re-compute K given P, and some information it can use to check if a future password guess matches P.

At a later point, a requester can provide some password P' to the lockbox, which will use its internal state to check if P' produces a match. If so, the lockbox recomputes and returns K to the requester. If the password does not produce a match, the lockbox decrements A. After A incorrect guesses the lockbox completely erases its internal content, preventing the value of K from ever being retrieved.

A critical aspect of lockbox hardware, particularly in contrast with OTM tokens of [33], is that they are widely available today in commodity hardware devices and cloud platforms. Apple's Secure Enclave Processor (SEP) [3], and Google's Titan M2 [35] co-processor, which are built into many smartphones and tablets, provide such a functionality. Google and Apple's cloud backup authentication systems also implement such a functionality by providing hardware-enforced passcode attempt limits. These systems are intended to moderate access to cloud data, and delete the encryption secrets needed to retrieve the data when attempt limits are exceeded.

We model the lockbox functionality as $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$, described in Figure 2. In this work, we study cryptography in the $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$-hybrid model.

Functionality $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$

Create: On input $(\mathsf{create}, P_i, P_j, sid, id, P, A)$ from party P_i where $A > 0$, send $(\mathsf{create}, P_i, P_j, sid, id)$ to P_j. Sample $K \in \{0,1\}^\lambda$, store the tuple $(P_i, P_j, sid, id, P, A, K, 0)$, and send K to P_i

Open: On input $(\mathsf{open}, P_i, sid, id, P')$ from party P_j:

- If a tuple $(P_i, P_j, sid, id, P, A, K, N)$ does not exist, then do nothing.
- If $N = A$ then delete the tuple and return expired.
- Otherwise if $P = P'$ then delete and replace the tuple with $(P_i, P_j, sid, id, P, A, K, 0)$ and return K.
- If $P \neq P'$ then delete and replace the tuple with $(P_i, P_j, sid, id, P, A, K, N+1)$ and return bad_guess.

Figure 2: Ideal functionality for a counter lockbox. This simplified interface assumes that the lockbox "secret" K is a random string of length λ and that the password guess is directly compared to a stored password.

On the communication model. Previous works using secure hardware tokens [33] assume a two-party model in which a *sender* provisions stateful tokens and sends them to the *receiver*, who then uses them to evaluate a one-time program. This model can be directly adapted to cloud-based lockbox functionalities by simply forwarding references to the appropriate online locations. Lockboxes on a fixed device, however, may require adapted usage. For example, unlike hardware tokens, lockboxes implemented within the Apple SEP are an integral component of the device and cannot easily be removed or replaced. Hence our results can rely on the following different usage scenarios:

1. In a cloud-based scenario, the sender provisions a series of lockboxes on a shared (accessible to both parties) server, such as an Apple Cloud Key Vault [49] HSM or Google Titan [58] HSM in a remote data center. The sender then provides the location (IP address or URL) of these lockboxes along with some auxiliary data to the receiver. The receiver accesses these lockboxes to evaluate the one-time program.

2. In a device-based scenario, the sender provisions a device (such as an Apple iOS device with a SEP) with lockboxes and then physically delivers the device to the receiver. Given the physical security of the SEP [3], these lockboxes are designed to resist device forensics. Auxiliary data can be transmitted within the regular device memory, and evaluation could even be facilitated by custom on-device software such as an iOS app if deemed acceptable to the evaluator.

3. In a further device-centric instantiation, the sender and receiver may not be physically co-located. To provision lockboxes on the receiver's secure hardware, the sender employs a cryptographic protocol that enables secure message transmission to the receiver's secure hardware, while entirely bypassing the receiver's ability to observe this provisioning. For example, Apple's SEP supports a cryptographic protocol for communications between the SEP and application processor within a single device. With appropriate key management, this could be repurposed to allow a remote party to communicate securely with a receiver's SEP.

In all three settings, we assume that the hardware itself is secure against logical and physical attacks: this means that the only way to access lockbox secrets is through the password interface the hardware exposes. By contrast we assume that, at least at program execution time, the receiver has full control of the remaining portions of the device processor and can query the lockbox interface arbitrarily.

Discussion. In all prior hardware-token models, the sender physically transmits the device to the receiver and it is assumed that there is no "backward communication channel" to the sender. Indeed, such a channel can lead to privacy loss for the receiver.

However, one could consider a stronger model, where the sender does in fact have the ability to inspect lockboxes after the receiver is done querying them. In such a model, to prevent the sender from learning receiver's input bits, it is important to ensure that the following three states of lockboxes remain indistinguishable – (1) lockboxes with leftover password attempts, (2) lockboxes that were "destroyed" because of failed password attempts and (3) ones that are still presumably "functional" because they were opened using the correct password. For the first kind, we can use a simple defense and ask the receiver to consume all password attempts on each.

For the remaining two forms, our ideal lockbox functionality implicitly assumes that an adversary cannot distinguish between hardware that outputs the secret and one where the secret was destroyed because of failed password attempts. In the above stronger model, hardware that matches this ideal functionality clearly will not "leak" extra information once its attempts have been expired. It simply outputs $\perp$, and there is no way to distinguish between "expired during evaluation without producing a secret" and "did output the secret but expired later as a defensive cleanup measure." While in general, real hardware may not behave like an ideal function, our definition of this ideal functionality is inspired by precise technical specifications from vendors such as Apple (see e.g. Apple iOS Security Guide), and there seems to be strong evidence that hardware will satisfy it. As a result, our constructions remain secure in this stronger model as long as the hardware behaves similarly to the ideal functionality.

Functionality $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$

Initialize: Upon receiving $(\mathsf{init}, sid, id, \mathsf{sen}, \mathsf{rec}, \{(\mathsf{m}_{i,0}, \mathsf{m}_{i,1})\}_{i\in[n]})$ from the sender sen, where $\{(\mathsf{m}_{i,0}, \mathsf{m}_{i,1})\}_{i\in[n]} \in \mathcal{M}^{2n}$, send $(\mathsf{init}, sid, id, \mathsf{sen}, \mathsf{rec})$ to the receiver rec and store the tuple $(sid, id, \mathsf{sen}, \mathsf{rec}, \{(\mathsf{m}_{i,0}, \mathsf{m}_{i,1})\}_{i\in[n]}, \mathcal{S}_1, \mathcal{S}_2, \mathsf{counter})$, where $\mathcal{S}_1 = \mathcal{S}_2 = \emptyset$ and $\mathsf{counter} = 0$.

Open: Upon receiving $(\mathsf{open}, sid, id, \mathsf{sen}, \mathsf{rec}, i, b, \mathsf{choice})$ from party rec, where $\mathsf{choice} \in \{\mathsf{both}, \mathsf{single}\}$, find the stored tuple $(sid, id, \mathsf{sen}, \mathsf{rec}, \{(\mathsf{m}_{i,0}, \mathsf{m}_{i,1})\}_{i\in[n]}, \mathcal{S}_1, \mathcal{S}_2, \mathsf{counter})$ (if no such tuple exists, do nothing).

- If $i \in \mathcal{S}_1$, do nothing.
- Else if $i \in \mathcal{S}_2$, send $m_{i,b}$ to rec.
- Else, do the following:
  - If $\mathsf{choice} = \mathsf{single}$, send $m_{i,b}$ to rec, then delete and replace the tuple with $(sid, id, \mathsf{sen}, \mathsf{rec}, \{(\mathsf{m}_{i,0}, \mathsf{m}_{i,1})\}_{i\in[n]}, \mathcal{S}_1 \cup \{i\}, \mathcal{S}_2, \mathsf{counter})$
  - else, if choice = both and $\mathsf{counter} = \gamma$ return forbidden. Else if $\mathsf{counter} < \gamma$ send $m_{i,b}$ to rec, then delete and replace the tuple with $(sid, id, \mathsf{sen}, \mathsf{rec}, \{(\mathsf{m}_{i,0}, \mathsf{m}_{i,1})\}_{i\in[n]}, \mathcal{S}_1, \mathcal{S}_2 \cup \{i\}, \mathsf{counter} + 1)$.

Figure 3: Ideal Functionality for leaky batch-OT

5 Leaky Batch-OT

In this section, we present and formalize a notion of *leaky batch-OT* and show how it can be realised using counter lockboxes.

5.1 Definition

Leaky batch oblivious transfer is a two-party functionality between a sender and receiver, where the sender initially inputs n pairs of messages $\{(\mathsf{m}_{i,0}, \mathsf{m}_{i,1})\}_{i\in[n]}$ where each $\mathsf{m}_{i,b}$ is in some message domain $\mathcal{M}$. For each $i \in [n]$, the receiver inputs a single bit $b \in \{0, 1\}$ and obtains $\mathsf{m}_{i,b}$. Additionally, at most γ times, the receiver is allowed to input i and obtain $\mathsf{m}_{i,1-b}$, assuming they have previously received $\mathsf{m}_{i,b}$.

Our specific formulation is more nuanced. We give a formal definition of this *reactive* functionality in Figure 3. This functionality proceeds in two main steps: In the first phase, the sender *initializes* the functionality with all its inputs $\{(\mathsf{m}_{i,0}, \mathsf{m}_{i,1})\}_{i\in[n]}$. In the second phase, the receiver queries the functionality on either index-input choice tuples (i, b, choice) to obtain the corresponding message $\mathsf{m}_{i,b}$, or on single index queries i where they receive the message $\mathsf{m}_{i,1-b}$ assuming they have previously made a (i, b, both) query.

To keep track of the queries made by the receiver and to ensure that the receiver obtains both $\mathsf{m}_{i,0}$ and $\mathsf{m}_{i,1}$ for only at most γ indices $i \in [n]$, the functionality stores two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ and a counter counter. Set $\mathcal{S}_1$ contains indices for which the receiver sends $\mathsf{choice} = \mathsf{single}$ and only obtains one message. Set $\mathcal{S}_2$ contains indices for which the receiver sends $\mathsf{choice} = \mathsf{both}$ and receives both messages, and counter counts the number of indices for which the receiver has made the both choice. Each time a query is made by the receiver, the functionality checks the value of the counter, checks if the query was previously made, and updates the sets $\mathcal{S}_1$ and $\mathcal{S}_2$ accordingly. On a new both query, the functionality checks if the counter has reached γ, and if not, increments it accordingly.

5.2 Construction

In this section, we construct a protocol for leaky batch-OT using counter lockboxes. Recall that our definition of leaky batch-OT only allows the receiver to obtain both messages for at most γ indices $i \in [n]$. Therefore, we show that if ℓ is set to $\lceil -\log_2(\frac{\gamma}{n}) \rceil + 1$ then except with some negligible probability in n a malicious receiver can successfully obtain keys of all 2ℓ lockboxes for at most γ indices $i \in [n]$.

Protocol. We now give a formal description of this protocol in the $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$-hybrid model.

- **Sender:** Let $\ell := \lceil -\log_2(\frac{\gamma}{n}) \rceil + 1$. Given inputs $\{(\mathsf{m}_{i,0}, \mathsf{m}_{i,1})\}_{i\in[n]}$, the sender sen samples a fresh sid. For each $i \in [n]$, do the following:
  1. Sample a random permutation $\pi_i : [2\ell] \to [2\ell]$.
  2. Sample 2ℓ unique ids $\{id_{i,j}\}_{j\in[2\ell]}$.
  3. For each $j \in [2\ell]$,
     - If $\pi_i(j) \leq \ell$, invoke $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$ on arguments $(\mathsf{create}, \mathsf{sen}, \mathsf{rec}, sid, id_{i,\pi_i(j)}, 0, 1)$ and obtain $K_{i,0}^{\pi_i(j)}$ in return.
     - Else, invoke $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$ on arguments $(\mathsf{create}, \mathsf{sen}, \mathsf{rec}, sid, id_{i,\pi_i(j)}, 1, 1)$ and get $K_{i,1}^{\pi_i(j)}$ in return.
  4. Compute $C_{i,0} := \mathsf{m}_{i,0} \oplus \bigoplus_{j=1}^{\ell} K_{i,0}^{j}$.
  5. Compute $C_{i,1} := \mathsf{m}_{i,1} \oplus \bigoplus_{j=\ell+1}^{2\ell} K_{i,1}^{j}$.
  6. Send $\{(C_{i,0}, C_{i,1})\}_{i\in[n]}$ to the receiver rec.
- **Receiver.** Given a set of input bits $\{b_i\}_{i\in[n]}$ and upon receiving $\{(sid, id_{i,\pi_i(j)}, \mathsf{sen}, \mathsf{rec})\}_{j\in[2\ell],i\in[n]}$ from the $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$ functionalities and $\{(C_{i,0}, C_{i,1})\}_{i\in[n]}$ from the sender, the receiver proceeds as follows for each $i \in [n]$:
  1. For each $j \in [2\ell]$, invoke $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$ on arguments $(\mathsf{open}, \mathsf{sen}, sid, id_{i,\pi_i(j)}, b_i)$ to receive either $K_{i,b_i}^{\pi_i(j)}$ or $\mathsf{bad\_guess}$, in which case set $K_{i,b_i}^{\pi_i(j)} = 0$.
  2. Compute $\mathsf{m}_{i,b_i} = C_{i,b_i} \oplus \bigoplus_{j=1}^{2\ell} K_{i,b_i}^{\pi_i(j)}$.

We now prove that the above protocol securely realizes the leaky adaptive batch-OT functionality.

Theorem 3. *There exists a protocol for securely realizing the leaky batch-OT functionality $\mathcal{F}_{(n,\gamma)}^{\mathsf{OT}}$ (Figure 3) against a malicious sender and receiver, in the $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$-hybrid model, where the sender only sends a single message to receiver, while the receiver does not to send any messages to the sender.*

Proof of Security. We start by proving security against a malicious sender. To prove this, we design a simulator that can extract any malicious sender's inputs in the ideal world. This simulator emulates the lockbox functionality and whenever a malicious sender instantiates a lockbox with a password, it stores the randomly sampled secrets corresponding to passwords 0 and 1 and ignores the secrets that were sampled for any other password $\notin \{0, 1\}$. Upon receiving ciphertexts $(C_{i,0}, C_{i,1})$ for each index $i \in [n]$, the simulator XORs these ciphertexts with the all the relevant secrets associated with index i and password 0 and 1 respectively, to extract sender's input pairs. It sends these extracted inputs to the $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ functionality. It is easy to see that the joint view of the corrupt sender and the output of the honest receiver in this case will be indistinguishable from that in the real protocol.

We now show that it is also secure against a malicious receiver. For this we start by describing the simulator and later show that it can produce a transcript for the receiver in the ideal world that is indistinguishable from its transcript in the real world. We use $\mathcal{A}$ to denote the adversarial receiver. For each $b \in \{0, 1\}$, we use b-lockbox to refer to the lockboxes whose password is set to b.

- **Simulator $\to$ $\mathcal{A}$:** The simulator receives $(SID, ID, \mathsf{sen}, \mathsf{rec})$ from ideal functionality $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$. It then samples a random sid and proceeds as follows for each $i \in [n]$:

  1. For each $b \in \{0, 1\}$, sample $C_{i,b} \xleftarrow{\$} \mathcal{M}$.
  2. Sample 2ℓ unique ids $\{id_{i,j}\}_{j\in[2\ell]}$.
  3. Sample a random permutation $\pi_i : [2\ell] \to [2\ell]$.
  4. For each $j \in [2\ell]$, simulate creating a lockbox using $sid, id_{i,\pi_i(j)}$ and the appropriate passwords as described in the real protocol and simulate sending $\big(sid, id_{i,\pi_i(j)}, \mathsf{sen}, \mathsf{rec}\big)$ to $\mathcal{A}$.
  5. Send $(C_{i,0}, C_{i,1})$ to $\mathcal{A}$.

- **$\mathcal{A}$'s queries:** When $\mathcal{A}$ makes an open query to $\mathcal{F}^{\mathsf{Lockbox}}_{\lambda}$, the simulator receives those queries and simulates the open action of the corresponding lockbox as follows for each $b \in \{0, 1\}$:

  For each $i \in [n]$, for the first $\ell - 1$ b-lockboxes that the adversary opens, if the guessed password is b, sample a random $K^j_{i,b} \xleftarrow{\$} \mathcal{M}$ and send it to $\mathcal{A}$. For the ℓ^{th} b-lockbox, the simulator does the following:

  – If the adversary has previously successfully opened each of the ℓ $(1 - b)$-lockboxes for i, or has not opened all the corresponding ℓ $(1 - b)$-lockboxes, but has also not guessed an incorrect password for any of them, the simulator queries $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ on inputs $(\mathsf{open}, SID, ID, \mathsf{sen}, \mathsf{rec}, i, b, \mathsf{both})$. If it gets $\mathsf{m}_{i,b}$ in return, it sets the corresponding $K^{\ell}_{i,b} = \mathsf{m}_{i,b} \oplus \bigoplus_{j=1}^{\ell-1} K^j_{i,b}$. Else, if it gets $\mathsf{forbidden}$, the simulator outputs bad and aborts.

  – If the adversary has guessed an incorrect password for at least one of the corresponding ℓ $(1 - b)$-lockboxes, the simulator queries $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ on inputs $(\mathsf{open}, SID, ID, \mathsf{sen}, \mathsf{rec}, i, b, \mathsf{single})$. It gets $\mathsf{m}_{i,b}$ in return, and sets the corresponding $K^{\ell}_{i,b} = \mathsf{m}_{i,b} \oplus \bigoplus_{j=1}^{\ell-1} K^j_{i,b}$.

We now show that the above simulated transcript is indistinguishable from the real world. For this, we first show that if the simulator does not output bad, then the two transcripts are indistinguishable. Later, we show that the probability that the simulator outputs bad is negligible. Let BAD denote the event when the simulator outputs bad.

Lemma 1. *Assuming that the simulator $\mathcal{S}$ described above does not output* bad*, then the real world execution is indistinguishable from the ideal execution.*

Proof. The adversary's view in the real and ideal executions begins with references to $2n\ell$ lockboxes and n $C_{i,b}$ pairs. The lockbox references are created via the same process in the real and ideal variants, and hence are identically distributed. In the real execution, each $C_{i,b}$ is set to $m_{i,0} \oplus \bigoplus_{j=1}^{\ell} K^j_{i,0}$, while in the ideal execution, each $C_{i,b}$ is uniformly sampled from $\mathcal{M}$. Since each $K^j_{i,0}$ is chosen at random, each $C_{i,b}$ in the real execution is also uniformly distributed over $\mathcal{M}$.

When the adversary successfully opens lockbox $id_{i,\pi_i(j)}$ with password b, if it is not the ℓ-th lockbox associated with $C_{i,b}$, in both executions they will receive a uniform random value $K_{i,b}^{\pi_i(j)}$. When the adversary opens the ℓ'th lockbox associated with a $C_{i,b}$, because the simulator did not output bad, it was able to query $\mathcal{F}_{(n,\gamma)}^{\mathsf{OT}}$and receive $m_{i,b}$. Thus, in both executions the adversary receives $m_{i,b} \oplus \bigoplus_{j=1}^{\ell-1} K_{i,b}^j$, making the views identical.

□

We now prove that the simulator only outputs bad with some negligible probability in n.

Lemma 2. *Let $n = O(\lambda)$, then* $\Pr[\mathsf{BAD}] \leq \mathsf{negl}(\lambda)$.

Proof. Throughout this proof, we will use "lockbox set" to refer to the 2ℓ lockboxes associated with a $(C_{i,0}, C_{i,1})$ pair. For the simulator to output bad, it must have received $\mathsf{forbidden}$ as a response from $\mathcal{F}_{(n,\gamma)}^{\mathsf{OT}}$. For this to happen, it must have sent $(\mathsf{open}, sid, id, \mathsf{sen}, \mathsf{rec}, i, b, \mathsf{both})$ to $\mathcal{F}_{(n,\gamma)}^{\mathsf{OT}}$for $\gamma + 1$ different values of i. The simulator only makes a "both-query" when the adversary successfully opens the ℓ lockboxes associated with a $C_{i,b}$ without making a wrong password guess for any of the remaining ℓ lockboxes associated with password $(1-b)$. This means that the adversary must have successfully guessed the password for all the 2ℓ lockboxes associated with $\gamma + 1$ indices.

Let BOTH_i denote the event when the simulator makes a "both-query" to the ideal functionality $\mathcal{F}_{(n,\gamma)}^{\mathsf{OT}}$ for index i. We start by proving the following claim.

Claim 1. *For each $i \in [n]$, it holds that:* $\Pr[\mathsf{BOTH}_i] \leq (\frac{1}{2})^\ell \cdot \frac{\ell!}{(2\ell-1)!!}$

Proof. To successfully learn both $\mathsf{m}_{i,0}$ and $\mathsf{m}_{i,1}$, the adversary must make 2ℓ consecutive correct password guesses, knowing only the proportion of lockboxes with password 1 and lockboxes with password 0 before each guess. Because the adversary has no information other than the ratio between the 0 and 1 password lockboxes, they can do no better than picking a lockbox and guessing that its password is from the majority. This leads to the following algorithm for the adversary, which maximizes their chance of correctly guessing all 2ℓ passwords:

1. If the proportion of 0 to 1 password lockboxes is even, choose a lockbox and guess a random bit $b \overset{\$}{\leftarrow} \{0,1\}$ for its password. If the guess was incorrect, abort and guess b for the remaining lockboxes. Otherwise, continue to step 2.

2. There are now more $1-b$ than b password lockboxes, so choose a lockbox and guess $1-b$ for its password. If the guess was incorrect, abort and guess $1-b$ for the remaining lockboxes. Otherwise, if any lockboxes remain, return to step 1.

Looking only at the first step, the adversary must correctly make ℓ correct 50/50 guesses, so the probability that they correctly guess each time in step 1 is $(\frac{1}{2})^\ell$.

In step two, the number of $b-1$ password lockboxes will start at ℓ and decrease by one each iteration (as a $b-1$ password lockbox must have been successfully unlocked on one of the two the previous steps), and the total number of lockboxes will start at $2\ell - 1$, as a lockbox has been eliminated in the first step, and decrease by 2 in each iteration. This makes the probability of guessing correctly each time in step two $\frac{\ell!}{(2\ell-1)!!}$, where $n!!$ indicates the product $n \cdot (n-2) \cdot (n-4) \cdot ... \cdot 1$

Thus the total probability that the adversary guesses the password for all the lockboxes in a lockbox set is at most: $p = (\frac{1}{2})^\ell \cdot \frac{\ell!}{(2\ell-1)!!}$

□

Let $\epsilon = \frac{\gamma}{n}$, i.e. the fraction of lockbox sets for which the adversary can successfully guess each password without the simulator outputting bad.

Let x_i be a random variable that is 1 if the adversary successfully opens each lockbox in lockbox set i (which occurs with probability p), and 0 otherwise. Let $X = \sum_{i=1}^{n} x_i$ and $\mu = E[X] = np$. If $X > n\epsilon$ then the adversary has beaten the robustness threshold and the simulator outputs bad. By the Chernoff bound,

$$\Pr[X > n\epsilon] = \Pr[X > \frac{n\epsilon p}{p}] = \Pr[X > \frac{\epsilon}{p}\mu]$$
$$= \Pr[X > (1 + (\frac{\epsilon}{p} - 1))\mu] < \left(\frac{e^{(\frac{\epsilon}{p}-1)}}{(\frac{\epsilon}{p})^{\frac{\epsilon}{p}}} \right)^{np}$$

Recall that the Chernoff bound is valid when $\frac{\epsilon}{p} - 1 > 0$, i.e. $\epsilon > p$. Since $\ell = \lceil -\log_2(\frac{\gamma}{n}) \rceil + 1$,

$$p = (\frac{1}{2})^{\ell} \cdot \frac{\ell!}{(2\ell-1)!!} < (\frac{1}{2})^{\ell} = (\frac{1}{2})^{\lceil -\log_2(\frac{\gamma}{n}) \rceil + 1}$$
$$\leq (\frac{1}{2})^{-\log_2(\frac{\gamma}{n})}(\frac{1}{2}) = \frac{\gamma}{n} \cdot \frac{1}{2} = \frac{\epsilon}{2}$$

As the value in the large parentheses is strictly decreasing in $\frac{\epsilon}{p}$ when $\epsilon > p$, we have,

$$\Pr[X > n\epsilon] < \left(\frac{e^{(\frac{\epsilon}{p}-1)}}{(\frac{\epsilon}{p})^{\frac{\epsilon}{p}}} \right)^{np} \leq (\frac{e}{4})^{np}$$

Now we must show that np is $O(\lambda)$. For this, we observe that our expression for p is equivalent to $\frac{\ell!\ell!}{(2\ell)!} = \frac{1}{\binom{2\ell}{\ell}} \geq \frac{1}{2^{2\ell}}$. Plugging in our value for ℓ:

$$\frac{1}{2^{2\ell}} = (\frac{1}{2})^{2\lceil -\log_2(\frac{\gamma}{n}) \rceil + 2} \geq (\frac{1}{2})^{-2\log_2(\frac{\gamma}{n}) + 3} = \frac{\epsilon^2}{8}$$

Since ϵ (and thereby p) is a constant fraction, we can conclude that np is $O(\lambda)$, and the probability that the simulator outputs bad is negligible in $O(\lambda)$. □

6 Robust Garbling

In this section, we formalize the notion of robust garbling for a class of admissible functions. We then present a robust garbling scheme for a sub-class of such functions, with fully adaptive, information-theoretic security.

6.1 Definitions

In a robust garbling scheme, we want to capture the requirement that even if the receiver obtains both labels for some of the input wires, it should only be able to learn exactly one output. However, this poses the following conundrum: on the one hand, we are allowing the receiver to obtain labels for *multiple* inputs. On the other hand, we do not want it to learn more than *one* output. How do we reconcile these requirements?

While achieving a reconciliation seems impossible for general functions, we can hope to do so for functions where the inputs have some level of *redundancy*. In other words, if only a subset of the input bits are sufficient to determine the output of the function, we can hope to construct a garbling scheme where even if the receiver learns multiple labels for the remaining bits, it will only learn at most one uniquely defined output.

We now give a formal definition of such a class of functions.

Definition 1 (Function Class $\mathcal{F}^{n,\gamma}$). *$\mathcal{F}^{n,\gamma}$ contains all functions $f : \{0,1\}^n \to \{0,1\}^* \cup \{\bot\}$ such that for any set $\mathcal{S} \subset [n]$ of size $(n-\gamma)$ and any set of bits $\{x_i\}_{i\in\mathcal{S}}$, there exists at most one "valid" $\{x_i\}_{i\in\overline{\mathcal{S}}}$ such that $f(x_1,\ldots,x_n) \neq \bot$.*

Further, there is an an associated function $\mathsf{Expand} : \{0,1\}^{(n-\gamma)} \to \{0,1\}^n$ such that for every $\{x_i\}_{i\in\mathcal{S}}$:

1. *If $\exists\{x_i\}_{i\in\overline{\mathcal{S}}}$, such that $f(x_1,\ldots,x_n) \neq \bot$, then $\mathsf{Expand}(\{x_i\}_{i\in\mathcal{S}}) = (x_1,\ldots,x_n)$.*
2. *Else, $f\left(\mathsf{Expand}(\{x_i\}_{i\in\mathcal{S}})\right) = \bot$.*

At a high level, the above definition implies that it is possible to determine the unique output associated with any $(n-\gamma)$ bits of input.

Next, we formalize the notion of *robust garbling* for this class of functions. In addition to the robustness property discussed above, we also want this garbling scheme to be "fully adaptive". That is, upon receiving the garbled circuit, the adversary should be allowed to choose its input bit-by-bit, depending on the labels received thus far. We note that this is stronger than the standard notion of adaptivity for garbled circuits [9, 8, 29], where the adversary must specify its *entire input* in one go, after receiving the garbled circuit.

Moreover, as discussed previously, we allow the adversary to receive both labels for some of the input wires. However, in case it plans to obtain the second label for any index, it must specify that at the time of making the first query for that index. This way, once the adversary has received at least one label for each input position, the simulator can determine the output based on the ones for which the adversary is guaranteed to not make a second query and simulate accordingly. Therefore, we model our simulator for robust garbling to essentially consist of three algorithms $(\mathsf{SimFunc}, \mathsf{SimIn}, \mathsf{SimInLast})$, where $\mathsf{SimFunc}$ simulates the garbled circuit using only "public-information" about the circuit (e.g., the size of the circuit). SimIn and $\mathsf{SimInLast}$ are used for simulating the input wire labels, where $\mathsf{SimInLast}$ is used specifically once the adversary has obtained at least one label for each input wire.

We now present a definition of robust garbling.

Definition 2 (Robust Garbling). *A robust garbling scheme for functions $f \in \mathcal{F}^{n,\gamma}$ consists of a tuple of PPT algorithms $(\mathsf{RobGarble}, \mathsf{RobGarbleInp}, \mathsf{RobEval})$ such that:*

- $(\tilde{f}, \mathsf{st}) \leftarrow \mathsf{RobGarble}(1^\lambda, f)$*: This is a PPT algorithm that takes as input the security parameter 1^λ and a function $f \in \mathcal{F}^{n,\gamma}$ and outputs a garbling $\tilde{f}$ and some private state information* st.
- $\mathsf{lab}_{i,x_i} \leftarrow \mathsf{RobGarbleInp}(\mathsf{st}, i, x_i)$*: This is a PPT algorithm that takes as input the state information* st*, an index $i \in [n]$ and an input bit x_i, and outputs the corresponding input label* lab_{i,x_i}.
- $y = \mathsf{RobEval}(\tilde{f}, \{\mathsf{lab}_{i,x_i}\}_{i\in[n]})$*: Given a garbling $\tilde{f}$ and a set of labels $\{\mathsf{lab}_{i,x_i}\}_{i\in[n]}$ it outputs a value $y \in \{0,1\}^k$.*

Correctness. *For every $\lambda \in \mathbb{N}$, $f \in \mathcal{F}^{n,\gamma}$, and for each $\mathbf{x} \in \{0,1\}^n$, it holds that:*

$$\Pr[\mathsf{RobEval}(\tilde{f}, \{\mathsf{lab}_{i,x_i}\}_{i\in[n]}) = f(\mathbf{x})] = 1,$$

where $(\tilde{f}, \mathsf{st}) \leftarrow \mathsf{RobGarble}(1^\lambda, f)$ *and* $\forall i \in [n]$, $\mathsf{lab}_{i,x_i} \leftarrow \mathsf{RobGarbleInp}(\mathsf{st}, i, x_i)$.

γ-**Robust Adaptive Security.** *There exists a PPT simulator* $\mathsf{Sim} = (\mathsf{SimFunc}, \mathsf{SimIn}, \mathsf{SimInLast})$ *such that, for any non-uniform PPT adversary* $\mathcal{A}$ *there exists a negligible function* v *such that:*

$$|\Pr[\mathsf{Exp}^{\mathsf{RobAdp}}_{\mathcal{A},\mathsf{GC},\mathsf{Sim}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Exp}^{\mathsf{RobAdp}}_{\mathcal{A},\mathsf{GC},\mathsf{Sim}}(1^\lambda, 1) = 1]| \leq v(\lambda)$$

where the experiment $\mathsf{Exp}^{\mathsf{RobAdp}}_{\mathcal{A},\mathsf{GC},\mathsf{Sim}}$ *is defined as follows:*

1. *The adversary specifies a function* $f \in \mathcal{F}^{n,\gamma}$ *and obtains* $\tilde{f}$, *where* $\tilde{f}$ *is created as follows:*
   - *If* $b = 0$*:* $(\tilde{f}, \mathsf{st}) \leftarrow \mathsf{RobGarble}(1^\lambda, f)$
   - *If* $b = 1$*:* $(\tilde{f}, \mathsf{st}) \leftarrow \mathsf{SimFunc}(1^\lambda, 1^{|f|})$ [5]
2. *Initialize* $\mathcal{S}_1 = \mathcal{S}_2 = \emptyset$ *and* $\mathsf{counter} = 0$. *For each* $j \in [n+\gamma]$, *the adversary* $\mathcal{A}$ *specifies a tuple* $(i_j, x_{i_j}, \mathsf{choice}_i)$, *where* $\mathsf{choice}_i \in \{\mathsf{single}, \mathsf{both}\}$.
   - *If* $\mathsf{choice}_i = \mathsf{single}$ *and* $(i, \cdot) \notin \mathcal{S}_1$, *update* $\mathcal{S}_1 = \mathcal{S}_1 \cup \{(i, x_{i_j})\}$. *Else if* $\mathsf{choice}_i = \mathsf{both}$, $i \notin \mathcal{S}_1 \cup \mathcal{S}_2$ *and* $\mathsf{counter} < \gamma$, *update* $\mathcal{S}_2 = \mathcal{S}_2 \cup \{i\}$ *and set* $\mathsf{counter} = \mathsf{counter} + 1$. *In both cases do the following:*
     - *If* $b = 0$, *output* $\mathsf{lab}_{i_j,x_{i_j}} \leftarrow \mathsf{RobGarbleInp}(\mathsf{st}, i_j, x_{i_j})$.
     - *If* $b = 1$ *and* $|\mathcal{S}_1 \cup \mathcal{S}_2| < n$, *output* $\mathsf{lab}_{i_j,x_{i_j}} \leftarrow \mathsf{SimIn}(\mathsf{st}, i_j, x_{i_j})$.
     - *If* $b = 1$ *and* $|\mathcal{S}_1 \cup \mathcal{S}_2| = n$,
       output $\mathsf{lab}_{i_j,x_{i_j}} \leftarrow \mathsf{SimInLast}(\mathsf{st}, i_j, x_{i_j}, \mathcal{S}, \mathsf{out})$, *where* $\mathcal{S} \subset [n]$ *is the set of indices* $i \in [n]$ *such that* $(i, \cdot) \in \mathcal{S}_1$ *and* $\mathsf{out} = f(f_{\mathsf{expand}}(\{x_i\}_{i \in \mathcal{S}}))$.
   - *Else if* $\mathsf{choice}_i = \mathsf{both}$, $i \notin \mathcal{S}_1$ *and* $i \in \mathcal{S}_2$, *do the following.*
     - *If* $b = 0$, *output* $\mathsf{lab}_{i_j,x_{i_j}} \leftarrow \mathsf{RobGarbleInp}(\mathsf{st}, i_j, x_{i_j})$.
     - *If* $b = 1$, *output* $\mathsf{lab}_{i_j,x_{i_j}} \leftarrow \mathsf{SimIn}(\mathsf{st}, i_j, x_{i_j})$.

   Finally, the adversary outputs a bit b', *which is the output of the experiment.*

6.2 Construction

In this section, we present an information-theoretically secure construction of robust garbling for functions of the form $f = (\mathbf{M}, \mathbf{u}, \mathbf{z}) \in \mathcal{F}^{n,\gamma}$, where $\mathbf{M} \in \{0,1\}^{k \times n}$, $\mathbf{u} \in \{0,1\}^k$ are public and $\mathbf{z} \in \{0,1\}^k$ is private, such that on any input $\mathbf{x} \in \{0,1\}^n$,

$$f(\mathbf{x}) = \begin{cases} \mathbf{z} & \text{if } \mathbf{u} = \mathbf{M}\mathbf{x} \\ \mathbf{z}' \xleftarrow{\$} \{0,1\}^k & \text{otherwise} \end{cases}$$

We use $\mathcal{F}^{n,\gamma}_{\mathsf{linear}}$ to denote this subclass of $\mathcal{F}^{n,\gamma}$. While all *invalid* inputs must to lead to a $\perp$ output in any $f \in \mathcal{F}^{n,\gamma}$, functions in $\mathcal{F}^{n,\gamma}_{\mathsf{linear}}$ instead output a random $\mathbf{z}'$. We note that depending on the context, this may not be a problem (and the above function can still be admissible), if the receiver can distinguish a *valid* output $\mathbf{z}$ from an *invalid* random $\mathbf{z}'$ potentially using some "additional information." In our OTP construction, the value $\mathbf{z}$ will correspond to labels of the garbled circuit that garbles the actual function

[5] Here, we implicitly assume that this simulator can get any public information about f, not just its size.

for which we compute the OTP. While these labels are also random vectors in $\{0,1\}^k$, the receiver gets "additional information" in the form of the garbled circuit where $\mathbf{z}$ is used as an input wire label. In case the output of the above function is a random unrelated value instead of a valid label, while evaluating, the receiver will be able to detect this and demarcate this output as essentially equivalent to $\perp$.

Garbling scheme. We now present a construction of robust garbling scheme for the above class of functions show why it satisfies γ-robust adaptive security. As discussed previously, this is adapted from the non-interactive *multi-party* computation (NIMPC) protocol for such functions proposed by Benhamouda et al [12].

- $\mathsf{RobGarble}(1^\lambda, f)$:
  1. Sample a random $\mathbf{s} \xleftarrow{\$} \{0,1\}^{k\times k}$.
  2. For each $i \in [n]$, sample a random $\mathbf{r}_i \in \{0,1\}^k$.
  3. Set $\mathsf{st} = \mathbf{s}, \{\mathbf{r_i}\}_{i\in[n]}$.
  4. Output garbling $\tilde{f} = \mathbf{z} \oplus \mathbf{s}\cdot\mathbf{u} \oplus \bigoplus_{i\in[n]} \mathbf{r}_i$.
- $\mathsf{RobGarbleInp}(\mathsf{st}, \mathcal{I}, \{x_i\}_{i\in[n]\setminus\mathcal{I}})$:
  1. Parse $\mathsf{st} = \mathbf{s}, \{\mathbf{r_i}\}_{i\in[n]}$.
  2. For each $i \in [n]$, compute $\mathbf{s}'_i = \mathbf{s}\cdot\mathbf{M}_{\cdot,i}$, where $\mathbf{M}_{\cdot,i}$ denotes the i-th column of $\mathbf{M}$
  3. For each $i \in \mathcal{I}$, compute and output $\mathsf{lab}_{i,0} = \mathbf{r}_i$ and $\mathsf{lab}_{i,1} = \mathbf{r}_i \oplus \mathbf{s}'_i$.
  4. For each $i \in [n]\setminus\mathcal{I}$, output $\mathsf{lab}_{i,x_i} = \mathbf{r}_i \oplus \mathbf{s}'_i \cdot x_i$.
- $\mathsf{RobEval}(\tilde{f}, \{\mathsf{lab}_{i,x_i}\}_{i\in[n]})$: Compute and output $\tilde{f} \oplus \bigoplus_{i\in[n]} \mathsf{lab}_{i,x_i}$.

Theorem 4. *There exists an information-theoretically secure robust adaptive garbling scheme for each every function $f \in \mathcal{F}^{n,\gamma}_{\mathsf{linear}}$.*

6.3 Proof of Security

We start by arguing correctness of that scheme.

Correctness. We need to show that if $\mathbf{u} = \mathbf{M}\mathbf{x}$, then $\mathbf{z} = \tilde{f} \oplus \bigoplus_{i\in[n]} \mathsf{lab}_{i,x_i}$.

$$\begin{aligned}
\tilde{f} \oplus \bigoplus_{i\in[n]} \mathsf{lab}_{i,x_i} &= \mathbf{z} \oplus \mathbf{s}\cdot\mathbf{u} \oplus \bigoplus_{i\in[n]} \mathbf{r}_i \oplus \bigoplus_{i\in[n]} \mathsf{lab}_{i,x_i} \\
=&\mathbf{z} \oplus \mathbf{s}\cdot\mathbf{u} \oplus \bigoplus_{i\in[n]} \mathbf{r}_i \oplus \bigoplus_{i\in[n]} \mathbf{r}_i \oplus \mathbf{s}'_i \cdot x_i \\
=&\mathbf{z} \oplus \mathbf{s}\cdot\mathbf{u} \oplus \bigoplus_{i\in[n]} \mathbf{s}'_i \cdot x_i = \mathbf{z} \oplus \mathbf{s}\cdot\mathbf{u} \oplus \bigoplus_{i\in[n]} \mathbf{s}\cdot\mathbf{M}_{.i}\cdot x_i \\
=&\mathbf{z} \oplus \mathbf{s}\cdot(\mathbf{u} \oplus \mathbf{M}\cdot\mathbf{x}) = \mathbf{z}
\end{aligned}$$

We now show that it also satisfies γ-robust adaptive security.

γ-Robust Adaptive Security. We start by describing constructions of functions $\mathsf{SimFunc}$, SimIn and $\mathsf{SimInLast}$.

- $\mathsf{SimFunc}(1^\lambda, \mathbf{M}, \mathbf{u})$: Sample a random $\tilde{f} \xleftarrow{\$} \{0,1\}^k$ and a random $\mathbf{s} \in \{0,1\}^{k\times k}$. Set $\mathsf{st} = (\mathbf{M}, \mathbf{u}, \mathbf{s}, \mathcal{L}_0, \mathcal{L}_1)$, where $\mathcal{L}_0 = \mathcal{L}_1 = \emptyset$.
- $\mathsf{SimIn}(\mathsf{st}, i, x_i)$:
  1. Parse $\mathsf{st} = (\mathbf{M}, \mathbf{u}, \mathbf{s}, \mathcal{L}_0, \mathcal{L}_1)$.
  2. Compute lab_{i,x_i} as follows:
     - If $(i, y) \in \mathcal{L}_{x_i}$, set $\mathsf{lab}_{i,x_i} = y$.
     - Else sample a random $\mathsf{lab}_{i,0} \xleftarrow{\$} \{0,1\}^k$. Compute $\mathbf{s}'_i = \mathbf{s} \cdot \mathbf{M}_{\cdot,i}$ and $\mathsf{lab}_{i,1} = \mathbf{s}'_i \oplus \mathsf{lab}_{i,0}$. Update $\mathcal{L}_0 = \mathcal{L}_0 \cup \{(i, \mathsf{lab}_{i,0})\}$ and $\mathcal{L}_1 = \mathcal{L}_1 \cup \{(i, \mathsf{lab}_{i,1})\}$.
  3. Output $\mathsf{lab}_{i,x_i}, \mathsf{st} = (\mathbf{M}, \mathbf{u}, \mathbf{s}, \mathcal{L}_0, \mathcal{L}_1)$.
- $\mathsf{SimInLast}(\mathsf{st}, i, x_i, \mathcal{S}, \mathsf{out})$:
  1. Parse $\mathsf{st} = (\mathbf{M}, \mathbf{u}, \mathbf{s}, \mathcal{L}_0, \mathcal{L}_1)$.
  2. Compute $\mathbf{v} = \mathbf{u} \oplus \bigoplus_{j\in\mathcal{S}} \mathbf{M}_{\cdot,j} \cdot x_j$.
  3. Sample a random $\mathsf{lab}_{i,0} \xleftarrow{\$} \{0,1\}^k$. Compute $\mathbf{s}'_i = \mathbf{s} \cdot \mathbf{M}_{\cdot,i}$ and $\mathsf{lab}_{i,1} = \mathbf{s}'_i \oplus \mathsf{lab}_{i,0}$.
  4. Compute $\tilde{f}' = \mathsf{out} \oplus \mathbf{s} \cdot \mathbf{v} \oplus \bigoplus_{j\in[n]\setminus\mathcal{S}} \mathsf{lab}_{j,0} \oplus \bigoplus_{j\in\mathcal{S}} \mathsf{lab}_{j,x_j}$.
  5. If $i \in \mathcal{S}$, update $\mathsf{lab}_{i,x_i} = \mathsf{lab}_{i,x_i} \oplus \tilde{f} \oplus \tilde{f}'$. Else, update $\mathsf{lab}_{i,0} = \mathsf{lab}_{i,0} \oplus \tilde{f} \oplus \tilde{f}'$ and $\mathsf{lab}_{i,1} = \mathsf{lab}_{i,1} \oplus \tilde{f} \oplus \tilde{f}'$.
  6. Update $\mathcal{L}_0 = \mathcal{L}_0 \cup \{(i, \mathsf{lab}_{i,0})\}$ and $\mathcal{L}_1 = \mathcal{L}_1 \cup \{(i, \mathsf{lab}_{i,1})\}$.
  7. Output $\mathsf{lab}_{i,x_i}, \mathsf{st} = (\mathbf{M}, \mathbf{u}, \mathbf{s}, \mathcal{L}_0, \mathcal{L}_1)$.

Indistinguishability Argument. SimFunc outputs a random string, while RobGarble outputs a string exclusive-or random $\mathbf{r}_i$ values. Hence, it is easy to see that the outputs of RobGarble and SimFunc are identically distributed. The output of SimIn is also identically distributed to RobGarbleInp, since we set $\mathsf{lab}_{i,0}$ to be a random value and $\mathsf{lab}_{i,1}$ is set to $\mathsf{lab}_{i,0} \oplus \mathbf{s} \cdot \mathbf{M}_{\cdot,i}$. We now argue indistinguishability between the labels output by SimInLast and those output by RobGarbleInp. We know that $\mathbf{M} \cdot \mathbf{x} = \bigoplus_{j\in\mathcal{S}} \mathbf{M}_{\cdot,j} \cdot x_j \oplus \bigoplus_{j\in[n]\setminus\mathcal{S}} \mathbf{M}_{\cdot,j} \cdot x_j$. In SimInLast we compute $\mathbf{v} = \mathbf{u} \oplus \bigoplus_{j\in\mathcal{S}} \mathbf{M}_{\cdot,j} \cdot x_j$.

$$\begin{aligned}
&\mathbf{s} \cdot \mathbf{v} \oplus \bigoplus_{j\in[n]\setminus\mathcal{S}} \mathsf{lab}_{j,0} \oplus \bigoplus_{j\in\mathcal{S}} \mathsf{lab}_{j,x_j} \\
&= \mathbf{s} \cdot \mathbf{u} \oplus \mathbf{s} \cdot \mathbf{M} \cdot \mathbf{x} \oplus \bigoplus_{j\in[n]\setminus\mathcal{S}} \mathbf{s} \cdot \mathbf{M}_{\cdot,j} \cdot x_j \oplus \bigoplus_{j\in[n]\setminus\mathcal{S}} \mathsf{lab}_{j,0} \oplus \bigoplus_{j\in\mathcal{S}} \mathsf{lab}_{j,x_j} \\
&= \mathbf{s} \cdot \mathbf{u} \oplus \mathbf{s} \cdot \mathbf{M} \cdot \mathbf{x} \oplus \bigoplus_{j\in[n]} \mathbf{s} \cdot \mathbf{M}_{\cdot,j} \cdot x_j \oplus \mathsf{lab}_{j,0} = \mathbf{s} \cdot \mathbf{u} \oplus \bigoplus_{j\in[n]} \mathsf{lab}_{j,0}
\end{aligned}$$

As a result, $\tilde{f}' = \mathsf{out} \oplus \mathbf{s} \cdot \mathbf{u} \oplus \bigoplus_{j\in[n]} \mathsf{lab}_{j,0}$. It is now easy to see that since $\tilde{f}'$ is identically distributed to the garbled function output by RobGarble, it follows that SimInLast perfectly simulates the labels for the input wire queried at the end.

7 One-Time Program

In this section we use the tools built in previous sections to construct a one-time program. In addition to leaky batch-OT and robust garbling for $\mathcal{F}^{n,\gamma}_{\mathsf{linear}}$, we make use of a standard adaptive, projective garbled circuit and linear error-correcting codes over $\mathbb{F}_2$.

We instantiate our one-time program construction using a $[n, k, \gamma+1]_2$-binary linear error-correcting code, where k is the message length, n is the code-word length, and $\gamma+1$ is the distance. We give a formal description of this construction in the $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$-hybrid model. While an honest receiver does not use the "leaky" aspect of our leaky batch-OT to receive both $(\mathsf{lab}'_{j,0}, \mathsf{lab}'_{j,1})$ for any index j, a malicious receiver can certainly try to exploit it. However, since the number of "double-labels" that they can obtain is capped at γ (and our robust garbling is secure as long as double-labels for at most γ input wires are revealed), they will never receive enough to successfully obtain both labels for any input wire of the adaptive garbled circuit. As a result, even a malicious receiver will only be able to learn the output for a single input.

Protocol. We now give a formal description of the OTP protocol in the $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$-hybrid model, using $[n, k, \gamma+1]$-binary linear error-correcting codes, an adaptive projective garbled circuit $(\mathsf{AdaGarbleCkt}, \mathsf{AdaGarbleInp}, \mathsf{AdaEvalCkt})$ and a robust function garbling scheme $(\mathsf{RobGarble}, \mathsf{RobGarbleInp}, \mathsf{RobEval})$ for $\mathcal{F}^{n,\gamma}_{\mathsf{linear}}$.

- **Sender:** Given an input f, the sender sen proceeds as follows:
  1. Express f as a circuit C, then compute $(\tilde{C}, \{\mathsf{lab}_{i,b}\}_{i\in[k],b\in\{0,1\}}) \leftarrow \mathsf{AdaGarbleCkt}(1^\lambda, C)$.
  2. Instantiate a linear error-correcting code with length n, rank k, minimum distance $\gamma+1$ and generating matrix $\mathbf{G}$.
  3. For each $i \in [k]$, and each $b \in \{0,1\}$, compute a matrix $\mathbf{M}_{i,b}$ and vector $\mathbf{u}_{i,b}$ such that $\mathbf{u}_{i,b} = \mathbf{M}_{i,b} \cdot \mathbf{y}$ if and only if $\mathbf{y}$ is a valid codeword generated using $\mathbf{G}$ and its corresponding word has bit b at position i, i.e. $\mathbf{u}_{i,b} = \mathbf{M}_{i,b} \cdot \mathbf{y} \iff \exists \mathbf{x} \in \{0,1\}^k, \mathbf{y}^\mathsf{T} = \mathbf{x}^\mathsf{T} \cdot G \wedge \mathbf{x}_i = b$. Then, define the following function:
  $$F_{i,b}(\mathbf{y}) = \begin{cases} \mathsf{lab}_{i,b} & \text{if } \mathbf{u}_{i,b} = \mathbf{M}_{i,b} \cdot \mathbf{y} \\ \mathbf{z}' \xleftarrow{\$} \{0,1\}^k & \text{otherwise} \end{cases}$$
  Next, compute $(\tilde{F}_{i,b}, \{\mathsf{robustLab}^{i,b}_{j,b'}\}_{j\in[n],b'\in\{0,1\}}) \leftarrow \mathsf{RobGarble}(1^\lambda, F_{i,b})$.
  4. Define $\overrightarrow{\mathsf{robustLab}}_{j,b'} := \{\mathsf{robustLab}^{i,b}_{j,b'}\}_{i\in[k],b\in\{0,1\}}$ for all $j \in [n], b' \in \{0,1\}$.
  5. Sample a fresh sid and id and invoke $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ on arguments $(\mathsf{init}, sid, id, \mathsf{send}, \mathsf{rec},$ $\{(\overrightarrow{\mathsf{robustLab}}_{j,0}, \overrightarrow{\mathsf{robustLab}}_{j,1})\}_{j\in[n],b'\in\{0,1\}})$
  6. Send $(\tilde{C}, \{\tilde{F}_{i,b}\}_{i\in[k],b\in\{0,1\}})$ to the receiver rec.

- **Receiver:** Given an input $\mathbf{x}$ and upon receiving $(sid, id, \mathsf{sen}, \mathsf{rec})$ from $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ and $(\tilde{C}, \{\tilde{F}_{i,b}\}_{i\in[k],b\in\{0,1\}})$ from the sender, the receiver proceeds as follows:
  1. Compute $\mathbf{y} := \mathbf{x}^\mathsf{T} \cdot \mathbf{G}$.
  2. For each $j \in [n]$, invoke $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ on arguments $(\mathsf{open}, sid, id, \mathsf{sen}, \mathsf{rec}, j, \mathbf{y}[j])$ and get $\overrightarrow{\mathsf{robustLab}}_{j,\mathbf{y}[j]} = \{\mathsf{robustLab}^{i,b}_{j,\mathbf{y}[j]}\}_{i\in[k],b\in\{0,1\}}$ in return.
  3. For each $i \in [k]$, compute $\mathsf{lab}_{i,x_i} = \mathsf{RobEval}(\tilde{F}_{i,\mathbf{x}[i]}, \mathsf{robustLab}^{i,\mathbf{x}[i]}_{j,\mathbf{y}[j]}\}_{j\in[n]})$.

4. Compute and output $z \leftarrow \mathsf{AdaEvalCkt}(\tilde{C}, \{\mathsf{lab}_{i,\mathbf{x}[i]}\}_{i\in[k]})$.

Note that for all $i \in [k]$ and $b \in \{0,1\}$, the function $F_{i,b}$ belongs to the $\mathcal{F}^{n,\gamma}_{\mathsf{linear}}$ class of functions described in Section 2.2. It is easy to identify when the output is $\bot$, as the output of each function is an input wire label for a garbled circuit. The use of an error-correcting code grants the properties required by $\mathcal{F}^{n,\gamma}_{\mathsf{linear}}$. By the definition of minimum distance, any set of $(n-\gamma)$ fixed bits will define only a single valid codeword, and the Expand function is simply a lookup for the codeword uniquely defined by those bits. Finally, each $F_{i,b}$ clearly meets the linear construction requirement of $\mathcal{F}^{n,\gamma}_{\mathsf{linear}}$.

Theorem 5. *Assuming the existence of one-way functions, there exists a non-interactive protocol for securely realizing $\mathcal{F}^{\mathsf{OTP}}_{f}$ against a semi-honest sender and malicious receiver in the $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$-hybrid model.*

We now prove that the above protocol securely realizes the one-time program functionality in the $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$-hybrid model. Since the sender does not receive any messages in the above protocol, it is trivially secure against a semi-honest sender. We now show that it is also secure against a corrupt receiver. For this we start by describing the simulator and later show that it can produce a transcript for the receiver in the ideal world, that is indistinguishable from its transcript in the real world.

Let $(\mathsf{SimC}, \mathsf{SimCIn})$ and $(\mathsf{SimFunc}, \mathsf{SimIn}, \mathsf{SimInLast})$ be the simulators for the adaptive and robust garbling schemes respectively. We assume that $\mathbf{G}$ and $\{(\mathbf{M}_{i,b}, \mathbf{u}_{i,b})\}_{i\in[n],b\in\{0,1\}}$ are public parameters.

- **Simulator $\rightarrow \mathcal{A}$:** The simulator receives $(SID, ID, \mathsf{sen}, \mathsf{rec})$ from ideal functionality $\mathcal{F}^{\mathsf{OTP}}_{f}$ and proceeds as follows:
  - Compute $(\tilde{C}, \mathsf{st}) \leftarrow \mathsf{SimC}(1^\lambda, 1^{|C|})$.
  - For each $i \in [k]$ and $b \in \{0,1\}$,
compute $(\tilde{F}_{i,b}, \mathsf{st}'_{i,b}) \leftarrow \mathsf{SimFunc}(1^\lambda, \mathbf{M}_{i,b}, \mathbf{u}_{i,b})$.
  - Sample a random sid and id and simulate initiating the $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ by sending $(sid, id, \mathsf{sen}, \mathsf{rec})$ to $\mathcal{A}$.
  - Send $(\tilde{C}, \{\tilde{F}_{i,b}\}_{i\in[k],b\in\{0,1\}})$ to $\mathcal{A}$.

- **$\mathcal{A}$'s queries:** The simulator then simulates the $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ functionality by initializing sets $\mathcal{S}_1 = \mathcal{S}_2 = \emptyset$ and counter $\mathsf{counter} = 0$. When $\mathcal{A}$ makes a query to $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ with arguments $(\mathsf{open}, sid, id, \mathsf{sen}, \mathsf{rec}, j, y_j, \mathsf{choice}_j)$, the simulator receives that query and simulates the open action as follows:
  - If $(j, \cdot) \in \mathcal{S}_1$, do nothing.
  - Else if $\mathsf{choice}_j = \mathsf{both}$, and $\mathsf{counter} = \gamma$, return $\mathsf{forbidden}$.
  - Else if $\mathsf{choice}_j = \mathsf{single}$ and $(j, \cdot) \notin \mathcal{S}_1$, update $\mathcal{S}_1 = \mathcal{S}_1 \cup \{(j, y_j)\}$. Or if $\mathsf{choice}_j = \mathsf{both}$, $i \notin (\mathcal{S}_1 \cup \mathcal{S}_2)$ and $\mathsf{counter} < \gamma$, update $\mathcal{S}_2 = \mathcal{S}_2 \cup \{j\}$ and set $\mathsf{counter} = \mathsf{counter} + 1$. In both cases do the following:
    - ⋆ If $|\mathcal{S}_1 \cup \mathcal{S}_2| < n$, for each $i \in [k], b \in \{0,1\}$ compute $\mathsf{robustLab}^{i,b}_{j,y_j} \leftarrow \mathsf{SimIn}(\mathsf{st}'_{i,b}, j, y_j)$, and send $\overrightarrow{\mathsf{robustLab}}_{j,y_j} = \{\mathsf{robustLab}^{i,b}_{j,y_j}\}_{i\in[k],b\in\{0,1\}}$ to $\mathcal{A}$.
    - ⋆ Else,
      - · Compute $\mathbf{y} = f(\mathsf{Expand}(\{y_i\}_{i\in\mathcal{S}}))$, where $\mathcal{S} \subset [n]$ is the set of indices $j \in [n]$ such that $(j, \cdot) \in \mathcal{S}_1$. Then use the decoding algorithm of the error-correcting code to decode $\mathbf{y}$ and get $\mathbf{x}$.

  - · Query $\mathcal{F}_f^{\mathsf{OTP}}$ on inputs $(\mathsf{run}, SID, ID, \mathbf{x})$ and get z in return. Then compute $\{\mathsf{lab}_{i,\mathbf{x}_i}\}_{i\in[k]} \leftarrow \mathsf{SimCIn}(\mathsf{st}, z)$.
  - · For each $i \in [k], b \in \{0,1\}$, compute $\mathsf{robustLab}^{i,b}_{j,y_j} \leftarrow \mathsf{SimInLast}(\mathsf{st}'_{i,b}, j, y_j, \mathcal{S}, \mathsf{lab}_{i,b})$ and send $\overrightarrow{\mathsf{robustLab}}_{j,y_j} = \{\mathsf{robustLab}^{i,b}_{j,y_j}\}_{i\in[k],b\in\{0,1\}}$ to $\mathcal{A}$.

- Else if $\mathsf{choice}_j = \mathsf{both}$ and $j \in \mathcal{S}_2$, for each $i \in [k], b \in \{0,1\}$ compute $\mathsf{robustLab}^{i,b}_{j,y_j} \leftarrow \mathsf{SimIn}(\mathsf{st}'_{i,b}, j, y_j)$, and send $\overrightarrow{\mathsf{robustLab}}_{j,y_j} = \{\mathsf{robustLab}^{i,b}_{j,y_j}\}_{i\in[k],b\in\{0,1\}}$ to $\mathcal{A}$. If $\mathsf{choice} = \mathsf{single}$, update $\mathcal{S}_1 = \mathcal{S}_1 \cup \{(j, y_j)\}$.

Lemma 3. $\mathsf{IDEAL}^{\mathcal{F}_f^{\mathsf{OTP}}}_{S,\mathcal{Z}} \approx \mathsf{REAL}_{\Pi^{\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}},\mathcal{A},\mathcal{Z}}$

Proof. We show that the transcript for the real and ideal world are statistically close by considering a sequence of hybrids

- **Hybrid** $\mathcal{H}_0$ In this experiment the simulator internally simulates a real execution of the protocol between sen and $\mathcal{A}$ (on their actual inputs), internally simulating the $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ while maintaining the sets $\mathcal{S}_1$ and $\mathcal{S}_2$, and outputs whatever the simulated $\mathcal{A}$ outputs. This is clearly identical to the the REAL scenario.

- **Hybrid** $\mathcal{H}_1$ In this experiment, rather than computing each $\overrightarrow{\mathsf{robustLab}}$ before sending its first messages to $\mathcal{A}$, the simulator instead computes the necessary labels after $\mathcal{A}$ submits its queries. When the simulator needs to respond with $\overrightarrow{\mathsf{robustLab}}_{j,b'}$ to one of $\mathcal{A}$'s queries, it sets $\overrightarrow{\mathsf{robustLab}}_{j,b'} := \{\mathsf{robustLab}^{i,b}_{j,b'}\}_{i\in[k],b\in\{0,1\}}$ and returns it to $\mathcal{A}$

  The values that $\mathcal{A}$ receives in response to its queries are identical to those that it receives in $\mathcal{H}_0$, the only difference being when the simulator calculates them. Therefore the distribution produced by this experiment is identical to $\mathcal{H}_0$

- **Hybrid** $\mathcal{H}_2$ This hybrid is similar to the previous one except that the simulator simulates all the robust garblings instead of computing them honestly. In particular, when $\mathcal{A}$ makes a query to $\mathcal{F}^{\mathsf{OT}}_{(n,\gamma)}$ with arguments $(\mathsf{open}, sid, id, \mathsf{sen}, \mathsf{rec}, j, y_j, \mathsf{choice}_j)$, the simulator receives that query and simulates the open action as follows:

  - If $(j, \cdot) \in \mathcal{S}_1$, do nothing.
  - Else if $\mathsf{choice}_j = \mathsf{both}$, and $\mathsf{counter} = \gamma$, return $\mathsf{forbidden}$.
  - Else if $\mathsf{choice}_j = \mathsf{single}$ and $(j, \cdot) \notin \mathcal{S}_1$, update $\mathcal{S}_1 = \mathcal{S}_1 \cup \{(j, y_j)\}$. Or if $\mathsf{choice}_j = \mathsf{both}$, $i \notin (\mathcal{S}_1 \cup \mathcal{S}_2)$ and $\mathsf{counter} < \gamma$, update $\mathcal{S}_2 = \mathcal{S}_2 \cup \{j\}$ and set $\mathsf{counter} = \mathsf{counter} + 1$. In both cases do the following:
    - ⁎ If $|\mathcal{S}_1 \cup \mathcal{S}_2| < n$, for each $i \in [n], b \in \{0,1\}$ compute $\mathsf{robustLab}^{i,b}_{j,y_j} \leftarrow \mathsf{SimIn}(\mathsf{st}'_{i,b}, j, y_j)$, and send $\overrightarrow{\mathsf{robustLab}}_{j,y_j} = \{\mathsf{robustLab}^{i,b}_{j,y_j}\}_{i\in[k],b\in\{0,1\}}$ to $\mathcal{A}$.
    - ⁎ Else, Compute $\mathbf{y} = f(\mathsf{Expand}(\{y_i\}_{i\in\mathcal{S}}))$, where $\mathcal{S} \subset [n]$ is the set of indices $j \in [n]$ such that $(j, \cdot) \in \mathcal{S}_1$. Then use the decoding algorithm of the error-correcting code to decode $\mathbf{y}$ and get $\mathbf{x}$. Then compute $\{\mathsf{lab}_{i,b}\}_{i\in[k],b\in\{0,1\}} \leftarrow \mathsf{AdaGarbleInp}(\mathsf{st}, \mathbf{x})$. For each $i \in [k], b \in \{0,1\}$, compute $\{\mathsf{robustLab}^{i,b}_{j,y_j}\}_{i\in[k],b\in\{0,1\}} \leftarrow \mathsf{SimInLast}(\mathsf{st}'_{i,b}, j, y_j, \mathcal{S}, \mathsf{lab}_{i,b})$ and send $\overrightarrow{\mathsf{robustLab}}_{j,y_j} = \{\mathsf{robustLab}^{i,b}_{j,y_j}\}_{i\in[k],b\in\{0,1\}}$ to $\mathcal{A}$.

- Else if $\mathsf{choice}_j = \mathsf{both}$ and $j \in \mathcal{S}_2$, for each $i \in [k], b \in \{0,1\}$ compute $\mathsf{robustLab}^{i,b}_{j,y_j} \leftarrow \mathsf{SimIn}(\mathsf{st}'_{i,b}, j, y_j)$, and send $\overrightarrow{\mathsf{robustLab}}_{j,y_j} = \{\mathsf{robustLab}^{i,b}_{j,y_j}\}_{i\in[k],b\in\{0,1\}}$ to $\mathcal{A}$. If $\mathsf{choice} = \mathsf{single}$, update $\mathcal{S}_1 = \mathcal{S}_1 \cup \{(j, y_j)\}$.

  Indistinguishability between this hybrid and the previous one follows via a sequence of $2k$ sub-hybrids, where we simulate one additional $F_{i,b}$ at a time. Indistinguishability between these sub-hybrids follows from γ-adaptive robustness of robust garbling.

- **Hybrid** $\mathcal{H}_3$ This hybrid is identical to the ideal execution.

  Indistinguishability between this hybrid and the previous one follows from adaptive security of the garbling scheme. By transitivity of computational indistinguishability, it follows that the real world execution is indistinguishable from the ideal world execution.

□

8 Reducing The Number of Lockboxes Using Laconic OT

In this section, we present our construction for further reducing the total number of lockboxes to be independent of the input size, by relying on stronger cryptographic assumptions.

8.1 Laconic Oblivious Transfer

In this section, we define Laconic OT [23]. In our setting, since the receiver never sends a message to the sender, receiver privacy is implicitly guaranteed and hence we do not require receiver-private laconic OT. We define two notions of sender privacy – one against semi-honest receivers and the other one against malicious receivers. In the next section, we will show how to construct a laconic OT scheme that is secure against malicious receivers using SNARKs and existing laconic OT that is secure against semi-honest receivers.

Definition 3 (Laconic OT). *A laconic OT (ℓOT) syntactically consists of four algorithms* crsGen, Hash, Send *and* $\mathsf{Receive}$.

- $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$. *It takes as input the security parameter 1^λ and outputs a common reference string* crs.
- $(\mathsf{digest}, \hat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$. *This is a deterministic function that takes as input a common reference string* crs *and a database $D \in \{0,1\}^*$ and outputs a digest* digest *of the database and a state $\hat{D}$.*
- $\mathsf{e} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$. *It takes as input a common reference string* crs*, a digest* digest*, a database location $L \in \mathbb{N}$, and two messages m_0 m_1 of length λ, and outputs a ciphertext* e.
- $m \leftarrow \mathsf{Receive}^{\hat{D}}(\mathsf{crs}, \mathsf{e}, L)$. *This is a RAM algorithm with random read access to $\hat{D}$. It takes as input a common reference string* crs*, a ciphertext* e*, and a database location $L \in \mathbb{N}$. It outputs a message m*

We require the following properties of an ℓOT scheme $(\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive})$.

- ***Correctness:*** *We require that it holds for any database D of size at most $M = \mathsf{poly}(\lambda)$ for any polynomial function $\mathsf{poly}(\cdot)$, any memory location $L \in [M]$, and any pair of messages $(m_0, m_1) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$ that*
$$\Pr\left[m = m_{D[L]} \middle| \substack{\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda) \\ (\mathsf{digest}, \hat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D) \\ \mathsf{e} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1) \\ m \leftarrow \mathsf{Receive}^{\hat{D}}(\mathsf{crs}, \mathsf{e}, L)} \right] = 1$$
where the probability is taken over the random choices made by crsGen *and* Send.

- ***Sender Privacy Against Semi-Honest Receivers:*** *There exists a PPT simulator* $\ell\mathsf{OTSim}$ *such that the following holds. For any database D of size at most $M = \mathsf{poly}(\lambda)$ for any polynomial function $poly(\cdot)$, any memory location $L \in [M]$, and any pair of messages $(m_0, m_1) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$, let* $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$ *and* $\mathsf{digest} \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$. *Then it holds that*
$$(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)) \approx (\mathsf{crs}, \ell\mathsf{OTSim}(D, L, m_{D[L]}))$$

- ***Sender Security Against Malicious Receivers:*** *For any PPT adversary $\mathcal{A}$, there exists a PPT simulator* $\ell\mathsf{OTSim} = (\ell\mathsf{OTSim}_1, \ell\mathsf{OTSim}_2)$, *such that for all auxiliary inputs $z \in \{0,1\}^{\mathsf{poly}(k)}$ and for any pair of messages $(m_0, m_1) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$ and any memory location $L \in [M]$, we have that*
$$\left| \Pr\left[\mathsf{IDEAL}^{\mathcal{F}_{\ell OT}}_{\ell\mathsf{OTSim}}(1^\lambda, L, m_0, m_1) = 1 \right] - \Pr\left[\mathsf{REAL}_{\mathcal{A}, \ell OT}(1^\lambda, L, m_0, m_1) = 1 \right] \right| \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$
Where experiments $\mathsf{IDEAL}^{\mathcal{F}_{\ell OT}}_{\ell\mathsf{OTSim}}$ *and* $\mathsf{REAL}_{\mathcal{A}, \ell OT}$ *are defined as follows:*

Exp $\mathsf{REAL}_{\mathcal{A}, \ell OT}(1^\lambda, L, m_0, m_1)$:	**Exp** $\mathsf{IDEAL}^{\mathcal{F}_{\ell OT}}_{\ell\mathsf{OTSim}}(1^\lambda, L, m_0, m_1)$:
– $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$	– $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$
– $\mathsf{digest} \leftarrow \mathcal{A}(\mathsf{crs})$	– $\mathsf{digest} \leftarrow \mathcal{A}(\mathsf{crs}, z)$
	– $D \leftarrow \ell\mathsf{OTSim}_1(\mathsf{crs}, z)$
– $\mathsf{e} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$	– $e \leftarrow \ell\mathsf{OTSim}_2(\mathsf{crs}, z, \mathsf{digest}, D, L, m_{D[L]})$
– Output $\mathcal{A}(\mathsf{e})$	– Output $\mathcal{A}(\mathsf{e})$

- ***Efficiency Requirement:*** *The length of* digest *is a fixed polynomial in λ independent of the size of the database. Moreover, the algorithm* Hash *runs in time* $|D| \cdot \mathsf{poly}(log|D|, \lambda)$, Send *and* Receive *run in time* $\mathsf{poly}(log|D|, \lambda)$.

8.2 Constructing Maliciously Secure Laconic OT

In this section, we present a construction of maliciously secure laconic OT using SNARKs and laconic OT that is secure against semi-honest receivers. The high level idea is to have the laconic OT receiver to compute a digest of its database using the Hash algorithm of the underlying semi-honest laconic OT and then attach a SNARK proof along with it to prove that the digest was honestly computed. Achieving malicious security by attaching proofs in this manner, typically requires starting from a scheme that is semi-maliciously secure [7]. However, since the Hash algorithm of the semi-honest laconic OT scheme [23] is deterministic, it is sufficient for us to start with a semi-honest laconic OT.

Theorem 6. *Let* $(\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive})$ *be a laconic OT scheme that is secure against semi-honest receivers. Let* $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Ver})$ *be a SNARK with adaptive proof of knowledge for relation* $\mathcal{R}_{\mathsf{Hash}}$ *that contains instances of the form* $y = (M_{\mathsf{Hash}}, x, t)$*, where* M_{Hash} *is the random access machine that takes inputs* (x, w)*, where values* w *are of the form* (crs, D)*, and accepts iff* $(x, \cdot) = \mathsf{Hash}(\mathsf{crs}, D)$ *in time* $t \leq T_{\mathsf{Hash}}$*. Then there exists a laconic OT scheme that is secure against malicious receivers.*

Construction. We now give a formal description of a Laconic OT $(\mathsf{crsGen}', \mathsf{Hash}', \mathsf{Send}', \mathsf{Receive}')$ that is secure against malicious receivers.

- $\mathsf{crsGen}'(1^\lambda)$: Compute $(\mathsf{prs}, \mathsf{vst}) \leftarrow \mathsf{Gen}(1^\lambda)$ and $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$ and output $\mathsf{crs}' = (\mathsf{crs}, (\mathsf{prs}, \mathsf{vst}))$
- $\mathsf{Hash}'(\mathsf{crs}', D \in \{0,1\}^{2\lambda})$:
  - Parse $\mathsf{crs}' = (\mathsf{crs}, (\mathsf{prs}, \mathsf{vst}))$
  - $(\mathsf{digest}, \hat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
  - $\pi \leftarrow \mathsf{Prove}(\mathsf{prs}, (\mathsf{M}, \mathsf{digest}, \mathsf{t}), \mathsf{D})$
  - $\mathsf{digest}' \leftarrow (\mathsf{digest}, \pi)$
  - Output $(\mathsf{digest}', \hat{D})$
- $\mathsf{Send}(\mathsf{crs}', \mathsf{digest}', L, m_0, m_1)$:
  - Parse $\mathsf{crs}' = (\mathsf{crs}, (\mathsf{prs}, \mathsf{vst}))$
  - Parse $\mathsf{digest}' = (\mathsf{digest}, \pi)$
  - $b \leftarrow \mathsf{Ver}(\mathsf{vst}, \mathsf{digest}, \pi)$
  - If $b = 0$ output $\perp$, else output $e = \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$
- $\mathsf{Receive}'^{\hat{D}}(\mathsf{crs}', \mathsf{e}, L)$:
  - Parse $\mathsf{crs}' = (\mathsf{crs}, (\mathsf{prs}, \mathsf{vst}))$
  - If $\mathsf{e} = \perp$ output $\perp$, else output $m \leftarrow \mathsf{Receive}^{\hat{D}}(\mathsf{crs}, \mathsf{e}, L)$

It is clear to see that if the original laconic OT scheme is correct then this construction is as well. Additionally, the efficiency requirements are maintained by the succinctness of the SNARK.

Security. The simulator $\ell\mathsf{OTSim}$ first uses the extractor $\mathcal{E}_{\mathcal{A}}$ for the underlying SNARK to extract a database D. It then checks if $1 =^? = \mathsf{Ver}(\mathsf{vst}, \mathsf{digest}, \pi)$ and $(\mathsf{digest}, \cdot) = \mathsf{Hash}(\mathsf{crs}, D)$. If either of these checks fail, the simulator outputs $\perp$. Else, the simulator queries the laconic OT ideal functionality $\mathcal{F}_{\ell OT}$ using D to obtain message $m_{D[L]}$. Finally, it invokes the simulator of the underlying semi-honest laconic OT scheme using $m_{D[L]}$ to simulate the sender message e.

To argue indistinguishability between the Real and Ideal experiments, we consider the following cases:

- $0 = \mathsf{Ver}(\mathsf{vst}, \mathsf{digest}, \pi)$: In this case, both the simulator and honest sender output $e = \perp$.
- $1 = \mathsf{Ver}(\mathsf{vst}, \mathsf{digest}, \pi)$ and $(\mathsf{digest}, \cdot) \neq \mathsf{Hash}(\mathsf{crs}, D)$: In this case, an honest sender will output a non-$\perp$ e, while the simulator outputs $e = \perp$. However, from adaptive proof of knowledge property of the underlying SNARK scheme, we know that the probability of this case happening is negligible.
- $1 = \mathsf{Ver}(\mathsf{vst}, \mathsf{digest}, \pi)$ and $(\mathsf{digest}, \cdot) = \mathsf{Hash}(\mathsf{crs}, D)$: Indistinguishability in this case follows from sender privacy against semi-honest receivers of the underlying semi-honest laconic OT scheme.

8.3 New One-Time Program Construction

In this section, we present our new construction of one-time program, where the total number of lockboxes are independent of the input length and only depend on the security parameter. As discussed in the technical overview of our paper, this construction makes use of laconic OT secure against malicious receivers, the original OTP scheme from our paper and adaptive garbled circuit.

Protocol. We now give a formal description of the OTP protocol in the $\mathcal{F}_f^{\mathsf{OTP}}$-hybrid model, using an adaptive projective circuit garbling scheme $(\mathsf{AdaGarbleCkt}, \mathsf{AdaGarbleInp}, \mathsf{AdaEvalCkt})$, and a malicious-receiver secure laconic OT scheme $(\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive})$. Let $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$.

- **Sender.** Given an input (crs, f), the sender sen proceeds as follows:
  1. Express f as a circuit C, then compute $(\tilde{C}, \{\mathsf{lab}_{i,b}\}_{i\in[k],b\in\{0,1\}}) \leftarrow \mathsf{AdaGarbleCkt}(1^\lambda, C)$
  2. Define a function f' that on input digest does the following:
     - For $i \in [k]$ compute $\mathsf{e}_i \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, i, \mathsf{lab}_{i,0}, \mathsf{lab}_{i,1})$
     - return $\{\mathsf{e}_i\}_{i\in[k]}$
  3. Sample a fresh sid and invoke $\mathcal{F}_{f'}^{\mathsf{OTP}}$ on parameters $(\mathsf{create}, \mathsf{sid}, \mathsf{sen}, \mathsf{rec})$
  4. Send $\tilde{C}$ to rec
- **Receiver.** Given an input $(\mathsf{crs}, \mathbf{x})$ and upon receiving $(\mathsf{create}, \mathsf{sid}, \mathsf{sen}, \mathsf{rec})$ from $\mathcal{F}_{f'}^{\mathsf{OTP}}$ and $\tilde{C}$ from the sender, the receiver proceeds as follows:
  1. Compute $(\mathsf{digest}, \hat{\mathbf{x}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, \mathbf{x})$
  2. Invoke $\mathcal{F}_{f'}^{\mathsf{OTP}}$ on arguments $(\mathsf{run}, \mathsf{sid}, \mathsf{sen}, \mathsf{digest})$ to get $\{\mathsf{e}_i\}_{i\in[k]}$
  3. For $i \in [k]$, compute $\mathsf{lab}_{i,\mathbf{x}_i} \leftarrow \mathsf{Receive}^{\hat{\mathbf{x}}}(\mathsf{crs}, \mathsf{e}_i, i)$
  4. Compute $y \leftarrow \mathsf{AdaEvalCkt}(\tilde{C}, \{\mathsf{lab}_{i,\mathbf{x}_i}\}_{i\in[k]})$

Theorem 7. *Assuming the existence of a malicious-receiver secure laconic OT scheme, there exists a non-interactive protocol for securely realizing $\mathcal{F}_f^{\mathsf{OTP}}$ against a semi-honest sender and malicious receiver in the $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$-hybrid model, where the number of calls made to the $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$ functionality are proportional to the security parameter and are independent of the length of the receiver's input.*

Proof of Security Let $(\mathsf{SimC}, \mathsf{SimIn})$ and $\ell\mathsf{OTSim} = (\ell\mathsf{OTSim}_1, \ell\mathsf{OTSim}_2)$ be the simulators for the adaptive garbling scheme and laconic OT scheme, respectively.

- **Simulator** $\rightarrow \mathcal{A}$: The simulator begins with crs, receives $(SID, ID, \mathsf{sen}, \mathsf{rec})$ from the ideal functionality $\mathcal{F}_f^{\mathsf{OTP}}$, and proceeds as follows:

- Compute $(\tilde{C}, \mathsf{st}) \leftarrow \mathsf{SimC}(1^\lambda, 1^{|C|})$
- Sample a random sid and id and simulate initiating $\mathcal{F}_{f'}^{\mathsf{OTP}}$ by sending $(sid, id, \mathsf{sen}, \mathsf{rec})$ to $\mathcal{A}$.
- Send $\tilde{C}$ to $\mathcal{A}$

- **$\mathcal{A}$'s Query:** When $\mathcal{A}$ makes a query to $\mathcal{F}_{f'}^{\mathsf{OTP}}$ with arguments $(\mathsf{run}, sid, id, \mathsf{digest})$, do the following:
  - Compute $\mathbf{x} \leftarrow \ell\mathsf{OTSim}_1(\mathsf{crs}, z)$, where z is the auxiliary input of the adversary $\mathcal{A}$.
  - Query $\mathcal{F}_{f}^{\mathsf{OTP}}$ on inputs $(\mathsf{run}, SID, ID, \mathbf{x})$ and get out in return
  - Compute $\{\mathsf{lab}_{i,\mathbf{x}_i}\}_{i \in [k]} \leftarrow \mathsf{SimIn}(\mathsf{st}, \mathsf{out})$
  - For each $i \in [k]$, compute $e_i \leftarrow \ell\mathsf{OTSim}_2(\mathsf{crs}, z, \mathsf{digest}, \mathbf{x}, i, \mathsf{lab}_{i,\mathbf{x}_i})$
  - Return $\{e_i\}_{i \in [k]}$ to $\mathcal{A}$

Lemma 4. $\mathsf{IDEAL}_{S,\mathcal{Z}}^{\mathcal{F}_f^{\mathsf{OTP}}} \approx \mathsf{REAL}_{\Pi^{\mathcal{F}_{f'}^{\mathsf{OTP}}},\mathcal{A},\mathcal{Z}}$

Proof. We now show that the transcript in the real and ideal world are computationally indistinguishable by considering a sequence of hybrids

- **Hybrid** $\mathcal{H}_0$ In this experiment the simulator internally simulates a real execution of the protocol between sen and $\mathcal{A}$ (on their actual inputs), internally simulating the $\mathcal{F}_{f'}^{\mathsf{OTP}}$ and outputs whatever the simulated $\mathcal{A}$ outputs. This is clearly similar to the the REAL scenario.

- **Hybrid** $\mathcal{H}_1$ In this experiment, when $\mathcal{A}$ makes a run query to $\mathcal{F}_{f'}^{\mathsf{OTP}}$, the simulator first computes $\mathbf{x} \leftarrow \ell\mathsf{OTSim}_1(\mathsf{crs}, z)$ before proceeding as in $\mathcal{H}_0$. This only adds additional steps to the computation, and is therefore similar to the previous hybrid.

- **Hybrid** $\mathcal{H}_{1+i}$ ($\forall i \in [M]$) in this experiment, when $\mathcal{A}$ makes a run query to $\mathcal{F}_{f'}^{\mathsf{OTP}}$, the simulator behaves identically to the previous experiment with the exception that it computes $e_i \leftarrow \ell\mathsf{OTSim}(\mathbf{x}, i, \mathsf{lab}_{i,x_i})$.

  Indistinguishability between this hybrid and the previous one follows from the sender privacy against malicious receiver of the laconic OT scheme.

- **Hybrid** $\mathcal{H}_{M+2}$ This hybrid is identical to the ideal execution.

  Indistinguishability between this hybrid and the previous one follows from adaptive security of the garbling scheme.

By transitivity of computational indistinguishability, it follows that the real world execution is indistinguishable from the ideal world execution. □

9 Lockboxes That Allow Multiple Attempts

Up to this point we have only considered lockboxes that allow a single attempt to guess the password. For some real-world instantiations of lockboxes this may not be a valid assumption. In this section we show how the leaky batch-OT scheme can be adapted to support lockboxes that allow for any number of password attempts.

9.1 Overview

Let z be the number of password attempts the lockboxes allow. In the updated scheme, once the sender decides that some lockbox will be a b-lockbox, they do not simply set its password to b. Instead, they generate z distinct, predictable bit strings as prefixes, select one at random, and then set the password to be the concatenation of that string and b.

Honest receivers can simply generate and try all the prefixes for their bit of choice in a given lockbox set, guaranteeing that they open all of the lockboxes they need to reconstruct the desired label.

The prefixes prevent adversarial receivers from getting any advantage from the additional attempts. As intuition, in order to "confirm" that a lockbox is a b-lockbox, they must attempt all the prefixes concatenated with b, thereby running out of attempts.

9.2 Protocol

Let $\mathsf{binaryStrings}(n)$ denote the set of binary strings representing the numbers $1, 2, \ldots, n$, e.g. $\mathsf{binaryStrings}(4) = \{1, 10, 11, 100\}$.

- **Sender.** The protocol remains the same as the protocol described in section 5.2 everywhere other than steps 3 and 6, which are replaced with the following:

  3. For each $j \in [2\ell]$
     - Sample $\mathsf{pw} \xleftarrow{\$} \mathsf{binaryStrings}(z)$
     - If $\pi_i(j) \leq \ell$, invoke $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$ on arguments $(\mathsf{create}, \mathsf{sen}, \mathsf{rec}, sid, id_{i,\pi_i(j)}, \mathsf{pw}||0, z)$ and get $K_{i,0}^{\pi_i(j)}$ in return.
     - Else, invoke $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$ on arguments $(\mathsf{create}, \mathsf{sen}, \mathsf{rec}, sid, id_{i,\pi_i(j)}, \mathsf{pw}||1, z)$ and get $K_{i,1}^{\pi_i(j)}$ in return.

  ...

  6. Send $(\{(C_{i,0}, C_{i,1})\}_{i\in[n]}, z)$ to the receiver

- **Receiver.** Given a set of input bits $\{b_i\}_{i\in[n]}$ and upon receiving $\{(sid, id_{i,\pi_i(j)}, \mathsf{sen}, \mathsf{rec})\}_{j\in[2\ell],i\in[n]}$ from the $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$ functionalities and $(\{(C_{i,0}, C_{i,1})\}_{i\in[n]}, z)$ from the sender, the receiver proceeds as follows for each $i \in [n]$:

  1. For each $j \in [2\ell]$,
     - For each $\mathsf{prefix} \in \mathsf{binaryStrings}(z)$:
       - invoke $\mathcal{F}_\lambda^{\mathsf{Lockbox}}$ on arguments $(\mathsf{open}, \mathsf{sen}, sid, id_{i,\pi_i(j)}, b_i)$ to receive either $K_{i,b_i}^{\pi_i(j)}$ or $\mathsf{bad\_guess}$
       - If $\mathsf{bad\_guess}$ is **not** received, break
     - If $K_{i,b_i}^{\pi_i(j)}$ was not received in the previous loop, set $K_{i,b_i}^{\pi_i(j)} = 0$
  2. Compute $\mathsf{m}_{i,b_i} = C_{i,b_i} \oplus \bigoplus_{j=1}^{2\ell} K_{i,b_i}^{\pi_i(j)}$.

Security. The proof of security for the updated protocol is nearly identical to that of the original protocol so we omit most of it here. We note that the proof of security for the original protocol does not explicitly depend on the number of attempts the adversary has to open each lockbox. Rather, it hinges on the probability of the adversary correctly guessing the password to a lockbox knowing only the proportion of 0 to 1 password lockboxes in a given lockbox-set.

To show that the adversary gains no advantage from the additional attempts, consider the situation where they are attempting to open a lockbox when the proportion of 0 to 1 password lockboxes is even. Given that they do not know whether it is a 0 or 1 password lockbox, nor which of the z possible prefixes the password has, they must guess the single correct random value from a set of $2 \cdot z$ possibilities within z attempts. By the hypergeometric distribution their probability of success in this case is $\frac{1}{2}$, thus granting the adversary no advantage over the original protocol.

Indeed, this on its own shows that the probability of the adversary learning both messages for any index is $\leq (\frac{1}{2})^{\ell}$, from which the rest of the proof follows.

10 Concrete Analysis

In this section, we present a concrete analysis to investigate the suitability of our schemes for real-world applications. In Section 10.1, we estimate the number of lockboxes required for different input lengths. In Section 10.2, we discuss how lockboxes can be instantiated using commodity hardware and the associated costs and finally in Section 10.3, we discuss some potential applications of our construction.

10.1 Number of Lockboxes

We use lockboxes to implement the leaky batch-OT functionality and the input to this functionality is an encoding of the "real" input of the receiver. For encoding, we require linear binary ECC with a constant rate. More often than not, finding optimal binary ECC for specific input lengths k typically requires iterating over all possible alphabets in the domain. In our case, the problem of choosing optimal codes, is made worse by the fact that we don't necessarily require codes with optimal distance γ or the smallest codeword length n. Instead, we want a code that gives the smallest value of $2n\ell$, while ensuring that $\left(\frac{e^{(\epsilon/p-1)}}{(\epsilon/p)^{\epsilon/p}}\right)^{np} < \frac{1}{2^{O(\lambda)}}$, where $p = (\frac{1}{2})^{\ell} \cdot \frac{\ell!}{(2\ell-1)!!}$ and $\epsilon = \gamma/n$ (See Section 5.2 for details). To simplify this problem and to get an estimate of how many lockboxes are required, we pick a particular binary ECC with constant rate and find values of n, γ and ℓ that give the smallest value of $2n\ell$ withing this encoding scheme. In particular, we use Justesen codes [45].

Encoding with Justesen codes. Justesen codes are derived as the code concatenation of a Reed–Solomon code and the Wozencraft ensemble. The encoding algorithm works as follows – the given binary input string of length k is divided into k' blocks of length m each. This new vector of length k' is encoded using the Reed Solomon code $(n', k', n' - k' + 1)$ over field $GF(2^m)$. Finally, the resulting n' blocks of length m each are encoded using Wozencraft ensemble. We use a particular Wozencraft ensemble [51], that yields a final codeword of length $2mn'$. The minimum distance γ of the resulting code is $\sum_{i \in [g]} i \cdot \binom{2m}{i}$, where g is the smallest integer such that $\sum_{i \in [g]} \binom{2m}{i} \leq n' - k' + 1$.

Estimating the optimal no. of lockboxes. Since, n' here can potentially take any value $< 2^m$ (and $m \in [1, k]$), a bruteforce approach to find optimal values even within Justesen code will result in an exponential search. To reduce the search space, we observe that for any given input length k and distance γ, it suffices to only look at the smallest admissible value of n'. Greater values of n' for the same k and γ yield worse security and larger values of $2n\ell$. We use this observation to deploy the following strategy – for any input length k, iterate over all possible values of $m \in [1, k]$, compute all corresponding admissible values of g, γ and set $n' = k' + \left(\sum_{i\in[g]} \binom{2m}{i}\right) - 1$ (this significantly reduces potential domain for n'). For each such combination of (m, n', γ), we calculate security for reasonable values of ℓ and find the combination of $(n', k', m, \gamma, \ell)$ that results in the fewest total number of lockboxes, while ensuring that the security is at least 2^{-50}.

We report the number of lockboxes required for some input lengths in Table 1. As expected, the number of lockboxes per input wire decreases as the number of inputs increase. By replacing Wozencraft ensemble with BCH codes [15], we can hope to get small improvements for larger input lengths; however, for smaller inputs, BCH codes are unlikely to help. Overall, due to the lack of efficient binary linear ECC, the number of required lockboxes are unlikely to be significantly better than the ones computed using Justesen codes. Our laconic OT-based construction offers some relief in this regard: for instance, if the length of digest output by the receiver is 256 bits, we require 10,528 total lockboxes for *any* input length.

Input	Codeword	(n', k', m, γ)	ℓ	Total LB	LB / Bit
Length (k)	Length (n)			$(2n\ell)$	$(2n\ell/k)$
192	496	(43,32,6,12)	7	7224	37.625
256	752	(47,32,8,16)	7	10528	41.125
560	1302	(93,80,7,14)	7	18228	32.55
1024	2400	(143,128,8,16)	7	32032	31.28125
5000	14180	(709,500,10,400)	4	113440	22.688
300000	735720	(24524,20000,15,13080)	4	5885760	19.6192

Table 1: Lockboxes required for various input lengths with statistical security parameter $\lambda \geq 50$

10.2 Instantiating Lockboxes

To realize counter lockboxes from the widely-available device- and cloud-based hardware, some implementation considerations arise. In this section, we provide brief background on each candidate lockbox and the practical considerations involving their use.

Cloud-based Backup Services. Apple's Cloud Key Vault was introduced in 2016 when Apple added functionality to encrypt and store user-controlled encryption keys within hardware security modules (HSM) to remove Apple's own ability to access them. Each iCloud account (registered email address) has access to a Cloud Key Vault record, which corresponds to a password-protected HSM entry which allows up to ten attempts[6] via the Secure Remote Password [4, 5] (SRP) protocol. Notably, this requires one email address per lockbox, as Apple allocates a single Cloud Key Vault entry to each user account.

Similar to Apple's Cloud Key Vault, Google introduced HSM-based user-controlled encryption to protect backups even from insider threats [47]. Their system relies on the Titan [58] HSM hardware, and similarly implements a password-based attempt-limited authentication service which can naturally be viewed as a counter lockbox. Akin to Apple's Cloud Key Vault, Google allocates a single backup service instance per user account, and so each lockbox requires a registered Google account (email address) to be deployed. Both iCloud and Google accounts can be acquired for free, but acquiring multiple accounts can require evading anti-spam measures.

Signal, the secure messaging platform, offers users a backup method relying on user-controlled encryption inaccessible to Signal's servers. This service is called Secure Value Recovery, or SVR. SVR allows

[6] In Section 2.4, we discuss generic techniques to convert a multiple-attempt (e.g. 10) lockbox into a single-attempt, including simply "burning" $n - 1$ attempts of each n-attempt lockbox before transmitting their locations to the receiver.

users to set a PIN, and gives them ten attempts to authenticate to an Intel SGX enclave to retrieve their backup data. As a secure enclave, SGX itself is capable of running one-time programs. However, to end users only a basic API is exposed which allows authentication attempts over a secure connection. Rather than email-based registration, Signal requires phone numbers, specifically to receive a confirmation SMS. Therefore, each SVR lockbox requires a phone number able to receive an SMS; such numbers cost $0.50 USD/month each at scale with a service like Twilio [60].

iOS Devices. Apple also offers the eponymous counter lockbox as hardware within modern iOS devices (smartphones and tablets) available since Fall 2020. This component emerged with the second-generation Secure Enclave Processor [3] (SEP) and was designed to prevent forensic attacks against the passcode attempt counter which moderates access to a device and its filesystem. Although there are few official documents, initial exploration seems to imply that iOS devices are able to support up to 1024 counter lockbox instances simultaneously. Since counter lockboxes are intended for use by iOS itself, third-party developers must interact with them directly on jailbroken devices. Finally, a note on monetary costs: currently, iPad air 4th generation can be purchased for about $300. Thus, the average cost of each lockbox can be estimated to be about $0.30 USD.

10.3 Applications

Given the cost of each lockbox and the notable expansion between input length and total lockboxes as seen in Table 1, at present the real-world applicability of our constructions is somewhat limited. However, compelling applications involving small input lengths are within reach: Bitcoin addresses are 160-bit hashes, which could be input into a delegated signature one-time program. Down-sampled biometric measurements could be input to fuzzy matching algorithms, or passwords into client-side key derivations for user authentication. Compressed descriptions of aggregations could be input to an offline differentially-private database service to maintain privacy budgets. As lockbox availability grows, these domains will only expand.

11 Conclusions

In this work,we investigate whether real-world one-time programs (and all their numerous results: obfuscation, authentication, autonomous ransomware etc.) can be brought closer to fruition using commodity hardware. We formalize a new, simple, and widely-available hardware functionality – the counter lockbox – and use it to build one-time programs. Our current techniques are most suitable for small program input lengths, and require numerous lockboxes for moderate to large input lengths, which may be expensive for many applications. However, not only can these costs decrease with the further proliferation of lockbox hardware, we lay the groundwork for newer cryptographic techniques to further reduce these costs.

12 Acknowledgements

The first, second and fourth authors were supported in part by NSF CNS-1814919, NSF CAREER 1942789 and Johns Hopkins University Catalyst award. The fourth author was additionally supported in part by AFOSR Award FA9550-19-1-0200 and the Office of Naval Research Grant N00014-19-1-2294. The first, third and fifth authors were supported by the National Science Foundation under awards CNS-1653110 and CNS-1801479 and by DARPA under Agreements No. HR00112020021 and Agreements No. HR001120C0084. The fifth author was additionally supported by a Google Security & Privacy Award. This work was done in part while the second author was a student at Johns Hopkins University and while the second and fourth authors were visiting University of California, Berkeley. 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.

References

[1] Navid Alamati, Pedro Branco, Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, and Sihang Pu. Laconic private set intersection and applications. Cryptology ePrint Archive, Report 2021/728, 2021. https://eprint.iacr.org/2021/728.

[2] Ghada Almashaqbeh, Fabrice Benhamouda, Seungwook Han, Daniel Jaroslawicz, Tal Malkin, Alex Nicita, Tal Rabin, Abhishek Shah, and Eran Tromer. Gage mpc: Bypassing residual function leakage for non-interactive mpc. *Proceedings on Privacy Enhancing Technologies*, 2021(4):528–548, 2021.

[3] Apple Inc. Secure Enclave. https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web.

[4] Apple Inc. Escrow security for iCloud Keychain. https://support.apple.com/guide/security/escrow-security-for-icloud-keychain-sec3e341e75d/web, 2021.

[5] Apple Inc. HomeKit communication security. https://support.apple.com/guide/security/homekit-communication-security-sec3a881ccb1/web, 2021.

[6] ARM Holdings. Trusted Base System Architecture Documents. https://www.arm.com/technologies/trustzone-for-cortex-a/tee-reference-documentation. Subject to Non-Disclosure Agreement.

[7] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 483–501. Springer, Heidelberg, April 2012.

[8] Michael Backes, Rainer W. Gerling, Sebastian Gerling, Stefan Nürnberger, Dominique Schröder, and Mark Simkin. WebTrust - A comprehensive authenticity and integrity framework for HTTP. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 14*, volume 8479 of *LNCS*, pages 401–418. Springer, Heidelberg, June 2014.

[9] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 134–153. Springer, Heidelberg, December 2012.

[10] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *CCS '12*, 2012.

[11] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019.

[12] Fabrice Benhamouda, Hugo Krawczyk, and Tal Rabin. Robust non-interactive multiparty computation against constant-size collusion. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 391–419. Springer, Heidelberg, August 2017.

[13] Alpesh Bhudia, Daniel O'Keeffe, Daniele Sgandurra, and Darren Hurley-Smith. Ransomclave: Ransomware key management using sgx. In *The 16th International Conference on Availability, Reliability and Security*, 2021.

[14] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. The hunting of the SNARK. *J. Cryptol.*, 30(4):989–1066, 2017.

[15] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, 1960.

[16] Anne Broadbent, Gus Gutoski, and Douglas Stebila. Quantum one-time programs - (extended abstract). In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 344–360. Springer, Heidelberg, August 2013.

[17] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Breaking virtual memory protection and the SGX ecosystem with foreshadow. *IEEE Micro*, 39(3):66–74, 2019.

[18] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS '01*, pages 136–145. IEEE, 2001.

[19] Rahul Chatterjee, Anish Athayle, Devdatta Akhawe, Ari Juels, and Thomas Ristenpart. password typos and how to correct them securely. In *S&P '16*. IEEE, 2016.

[20] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, August 1993.

[21] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[22] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 769–793. Springer, Heidelberg, May 2020.

[23] Chongwon Cho, Nico Döttling, Sanjam Garg, Divya Gupta, Peihan Miao, and Antigoni Polychroniadou. Laconic oblivious transfer and its applications. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 33–65. Springer, Heidelberg, August 2017.

[24] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):171–191, 2018.

[25] Oscar Delgado-Mohatar, José María Sierra-Cámara, and Eloy Anguiano. Blockchain-based semi-autonomous ransomware. *Future Generation Computer Systems*, 112:589–603, 2020.

[26] Nico Döttling, Sanjam Garg, Vipul Goyal, and Giulio Malavolta. Laconic conditional disclosure of secrets and applications. In David Zuckerman, editor, *60th FOCS*, pages 661–685. IEEE Computer Society Press, November 2019.

[27] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, 2006.

[28] Dario Fiore and Anca Nitulescu. On the (in)security of SNARKs in the presence of oracles. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 108–138. Springer, Heidelberg, October / November 2016.

[29] Sanjam Garg and Akshayaram Srinivasan. Adaptively secure garbling with near optimal online complexity. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 535–565. Springer, Heidelberg, April / May 2018.

[30] Oded Goldreich. *Foundations of Cryptology: Basic Tools*. Cambridge, 2001.

[31] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[32] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 113–122. ACM Press, May 2008.

[33] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. One-time programs. In *Annual International Cryptology Conference*, pages 39–56, 2008.

[34] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[35] Google. Google Tensor debuts on the new Pixel 6 this fall. https://blog.google/products/pixel/google-tensor-debuts-new-pixel-6-fall/, 2021.

[36] Rishab Goyal and Vipul Goyal. Overcoming cryptographic impossibility results using blockchains. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 529–561. Springer, Heidelberg, November 2017.

[37] Rishab Goyal, Satyanarayana Vusirikala, and Brent Waters. New constructions of hinting PRGs, OWFs with encryption, and more. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 527–558. Springer, Heidelberg, August 2020.

[38] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 308–326. Springer, Heidelberg, February 2010.

[39] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[40] Danny Harnik, Joe Kilian, Moni Naor, Omer Reingold, and Alon Rosen. On robust combiners for oblivious transfer and other primitives. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 96–113. Springer, Heidelberg, May 2005.

[41] Carmit Hazay and Yehuda Lindell. Constructions of truly practical secure protocols using standard-smartcards. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 491–500. ACM Press, October 2008.

[42] Brett Hemenway, Zahra Jafargholi, Rafail Ostrovsky, Alessandra Scafuro, and Daniel Wichs. Adaptively secure garbled circuits from one-way functions. In *Annual International Cryptology Conference*. Springer, 2016.

[43] Intel. Overview on signing and whitelisting for intel software guard extension (sgx) enclaves. https://www.intel.com/content/dam/develop/external/us/en/documents/overview-signing-whitelisting-intel-sgx-enclaves-737361.pdf.

[44] Ari Juels and Madhu Sudan. A fuzzy vault scheme. *Designs, Codes and Cryptography*, 38(2):237–257, 2006.

[45] Jørn Justesen. Class of constructive asymptotically good algebraic codes. *IEEE Transactions on Information Theory*, 18(5):652–656, 1972.

[46] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *NDSS '19*, 2019.

[47] Troy Kensinger. Google and Android have your back by protecting your backups. https://security.googleblog.com/2018/10/google-and-android-have-your-back-by.html, 10 2018.

[48] Slavik Krassovsky and Gabriel et al Cadden. Security of End-To-End Encrypted Backups. https://scontent.whatsapp.net/v/t39.8562-34/241394876546674233234181890713788950030 1879n.pdf/WhatsAppSecurityEncryptedBackupsWhitepaper.pdf?ccb=1-5&ncsid=2fbf2a&ncohc=4K040x7GheAAX-4c-&ncht=scontent.whatsapp.net&oh=01AVxDv1cRlVElvg0Fv89URSUXOQUupw70bDPw6o2w0LEWg&oe=6211F5FC, 2021.

[49] Ivan Krstić. Behind the scenes with iOS security. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf, 2016.

[50] J. Lund. https://signal.org/blog/secure-value-recovery/, 12 2019. Accessed 2 May 2022.

[51] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The Theory of Error-Correcting Codes*. North-Holland Pub. Co., 1977.

[52] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *HASP '16*. ACM, 2016.

[53] Remo Meier, Bartosz Przydatek, and Jürg Wullschleger. Robuster combiners for oblivious transfer. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 404–418. Springer, Heidelberg, February 2007.

[54] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[55] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *S&P '20*. IEEE, 2020.

[56] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6), 2019.

[57] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 94–124, Virtual Event, August 2021. Springer, Heidelberg.

[58] Uday Savagaonkar, Nelly Porter, Nadim Taha, Benjamin Serebrin, and Neal Mueller. Titan in depth: Security in plaintext. https://cloud.google.com/blog/products/identity-security/titan-in-depth-security-in-plaintext, 2017.

[59] Michael Sipser and Daniel A Spielman. Expander codes. *IEEE transactions on Information Theory*, 42(6):1710–1722, 1996.

[60] Twilio. https://www.twilio.com/sms/pricing/us, 2022.

[61] Qixiang Xu. ARM-software/tf-issues. https://github.com/ARM-software/tf-issues/issues/534, 2017.

[62] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[63] yubico. YubiHSM 2. https://www.yubico.com/product/yubihsm-2/.

Appendix

A Preliminaries

We begin by recalling the definition of *computational indistinguishability*:

Definition 4. *(Computational Indistinguishability) Two distribution ensembles $X := \{X_n\}_{n \in \boldsymbol{N}}$ and $Y := \{Y_n\}_{n \in \boldsymbol{N}}$ are* ***computationally indistinguishable*** *(written as $X \sim Y$) if for every probabilistic polynomial-time algorithm D, every positive polynomial $p(\cdot)$, and all sufficiently large n's,*

$$|\Pr[D(X_n, 1^n) = 1] - \Pr[D(Y_n, 1^n) = 1]| \leq \frac{1}{p(n)}.$$

The statistical difference *between two distribution ensembles $X := \{X_n\}_{n \in \boldsymbol{N}}$ and $Y := \{Y_n\}_{n \in \boldsymbol{N}}$ is defined by*

$$\Delta(n) = \frac{1}{2} \sum_{\alpha} |\Pr[X_n = \alpha] - \Pr[Y_n = a]|.$$

Ensembles X and Y are statistically close *if their statistical difference is negligible in n.*

A.1 The UC-Framework

We reproduce here, with a few changes, a description of the UC-framework as presented in [38]. We use the UC-framework of Canetti [18] to capture the general notion of secure computation of (possibly reactive) functionalities. We will only consider the two-party case, referring to one party as a "sender" and to another as a "receiver". We begin by defining protocol syntax, and then informally review the UC-framework. For more details, see [18].

Protocol Syntax Following [34] and [30], a protocol is represented as a system of probabilistic interactive Turing machines (ITMs), where each ITM represents the program to be run within a different party. Specifically, the input and output tapes model inputs and outputs that are received from and given to other programs running on the same machine, and the communication tapes model messages sent to and received from the network. Adversarial entities are also modeled as ITMs. The construction of a protocol in the UC-framework proceeds as follows: first, an *ideal functionality* is defined, which is a "trusted party" that is guaranteed to accurately capture the desired functionality. Then, the process of executing a protocol in the presence of an adversary and in a given computational environment is formalized. This is called the *real-life* model. Finally, an *ideal process* is considered, where the parties only interact with the ideal functionality, and not amongst themselves. Informally, a protocol realizes an ideal functionality if running of the protocol amounts to "emulating" the ideal process for that functionality. Let $\Pi = (P_1, P_2)$ be a protocol, and $\mathcal{F}$ be the ideal-functionality. We describe the ideal and real world executions.

The real-life model The real-life model consists of the two parties P_1 and P_2, the environment Z, and the adversary $\mathcal{A}$. Adversary $\mathcal{A}$ can communicate with environment $\mathcal{Z}$ and can corrupt any party. When A corrupts party P_i, it learns P_i's entire internal state, and takes complete control of P_i's input/output behaviour. The environment $\mathcal{Z}$ sets the parties' initial inputs. Let $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$be the distribution ensemble that describes the environment's output when protocol Π is run with adversary A.

The ideal process The ideal process consists of two "dummy parties" $\hat{P}_1$ and $\hat{P}_2$, the ideal functionality $\mathcal{F}$, the environment $\mathcal{Z}$, and the ideal world adversary S, called the simulator. In the ideal world, the uncorrupted dummy parties obtain their inputs from environment $\mathcal{Z}$ and simply hand them over to $\mathcal{F}$. As in the real world, adversary S can corrupt any party. Once it corrupts party $\hat{P}_i$, it learns $\hat{P}_i$'s input, and takes complete control of its input/output behaviour. Let $\mathsf{IDEAL}^{\mathcal{F}}_{S,\mathcal{Z}}$ be the distribution ensemble that describes the environment's output in the ideal process.

Definition 5. *(Realizing an Ideal Functionality) Let $\mathcal{F}$ be an ideal functionality, and Π be a protocol. We say Π* ***realizes*** *$\mathcal{F}$ if for any real-world adversary $\mathcal{A}$, there exists an ideal process adversary S such that for every environment $\mathcal{Z}$,*

$$\mathsf{IDEAL}^{\mathcal{F}}_{S,\mathcal{Z}} \approx \mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$$

A.2 Garbling Schemes.

Our construction depends on an *adaptive*, *projective* garbling scheme. *Adaptive* security requires that the garbled circuit remain secure even when the receiver specifies its input after seeing the garbled circuit.

In a garbling scheme the circuit garbling algorithm produces some state, which is then used in the input garbling algorithm to produce some garbled input. In a *projective* garbling scheme the state produced by the garbling algorithm is a set of label pairs, one pair for each input wire of the circuit. The task of the input garbling algorithm is then reduced to selecting these labels by the bits of the input to be garbled.

We now give a formal definition of a garbling scheme and adaptive security. We use the definitions as given in [42].

Definition 6. *An adaptive garbling scheme for circuits is a tuple of PPT algorithms* $(\mathsf{AdaGarbleCkt}, \mathsf{AdaGarbleInp}, \mathsf{AdaEvalCkt})$

- $(\tilde{C}, \mathsf{st}) \leftarrow \mathsf{AdaGarbleCkt}(1^\lambda, C)$*: Is a PPT algorithm that takes as input the security parameter* 1^λ *(encoded in unary) and a circuit* $C : \{0,1\}^n \rightarrow \{0,1\}^m$ *as input and outputs a garbled circuit* $\tilde{C}$ *and state information* st
- $\tilde{x} \leftarrow \mathsf{AdaGarbleInp}(\mathsf{st}, x)$*: It is a PPT algorithm that takes as input the state information* st *and an input* $x \in \{0,1\}^n$ *and outputs the garbled input* $\tilde{x}$
- $y = \mathsf{AdaEvalCkt}(\tilde{C}, \tilde{x})$*: Given a garbled circuit* $\tilde{C}$ *and a garbled input* $\tilde{x}$*, it outputs a value* $y \in \{0,1\}^m$

Correctness. *For every* $\lambda \in \mathbb{N}, C : \{0,1\}^n \rightarrow \{0,1\}^m$ *and* $x \in \{0,1\}^n$ *it holds that:*

Adaptive Security. *There exists a PPT simulator such that, for any non-uniform PPT adversary* $\mathcal{A}$ *there exists a negligible function* v *such that:*

$$|\Pr[\mathsf{Exp}^{\mathsf{adaptive}}_{\mathcal{A},\mathsf{GC},\mathsf{Sim}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{GC},\mathsf{Sim}}(1^\lambda, 1) = 1]| \leq v(\lambda)$$

where the experiment $\mathsf{Exp}^{\mathsf{adaptive}}_{\mathcal{A},\mathsf{GC},\mathsf{Sim}}(1^\lambda$ *is defined as follows:*

1. *The adversary specifies the circuit* C *and obtains* $\tilde{C}$ *where* $\tilde{C}$ *is created as follows:*
   - If $b = 0 : (\tilde{C}, \mathsf{st}) \leftarrow \mathsf{AdaGarbleCkt}(1^\lambda, C)$
   - If $b = 1 : (\tilde{C}, \mathsf{st}) \leftarrow \mathsf{SimC}(1^\lambda, 1^{|C|})$
2. *The adversary* $\mathcal{A}$ *specifies the input* x *and gets* $\tilde{x}$ *created as follows:*
   - *If* $b = 0, \tilde{x} \leftarrow \mathsf{AdaGarbleInp}(\mathsf{st}, x)$.
   - *If* $b = 1, \tilde{x} \leftarrow \mathsf{SimIn}(\mathsf{st}, C(x))$
3. *Finally, the adversary outputs a bit* b'*, which is the output of the experiment*

We now repeat the definition of *projective* garbling as given in [10].

Definition 7. *(Projective Garbled Circuits) A garbling scheme is said to be projective if the* st *output by* $\mathsf{AdaGarbleCkt}$ *encodes a list of tokens, one pair for each bit* $i \in [n]$. $\mathsf{AdaGarbleInp}$ *then uses the bits of* $x = x_1 ... x_n$ *to select from* $\mathsf{st} = (\mathsf{lab}_{1,0}, \mathsf{lab}_{1,1}, ..., \mathsf{lab}_{n,0}, \mathsf{lab}_{n,1})$. *Formally, we say that the garbling scheme* $(\mathsf{AdaGarbleCkt}, \mathsf{AdaGarbleInp}, \mathsf{AdaEvalCkt})$ *is* projective *if for all* $C, x, x' \in \{0,1\}^n, \lambda \in \mathbb{N}$*, and* $i \in [n]$*, when* $(\tilde{C}, \mathsf{st}) = \mathsf{AdaGarbleCkt}(1^\lambda, C), \tilde{x} = \mathsf{AdaGarbleInp}(\mathsf{st}, x)$ *and* $\tilde{x}' = \mathsf{AdaGarbleInp}(\mathsf{st}, x')$*, then* $\tilde{x} = (\mathsf{lab}_1 ... \mathsf{lab}_n)$ *and* $\tilde{x} = (\mathsf{lab}'_1 ... \mathsf{lab}'_n)$ *are* n *vectors,* $|\tilde{x}_i| = |\tilde{x}'_i|$*, and* $\tilde{x}_i = \tilde{x}'_i$ *if* x *and* x' *have the same* i*th bit.*

A.3 Linear Error Correcting Codes

We make use of binary linear error correcting codes. Our primary use of such codes is their *minimum distance* property, ensuring that any two valid codewords have a guaranteed minimum Hamming distance between them. Linear codes have the additional property that they can be defined with a generating matrix G that can be used to perform the encoding and decoding steps via matrix multiplication.

Definition 8. *(Linear Error Correcting Codes for $\mathbb{F}_2$) A $[n, k, \gamma]_2$ linear code is a linear subspace C with dimension k of $\mathbb{F}_2^n$, such that the Hamming Distance of any two distinct codewords $\mathbf{x}, \mathbf{x}' \in C$ is at least γ (called the minimum distance of C). A generating matrix G of C is a $n \times k$-matrix whose rows generate the subspace C. The* relative distance *of a code is $\delta = \frac{\gamma}{n}$, and the* rate *of a code is $R = \frac{k}{n}$.*

We recall that there exist linear error correcting codes for $\mathbb{F}_2$ with constant rate and relative distance, such as expander [59] and Justesen [45] codes.

A.4 Succinct Non-Interactive Arguments of Knowledge

In this section, we provide a formal definition for the notion of succinct non-interactive arguments of knowledge (SNARKs). This definition is taken from [28].

Definition 9 (SNARKs). *A succinct non-interactive argument (SNARK) of knowledge for a relation $\mathcal{R} \in \mathcal{R}_{\mathcal{U}}$ is a triple of algorithms $\Pi = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Ver})$ working as follows*

- $\mathsf{Gen}(1^\lambda, \mathsf{T}) \rightarrow \mathsf{crs}$*: on input security parameter $\lambda \in \mathbb{N}$ and a time bound $T \in \mathbb{N}$, the generation algorithm outputs a common reference string $\mathsf{crs} = (\mathsf{prs}, \mathsf{vst})$ consisting of a public prover reference string* prs *and a verification state* vst.
- $\mathsf{Prove}(\mathsf{prs}, \mathsf{y}, \mathsf{w}) \rightarrow \pi$*: given a prover reference string* prs*, an instance $y = (M, x, t)$ with $t \leq T$ and a witness w s.t. $(y, w) \in \mathcal{R}$, this algorithm produces a proof π.*
- $\mathsf{Ver}(\mathsf{vst}, \mathsf{y}, \pi) \rightarrow \mathsf{b}$*: on input a verification state* vst*, an instance y, and a proof π, the verifier algorithm outputs $b = 0$ (reject) or $b = 1$ (accept).*

and satisfying *completeness, succinctness,* and *(adaptive) proof of knowledge* as described below*:*

- ***Completeness.*** *For every time bound $T \in \mathbb{N}$, every valid $(y, w) \in \mathcal{R}$ with $y = (M, x, t)$ and $t \leq T$, there exists a negligible function* negl *such that*

$$\Pr\left[\mathsf{Ver}(\mathsf{vst}, \mathsf{y}, \pi) = 0 \middle|^{(\mathsf{prs},\mathsf{vst}) \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{T})}_{\pi \leftarrow \mathsf{Prove}(\mathsf{prs},\mathsf{y},\mathsf{w})}\right] \leq \mathsf{negl}(\lambda)$$

- ***Succinctness.*** *There exists a fixed polynomial $p(\cdot)$ independent of $\mathcal{R}$ such that for every large enough security parameter $\lambda \in \mathbb{N}$, every time bound $T \in \mathbb{N}$, and every instance $y = (M, x, t)$ such that $t \leq T$, we have*
  - Gen *runs in time*

$$\begin{cases} p(\lambda + logT) & \text{\textit{for a fully-succinct SNARG}} \\ p(\lambda + T) & \text{\textit{for a pre-processing SNARG}} \end{cases}$$

- Prove *runs in time*

$$\begin{cases} p(\lambda + |M| + |x| + t + \mathit{log}T) & \textit{for a fully-succinct SNARG} \\ p(\lambda + |M| + |x| + T) & \textit{for a pre-processing SNARG} \end{cases}$$

- Ver *runs in time* $p(\lambda + |M| + |x| + \mathit{log}T)$
- *an honestly generated proof has size* $|\pi| = p(\lambda + \mathit{log}T)$

• ***Adaptive Proof of Knowledge.*** *For every non-uniform prover $\mathcal{A}$ of size $s(\lambda) = \mathsf{poly}(\lambda)$ there exists a non-uniform extractor $\mathcal{E}_{\mathcal{A}}$ of size $t(\lambda) = \mathsf{poly}(\lambda)$ and a negligible function $\epsilon(\lambda)$ such that for every auxiliary input $aux \in \{0,1\}^{\mathsf{poly}(\lambda)}$, and every time bound $T \in \mathbb{N}$,*

$$\Pr\left[\begin{smallmatrix}\mathsf{Ver}(\mathsf{vst},\mathsf{y},\pi)=1\\ \wedge(y,w)\notin\mathcal{R}\end{smallmatrix}\middle|\begin{smallmatrix}(\mathsf{prs},\mathsf{vst})\leftarrow\mathsf{Gen}(1^\lambda,\mathsf{T})\\ (y,\pi)\leftarrow\mathcal{A}(\mathsf{prs},\mathsf{aux})\\ w\leftarrow\mathcal{E}_{\mathcal{A}}(\mathsf{prs},\mathsf{aux})\end{smallmatrix}\right] \leq \epsilon(\lambda)$$

Furthermore, we say that Π satisfies (s,t,ϵ)-adaptive proof of knowledge if the above condition holds for concrete values (s,t,ϵ).

In recent years, several SNARK constructions have been proposed [39, 22, 11, 54, 14] both in the random oracle model and the standard model based on a variety of assumptions.