# Rate-Limited Secure Function Evaluation: Definitions and Constructions

Özgür Dagdelen[1], Payman Mohassel[2], and Daniele Venturi[3]

[1] Technische Universität Darmstadt, Germany
[2] University of Calgary, Canada
[3] Aarhus University, Denmark

**Abstract.** We introduce the notion of rate-limited secure function evaluation (RL-SFE). Loosely speaking, in an RL-SFE protocol participants can monitor and limit the number of distinct inputs (i.e., *rate*) used by their counterparts in multiple executions of an SFE, in a private and verifiable manner. The need for RL-SFE naturally arises in a variety of scenarios: e.g., it enables service providers to "meter" their customers' usage without compromising their privacy, or can be used to prevent oracle attacks against SFE constructions.
We consider three variants of RL-SFE providing different levels of security. As a stepping stone, we also formalize the notion of commit-first SFE (cf-SFE) wherein parties are committed to their inputs before each SFE execution. We provide compilers for transforming any cf-SFE protocol into each of the three RL-SFE variants. Our compilers are accompanied with simulation-based proofs of security in the standard model and show a clear tradeoff between the level of security offered and the overhead required. Moreover, motivated by the fact that in many client-server applications clients do not keep state, we also describe a general approach for transforming the resulting RL-SFE protocols into *stateless* ones.
As a case study, we take a closer look at the oblivious polynomial evaluation (OPE) protocol of Hazay and Lindell, show that it is commit-first and instantiate efficient rate-limited variants of it.

**Keywords.** secure function evaluation, foundations, secure metering, oracle attacks, oblivious polynomial evaluation

## 1 Introduction

Secure function evaluation (SFE) allows a set of mutually distrustful parties to securely compute a function $f$ of their private inputs. Roughly speaking, SFE protocols guarantee that the function is computed correctly and that the parties will not learn any information from the interaction other than their output and what is inherently leaked from it. Seminal results in SFE show that one can securely compute any functionality [29,30,15,8,2]. There has been a large number of follow-up work improving the security, strengthening adversarial models, and studying efficiency. Recent work on practical SFE has also led to real-world deployments [7,6], and the design and implementation of several SFE frameworks [24,5,12,20,21].

In practice, most applications of SFE considered in the literature need to accommodate *multiple executions* of a protocol.[1] Consider a client that searches for multiple patterns in a large text via a secure pattern matching protocol [17,19], searches several keywords in a private database via an oblivious keyword search [27,13], or an individual who needs to run a software diagnostic program, or an intrusion detection system (IDS) to analyze data via an oblivious branching program (OBP) or an automaton evaluation (OAE) protocol [22,28].

Invoking an SFE protocol multiple times raises important practical issues that are outside the scope of standard SFE, and hence are not addressed by the existing solutions. We point out two such issues and introduce *rate-limited* SFE as a means to address them. The reason for the choice of name is that rate-limiting is commonly used in network and web applications to refer to restrictions put on clients' usage (on a per user, or a per IP address basis). In this work we consider similar restrictions on a user's inputs to services that maybe implemented using SFE.

Secure Metering of SFE. Service providers tend to charge their clients according to their level of usage: a location-based service may wish to charge its clients based on the number of locations they use the service from; a database owner based on the number of distinct search queries; an IDS provider based on the number of suspicious files sent for vulnerability analysis. Service providers would be more willing to adopt SFE protocols if it is possible to efficiently enforce such a metering mechanism. The challenge is to do so without compromising the client's privacy, or allowing the server or the client to cheat the metering system.

Oracle attacks. Consider multiple executions of a two-party SFE protocol (such as those mentioned above), where the first party's input stays the same in different executions but the second party's input varies. A malicious second party who "adaptively" uses different inputs in each execution, can gradually learn significant information about the first party's input, and, in the worst case, fully recover it. For instance, consider an oblivious polynomial evaluation (OPE) protocol (e.g., used in oblivious keyword search) wherein the server holds a polynomial $p$ while the client holds a private point $x$ and wants to learn $p(x)$, but cannot learn more than it. Evaluating the polynomial $p$ on sufficiently many points allows a malicious client to interpolate and recover $p$. A similar attack can be applied to OBP and OAE protocols to learn the private branching program or automaton which may embed propriety information. Learning attacks of this sort are well-understood and have been previously identified as important threats in the context of SFE; they are sometimes referred to as *oracle attacks* since the attacker has black-box access to input/output values from multiple executions (e.g., see the discussion in [1]).

A naïve solution to the problems discussed above is to limit the total number of executions of an SFE protocol, ignoring the actual input values. However, this approach does not provide a satisfactory solution in most scenarios. For

[1] Depending on the application, a subset of the participants may use the same input in different executions.

example, in case of secure metering, fixing an a priori upper bound on the total number of executions would mean charging legitimate clients multiple times for using the service with the same input; a disadvantage for clients who may need to use the same input from multiple devices, or reproduce a result due to communication errors, device upgrades, or perhaps to prove the validity of the outcome to a third-party by re-running the protocol. Similarly, in case of oracle attacks, clients need not be disallowed to use the same input multiple times since querying the same input many times does not yield new information to an attacker.

RATE-LIMITED SFE. A more accurate (and challenging) solution is to *limit* and/or *monitor* the number of distinct inputs used by an SFE participant in multiple executions. Obviously, this should be done in a secure and efficient manner, i.e., a party should not be able to exceed an agreed-upon limit, and its counterpart should not learn any additional information about his private inputs, or impose a lower limit than the one they agreed on.[2] We refer to the number of distinct inputs used by a participant as his *rate*, and call a SFE protocol that monitors/limits this number, a rate-limited SFE.

Of course, achieving RL-SFE is more costly than the naïve solution discussed above. However, at a minimum we require the proposed solution to avoid storing and/or processing the complete transcripts of all previous executions. (We discuss the exact overhead of our solutions in detail below.)

We note that the complementary question of what functions are *unsafe* for use in SFE (leak too much information) has also been studied, e.g., by combining SFE and differential privacy [3,26], or belief tracking techniques [25]. These works are orthogonal to ours, and can potentially be used in conjunction with rate-limited SFE as an *enforcement mechanism*. For instance, the former works can be invoked to negotiate on a function $f$ with a measurable "safeness" from which the rate for each user can be derived. Subsequently, the abidance of this rate can be enforced through our rate-limited SFE.

**Our Contribution.** Motivated by the discussion above, we initiate the study of rate-limited SFE. For simplicity, in this paper we focus on the two-party case, but point out that the definitions and some of the constructions are easily extendible to the multiparty setting. Our main contributions are as follows.

DEFINITIONS. We introduce three definitions for rate-limited secure function evaluation: (i) rate-hiding, (ii) rate-revealing and (iii) pattern-revealing. All our definitions are in the real-world/ideal-world simulation paradigm and are concerned with *multiple* sequential executions of an SFE protocol. They reduce to the standard simulation-based definition (stand alone) for SFE, when applied to a single execution.

In a *rate-hiding* RL-SFE, in each execution, the only information revealed to the parties is whether the agreed-upon rate limit has been exceeded or not.

[2] In fact the problem becomes significantly easier when the parties are assumed to be semi-honest.

In a *rate-revealing* RL-SFE, the parties additionally learn the current rate (i.e., the number of distinct inputs used by their counterpart so far). In a *pattern-revealing* RL-SFE, parties also learn the pattern of occurrences of each other's inputs during the previous executions. These notions provide a useful spectrum of tradeoffs between security and efficiency: our constructions become more efficient as we move to the more relaxed notions, to the extent that *our pattern-revealing transformation essentially adds no overhead* to the underlying SFE protocol.

Commit-first SFE. In order to design rate-limited SFE protocols, we formalize the auxiliary notion of commit-first SFE (cf-SFE). Roughly speaking, a protocol is commit-first if it can be naturally divided into a (i) *committing phase*, where each party becomes committed to its input for the second phase, and (ii) a *function evaluation phase*, where the function $f$ is computed on the inputs committed to in the first phase.[3]

We utilize cf-SFE as a stepping stone to design rate-limited SFE. It turns out that the separation between the input commitment phase and the function evaluation phase facilitates the design of efficient rate-limited SFE. In particular, now a party only needs to provide some evidence of a particular relation between the committed inputs in the first phase. In contrast, if we had not started with a commit-first protocol, such an argument would have involved the complete history of the transcripts for all the previous executions, rendering such an approach impractical.

The related notion of "evaluating on committed inputs" is well-known (e.g. see [15,23]), but we need and put forth a formal (and general) definition for cf-SFE in order to prove our RL-SFE protocols secure. We then show that several existing SFE constructions are either commit-first or can be efficiently transformed into one. Examples include variants of Yao's garbled circuit protocol, the oblivious polynomial evaluation of Hazay and Lindell [16], the private set intersection protocol of Hazay and Nissim [18], and oblivious automaton evaluation of Gennaro *et al.* [14]. We also show that the GMW compiler [15], outputs a commit-first protocol. This is of theoretical interest as it provides a general compiler for transforming a semi-honest SFE protocol into a malicious cf-SFE (and eventually a rate-limited SFE using the compilers in this paper). We elaborate on these cf-SFE instantiations in the full version of this paper [11].

Compilers & Techniques. We design three compilers for transforming a cf-SFE into each of the three variants of RL-SFE discussed above, and provide simulation-based proofs of their security. All our compilers start from a cf-SFE protocol and add a "proof of repeated-input phase" between the committing phase and the function evaluation phase. An exception is our pattern-revealing compiler, where a proof of repeated-input is implicit given that we force the commitments to be deterministic. In our first compiler (rate-hiding), whenever the $j$-th execution begins, party $P_1$ first checks whether its input is "fresh" or has

[3] Note that adding input commitments to the beginning of a protocol does not automatically yield a cf-SFE, since parties are not necessarily bound to using the committed inputs in their evaluation.

already been used in a previous run. In the former case, $P_1$ encrypts the value "1" and, otherwise, the value "0" using a semantically secure public-key encryption scheme $(\mathsf{E}, \mathsf{D})$ for which it holds the secret key $\mathsf{sk}$. Denote the resulting ciphertext with $c^j$. Party $P_1$ forwards to $P_2$ a ZK proof of the following statement:

$$\begin{aligned}(&\text{"committed to old input"} \wedge\ \mathsf{E}(0)\\ &\vee (\text{"committed to new input"} \wedge \mathsf{E}(1) \wedge \text{"}\textstyle\sum_{i \le j} \mathsf{D}(\mathsf{sk}, c^i) \le \text{rate"}).\end{aligned}$$

Intuitively, the proof above only leaks the fact that the rate is not exceeded in the current execution, but nothing else. In order to generate this proof (resp. verify the proof generated by the counterpart), $P_1$ needs to store all the commitments and ciphertexts sent to (resp. received from) $P_2$ in previous executions.

For our second compiler (rate-revealing), we can do without the encryptions. Parties can instead prove a simpler statement giving evidence that the current (committed) input corresponds to one of the commitments the other party received earlier. Clearly, this approach reveals the current rate, but as we prove nothing more.

Finally, our third compiler (pattern-revealing) exploits a PRF to generate the randomness used in the committing phase of the underlying cf-SFE protocol. In this way, the commitment becomes deterministic (given the input), allowing the other party to check whether the current input has already been used and *in which runs*. This approach discloses the pattern of inputs used by the parties; on the other hand, it is extremely efficient adding little computational overhead (merely one invocation of a PRF) to the original cf-SFE protocol.

Making RL-SFE stateless. The above compilers suffer from the limitation that the parties need to keep a state which grows linearly in the total number of executions of the underlying SFE protocol. In many applications, clients do not keep state (and outsource this task to the servers), either due to lack of resources or because they need to use the service from multiple locations/devices. We show a general approach for transforming the stateful RL-SFE protocols generated above into *stateless* ones. Here, the client keeps merely a small secret (whose size is independent of the total number of executions), but is still able to prevent cheating by a malicious server, and preserve privacy of his inputs. At a high level, the transformation requires the client to store its *authenticated* (MACed) state information on the server side and retrieve/verify/update it on-the-fly as needed. We show how to apply this transformation to our rate-revealing compiler to obtain a stateless variant and prove its security. A similar technique can be applied to our rate-hiding compiler. Our pattern-revealing compiler is already stateless (client only needs to store a PRF key) for the party who plays the role of the client.

Case Study. We take a closer look at the oblivious polynomial evaluation protocol of Hazay and Lindell [16]. Their protocol is secure against malicious adversaries. We show that it is also a commit-first OPE, by observing that a homomorphic encryption of the parties' inputs together with ZK proofs of their validity, can be interpreted as a commitment to their inputs. This immediately

yields an efficient pattern-revealing RL-SFE for the OPE problem, based on the compiler we design. We also provide an efficient rate-hiding and rate-revealing RL-OPE by instantiating the ZK proofs for membership in the necessary languages, efficiently.

**Roadmap.** We start introducing notations and our model for commit-first SFE in Section 2 and 3. In Section 4 we give the definition of rate-limited SFE. A fortaste of our rate-hiding, rate-revealing and pattern-revealing compilers are given in Section 5, whereas Section 6 describes the stateless version of the rate-revealing compiler. Finally, Section 7 deals with concrete instantiations for the case of OPE.

## 2 Preliminaries

Throughout the paper, we denote the security parameter by $\lambda$. A function $negl(\lambda)$ is negligible in $\lambda$ (or just negligible) if it decreases faster than the inverse of every polynomial in $\lambda$. A machine is said to run in polynomial-time if its number of steps is polynomial in the security parameter.

Let $X = \{X_\lambda\}_{\lambda\in\mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda\in\mathbb{N}}$ be two distribution ensembles. We say $X$ and $Y$ are computationally indistinguishable (and we write $X \equiv_c Y$) if for every non-uniform polynomial-time adversary $\mathcal{A}$ there exists a negligible function $negl$ such that $|\Pr[\mathcal{A}(X) = 1] - \Pr[\mathcal{A}(Y) = 1]| \leq negl(\lambda)$. Note that all our security statements can be straightforwardly proven for uniform polynomial-time adversaries, as well.

If $x$ is a string, $|x|$ denotes the length of $x$. Vectors are denoted boldface; given vector $\mathbf{x}$, we write $\mathbf{x}[j]$ for the $j$-th element of $\mathbf{x}$. If $\mathcal{X}$ is a set, $\#\mathcal{X}$ represents the number of elements in $\mathcal{X}$. When $x$ is chosen randomly in $\mathcal{X}$, we write $x \leftarrow \mathcal{X}$. When $\mathcal{A}$ is an algorithm, $y \leftarrow \mathcal{A}(x)$ denotes a run of $\mathcal{A}$ on input $x$ and output $y$; if $\mathcal{A}$ is randomized, then $y$ is a random variable and $\mathcal{A}(x; r)$ denotes a run of $\mathcal{A}$ on input $x$ and random coins $r$.

Our compilers make use of standard cryptographic primitives. Due to space limitations, we assume familiarity of these primitives and define them formally in the full version of this paper [11].

## 3 Commit-First SFE

In this section, we formally define the notion of *commit-first secure function evaluation* (cf-SFE). Our three compilers $\Psi_{\mathsf{RH}}$, $\Psi_{\mathsf{RR}}$ and $\Psi_{\mathsf{PR}}$ for designing rate-limited SFE, leverage commit-first protocols as a building block. We call a protocol $\pi$ commit-first if it can be naturally divided into two phases. In the first phase (committing phase), both parties $P_1$ and $P_2$ become committed to their inputs. At the end of this phase, no information about the parties' inputs is revealed (the hiding property), and neither party can use a different input than what it is committed to in the remainder of the protocol (the binding property). In the

second phase (function evaluation phase), the function $f$ will be computed on the inputs committed to in the last phase.

We now specify the two separate phases. Consider a polynomial-time functionality $f = (f_1, f_2)$ with $f_i : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$. Then, a cf-SFE protocol $\pi = (\pi_1, \pi_2)$ for evaluating $f$ on parties' inputs $x_1$ and $x_2$ proceeds as follows.

**Committing Phase:** Parties $P_1$ and $P_2$ execute $\pi_1$ which is defined by the functionality $((x_1, r_1), (x_2, r_2)) \mapsto (\mathsf{C}_2(x_2, r_2), \mathsf{C}_1(x_1, r_1))$. Note that the commitment schemes $\mathsf{C}_1, \mathsf{C}_2$ can be arbitrary schemes (often different for each cf-SFE protocol), as long as they satisfy the hiding and the binding properties required.

**Function Evaluation Phase:** Afterwards, $P_1$ and $P_2$ execute $\pi_2$ on the same inputs as in the committing phase; $\pi_2$ is defined by the functionality $((x_1, \mathsf{C}_2(x_2)), (x_2, \mathsf{C}_1(x_1))) \mapsto (f_1(x_1, x_2), f_2(x_1, x_2))$. Note that $P_1$ and $P_2$, can use their state information from the previous phase in the function evaluation phase, too.

Next, we formalize the security definition for a cf-SFE using the real/ideal world simulation paradigm.

THE REAL WORLD. In each execution, a non-uniform adversary $\mathcal{A}$ following an arbitrary polynomial-time strategy can send messages in place of the corrupted parties (whereas the honest parties continue to follow $\pi$). Let $i \in \{1, 2\}$ be the index of the corrupted party. A real execution of $\pi = (\pi_1, \pi_2)$ on inputs $(x_1, x_2)$, auxiliary input $z$ to $\mathcal{A}$ and the security parameter $\lambda$, denoted by $\text{REAL}^{\mathsf{cf-SFE}}_{\pi,\mathcal{A}(z),i}(x_1, x_2, \lambda)$ is defined as the output of the honest party and the adversary upon execution of $\pi$.

THE IDEAL WORLD. Let $i \in \{1, 2\}$ be the index of the corrupted party. We define the ideal world in two steps. During the ideal execution, the honest party sends its input $x_{3-i}$, and a uniformly random string $r_{3-i}$ used by the commitment scheme, to the trusted party. Party $P_i$ which is controlled by the ideal adversary $\mathcal{S}$, called the simulator, may either abort (sending a special symbol $\perp$) or send input $x'_i$, and an arbitrary randomness $r'_i$ (not necessarily uniform) chosen based on the auxiliary input $z$, and $P_i$'s original input $x_i$. Denote by $((x'_1, r'_1), (x'_2, r'_2))$ the values received by the trusted party. If the trusted party receives $\perp$, the value $\perp$ is forwarded to both $P_1$ and $P_2$ and the ideal execution terminates; else the trusted party computes $\gamma_1 = \mathsf{C}_1(x'_1; r'_1)$ and $\gamma_2 = \mathsf{C}_2(x'_2; r'_2)$, respectively. The TTP sends $\gamma_{3-i}$ to $\mathcal{S}$, which can either continue or abort by sending $\perp$ to the TTP. In case of an abort, the TTP sends $\perp$ to the honest party; otherwise, it sends $\gamma_i$.

In the second phase, the honest party continues the ideal execution by sending to the TTP a continue flag, or abort by sending $\perp$. $\mathcal{S}$ sends either $\perp$ or continue based on the auxiliary input $z$, $P_i$'s original input, and the value $\gamma_{3-i}$. If the trusted party receives $\perp$, the value $\perp$ is forwarded to both $P_1$ and $P_2$ and the ideal execution terminates; else the trusted party computes $y_1 = f_1(x'_1, x'_2)$ (resp. $y_2 = f_2(x'_1, x'_2)$).

The TTP sends $y_i$ to $\mathcal{S}$. At this point, $\mathcal{S}$ can decide whether the trusted party should continue, and thus send the output $y_{3-i}$ to the honest party, or halt, in which case the honest party receives $\perp$. The honest party outputs the received value. The simulator $\mathcal{S}$ outputs an arbitrary polynomial-time computable function of $(z, x_i, y_i)$.

The ideal execution of $f$ on inputs $(x_1, x_2)$, auxiliary input $z$ to $\mathcal{S}$ and security parameter $\lambda$, denoted by $\text{IDEAL}^{\mathsf{cf-SFE}}_{f,\mathsf{C}_1,\mathsf{C}_2,\mathcal{S}(z),i}(x_1, x_2, \lambda)$ is defined as the output of the honest party and the simulator.

EMULATING THE IDEAL WORLD. We define a secure commit-first protocol $\pi$ as follows:

**Definition 1 (Commit-first Protocols).** *Let $\pi$ and $f$ be as above. We say that $\pi$ is* a commit-first protocol for computing $f = (f_1, f_2)$ in the presence of malicious adversaries with abort *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real world there exists a non-uniform probabilistic polynomial-time simulator $\mathcal{S}$ in the ideal world, such that for every $i \in \{1, 2\}$,*

$$\left\{\text{REAL}^{\mathsf{cf-SFE}}_{\pi,\mathcal{A}(z),i}(x_1, x_2, \lambda)\right\}_{x_1,x_2,z,\lambda} \equiv_c \left\{\text{IDEAL}^{\mathsf{cf-SFE}}_{f,\mathsf{C}_1,\mathsf{C}_2,\mathcal{S}(z),i}(x_1, x_2, \lambda)\right\}_{x_1,x_2,z,\lambda}$$

*where $x_1, x_2, z \in \{0,1\}^*$ and $\lambda \in \mathbb{N}$.*

## 4 Rate-Limited Secure Function Evaluation

In this section, we introduce three notions for rate-limited secure function evaluation (RL-SFE). In particular, we augment the standard notion of two-party SFE by allowing each player to monitor and/or limit, the number of distinct inputs (the *rate*) the other player uses in multiple executions. The idea is that each party can abort the protocol if the number of distinct inputs used in the previous executions raises above a threshold $\nrightarrow \in \mathbb{N}$. We call this threshold the *rate limit*, i.e. the maximum number of allowable executions with distinct inputs.

Naturally, our security definitions for RL-SFE are concerned with *multiple* executions of an SFE protocol and reduce to the standard simulation-based definition (stand alone) for SFE, when applied to a single run. We call a sequence of executions of a protocol $\pi$ $(\nrightarrow_1, \nrightarrow_2)$-limited if party $P_1$ (resp. $P_2$) can use at most $\nrightarrow_1$ (resp. $\nrightarrow_2$) distinct inputs in the executions. In this work, we assume that the executions take place *sequentially*, i.e. one execution after the other. We emphasize that the inputs used by the parties in each execution can depend on the transcripts of the previous executions, but honest parties will always use fresh randomness in their computation.

We provide three security definitions for rate-limited SFE: (i) rate-hiding, (ii) rate-revealing and (iii) pattern-revealing. In a *rate-hiding* RL-SFE, at the end of each execution, the only information revealed to the parties (besides the output from the function being computed), is whether the agreed-upon rate limit (threshold) has been exceeded or not, but nothing else. In a *rate-revealing* RL-SFE, in addition to the above, parties also learn the current rates (i.e.,

the number of distinct inputs used by their counterpart so far). Finally, in a *pattern-revealing* RL-SFE, parties further learn the pattern of occurrences of each others' inputs in the previous executions. In particular, each party learns which executions were invoked by the same input and which ones used different ones, but nothing else.

HIGH LEVEL DESCRIPTION. Let $f = (f_1, f_2)$ be a pair of polynomial-time functions such that $f_i$ is of type $f_i : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$. Consider an arbitrary number $\ell$ of sequential executions of two-party SFE protocol $\pi$ for evaluating $f$ on parties' inputs. During the $j$-th execution, party $P_i$ has input $x_i^j$ and should learn $y_i^j = f_i(x_1^j, x_2^j)$. We will define rate-limited SFE in the general case where *both* parties are allowed to change their input in each execution. The case of oracle attacks and secure metering, where one party's input is fixed and the other party's input changes, are found as a special case. (In the case of secure metering one can also think that a change in the service provider's input reflects a software update.)

In the ideal world, during the $j$-th execution, each party sends its input to a trusted authority. The following is then performed for both $i = 1, 2$. The trusted party checks whether value $x_i^j$ was already sent in a previous execution; in case it was not, a new entry $(x_i^j, j)$ is stored in an initially empty set $\mathcal{X}_i$. Otherwise, the index $j' < j$ corresponding to such input is recovered. Whenever $\#\mathcal{X}_i$ exceeds $\wedge_i$ the trusted party aborts. Otherwise, the current outputs $y_i^j = f_i(x_1^j, x_2^j)$ are computed. Finally: (i) in the rate-hiding definition party $P_i$ learns only $y_i^j$; (ii) in the rate-revealing definition party $P_i$ learns also $\#\mathcal{X}_{3-i}$, i.e. the (partial) total number of distinct inputs used by $P_{3-i}$ until the $j$-th execution; (iii) in the pattern-revealing definition party $P_i$ learns $j'$, i.e. the index corresponding to the query where $x_i^j$ was asked for the first time. Note that if the rate is exceeded, the trusted party aborts here, but, equivalently, we could simply ignore this execution and still allow to query previous inputs in subsequent executions.

We formalize the above intuitive security notions for all three flavors using the simulation-based ideal/real world paradigm. We first review the real execution which all three notions share.

THE REAL WORLD. In each execution, a non-uniform adversary $\mathcal{A}$ following an arbitrary polynomial-time strategy can send messages in place of the corrupted party (whereas the honest party continues to follow $\pi$). Let $i \in \{1, 2\}$ be the index of the corrupted party. The $j$-th real execution of $\pi$ on inputs $(x_1^j, x_2^j)$, auxiliary input $z^j$ to $\mathcal{A}$ and security parameter $\lambda$, denoted by $\text{REAL}^{\wedge}_{\pi,\mathcal{A}(z^j),i}(x_1^j, x_2^j, \lambda)_j$ is defined as the output of the honest party and the adversary in the $j$-th real execution of $\pi$. We denote by $\text{REAL}^{\wedge}_{\pi,\mathcal{A}(\mathbf{z}),i}(\mathbf{x}_1, \mathbf{x}_2, \lambda, \ell)$ the accumulative distribution at the end of the $\ell$-th execution, i.e.,

$$\text{REAL}^{\wedge}_{\pi,\mathcal{A}(\mathbf{z}),i}(\mathbf{x}_1, \mathbf{x}_2, \lambda, \ell) = \text{REAL}^{\wedge}_{\pi,\mathcal{A}(z^1),i}(x_1^1, x_2^1, \lambda)_1, \ldots, \text{REAL}^{\wedge}_{\pi,\mathcal{A}(z^\ell),i}(x_1^\ell, x_2^\ell, \lambda)_\ell$$

where $\mathbf{x}_1 = (x_1^1, \ldots, x_1^\ell)$, $\mathbf{x}_2 = (x_2^1, \ldots, x_2^\ell)$ and $\mathbf{z} = (z^1, \ldots, z^\ell)$.

THE IDEAL WORLD. The trusted party keeps two sets $\mathcal{X}_1$, and $\mathcal{X}_2$ initially set to $\emptyset$. Let $i \in \{1, 2\}$ be the index of the corrupted party. During the $j$-th ideal

execution, the honest party sends its input to the trusted party. Party $P_i$, which is controlled by the ideal adversary $\mathcal{S}$, called the simulator, may either abort (sending a special symbol $\perp$) or send input $x_i'^j$ to the trusted party chosen based on the auxiliary input $z^j$, $P_i$'s original input $x_i^j$, and its view in the previous $j-1$ ideal executions. Denote with $(x_1'^j, x_2'^j)$ the values received by the trusted party (note that if $i = 2$ then $x_1'^j = x_1^j$).

If the trusted party receives $\perp$, the value $\perp$ is forwarded to both $P_1$ and $P_2$ and the ideal execution terminates; else when the trusted party receives $x_1'^j$ as the first party's input, it checks whether an entry $(x_1'^j, j') \in \mathcal{X}_1$ already exists; if so, it sets $J_1 = j'$. Otherwise, it creates a new entry $(x_1'^j, j)$, adds it to $\mathcal{X}_1$, and sets $J_1 = j$. An identical procedure is applied to input of the second party $x_2'^j$ to determine an index $J_2$. At the end of the $j$-th ideal execution if $\sigma_1 := \#\mathcal{X}_1 \geq \nu_1$ *or* $\sigma_2 := \#\mathcal{X}_2 > \nu_2$, the value $\perp$ is forwarded to both $P_1$ and $P_2$ and the ideal execution terminates. Otherwise, the pair $(y_1^j, y_2^j) = (f_1(x_1'^j, x_2'^j), f_2(x_1'^j, x_2'^j))$ is computed.

At this point, the ideal executions will be different depending on the variant of RL-SFE being considered.

**Rate-Hiding** The trusted party forwards to the malicious party $P_i$ the output $y_i^j$. At this point, $\mathcal{S}$ can decide whether the trusted party should continue, and thus send the pair $y_{3-i}$ to the honest party, or halt, in which case the honest party receives $\perp$.

**Rate-Revealing** The trusted party forwards to the malicious party $P_i$ the pair $(y_i^j, \sigma_{3-i})$. At this point, $\mathcal{S}$ can decide whether the trusted party should continue, and thus send the pair $(y_{3-i}^j, \sigma_i)$ to the honest party, or halt, in which case the honest party receives $\perp$.

**Pattern-Revealing** The trusted party forwards to the malicious party $P_i$ the pair $(y_i^j, J_{3-i})$. The integer $1 \leq J_{3-i} \leq j$ represents the index of the first execution where the input $x_{3-i}^j$ has been used. At this point, $\mathcal{S}$ can decide whether the trusted party should continue, and thus send the pair $(y_{3-i}^j, J_i)$ to the honest party, or halt, in which case the honest party receives $\perp$.

The honest party outputs the received value. The simulator $\mathcal{S}$ outputs an arbitrary polynomial-time computable function of $(z^j, x_i^j, y_i^j)$.

The $j$-th ideal execution of $f$ on inputs $(x_1^j, x_2^j)$, auxiliary input $z^j$ to $\mathcal{S}$ and security parameter $\lambda$, denoted by $\text{IDEAL}^{\nu-\mathsf{X}}_{f,\mathcal{S}(z^j),i}(x_1^j, x_2^j, \lambda)_j$ is defined as the output of the honest party and the simulator in the above $j$-th ideal execution. Here, $\mathsf{X} \in \{\mathsf{RH}, \mathsf{RR}, \mathsf{PR}\}$ determines the flavor of rate-limited SFE. We denote by $\text{IDEAL}^{\nu-\mathsf{X}}_{f,\mathcal{S}(\mathbf{z}),i}(\mathbf{x}_1, \mathbf{x}_2, \lambda, \ell)$ the accumulative distribution at the end of the $\ell$-th execution, i.e.,

$$\text{IDEAL}^{\nu-\mathsf{X}}_{f,\mathcal{S}(\mathbf{z}),i}(\mathbf{x}_1, \mathbf{x}_2, \lambda, \ell) = \text{IDEAL}^{\nu-\mathsf{X}}_{f,\mathcal{S}(z^1),i}(x_1^1, x_2^1, \lambda)_1, \ldots, \text{IDEAL}^{\nu-\mathsf{X}}_{f,\mathcal{S}(z^\ell),i}(x_1^\ell, x_2^\ell, \lambda)_\ell$$

where $\mathbf{x}_1 = (x_1^1, \ldots, x_1^\ell)$, $\mathbf{x}_2 = (x_2^1, \ldots, x_2^\ell)$ and $\mathbf{z} = (z^1, \ldots, z^\ell)$.

EMULATING THE IDEAL WORLD. Roughly speaking, $\ell$ sequential executions of a protocol $\pi$ are secure under the rate limit $\nu = (\nu_1, \nu_2)$ if the real executions

can be simulated in the above mentioned ideal world. More formally, we define a secure $(\pi_1, \pi_2)$-limited protocol $\pi$ as follows:

**Definition 2 (RL-SFE).** *Let $\pi$ and $f$ be as above, and consider $\ell = poly(\lambda)$ sequential executions of protocol $\pi$. For $\mathsf{X} \in \{\mathsf{RH}, \mathsf{RR}, \mathsf{PR}\}$, we say protocol $\pi$ is a* secure $\mathsf{X}$ $\pi$-limited SFE *for computing $f = (f_1, f_2)$,* in presence of malicious adversaries with abort *with $\pi = (\pi_1, \pi_2)$, if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ there exists a non-uniform probabilistic polynomial-time simulator $\mathcal{S}$, such that for every $i \in \{1, 2\}$,*

$$\left\{\mathrm{REAL}^{\pi}_{\pi,\mathcal{A}(\mathbf{z}),i}(\mathbf{x}_1, \mathbf{x}_2, \lambda, \ell)\right\}_{\mathbf{x}_1,\mathbf{x}_2,\mathbf{z},\lambda} \equiv_c \left\{\mathrm{IDEAL}^{\pi-\mathsf{X}}_{f,\mathcal{S}(\mathbf{z}),i}(\mathbf{x}_1, \mathbf{x}_2, \lambda, \ell)\right\}_{\mathbf{x}_1,\mathbf{x}_2,\mathbf{z},\lambda}$$

*where $\mathbf{x}_1, \mathbf{x}_2, \mathbf{z} \in (\{0,1\}^*)^\ell$, such that $|\mathbf{x}_1[j]| = |\mathbf{x}_2[j]|$ for all $j$, and $\lambda \in \mathbb{N}$.*

It is easy to see that the rate-hiding notion is strictly stronger than the rate-revealing notion, which in turn is strictly stronger than the pattern-revealing notion. A proof to this fact can be found in the full version [11].

## 5 Compilers for Rate-Limited SFE

In this section, we introduce our three compilers to transform an arbitrary (two-party) cf-SFE protocol into a *rate-limited* protocol for the same functionality.

Our first compiler $\Psi_{\mathsf{RH}}$ achieves the notion of rate-hiding RL-SFE through the use of general ZK proofs and (additively) homomorphic public key encryption. Our second compiler $\Psi_{\mathsf{RR}}$ achieves the notion of rate-revealing RL-SFE and is more efficient in that it needs to prove a simpler statement and does not rely on homomorphic encryption. Our last compiler $\Psi_{\mathsf{PR}}$ introduces essentially no overhead and avoids the use of general ZK proofs, yielding our third notion of pattern-revealing RL-SFE.

Let $\pi_f$ be a two-party (single-run) commit-first protocol for secure function evaluation of a function $f = (f_1, f_2)$ (cf. Definition 1). Our compilers get as input (a description of) $\pi_f$, together with the rate $\pi = (\pi_1, \pi_2)$, and the number of executions $\ell$, and output (a description of) $\widehat{\pi}_f \leftarrow \Psi(\pi_f, \pi, \ell)$. The compilers are functionality preserving, meaning that protocol $\widehat{\pi}$ repeatedly computes the same functionality $f$.

Due to space limitations, in this section we only provide a full description and analysis for the rate-revealing compiler. The other two compilers (rate-hiding $\Psi_{\mathsf{RH}}$, and pattern-revealing $\Psi_{\mathsf{PR}}$) are only covered at a high level here. The complete descriptions and analyses are given in the full version of this paper [11].

### 5.1 A Rate-Hiding Compiler

The Overview. We naturally divide the cf-SFE protocol into a committing phase and a function evaluation phase and introduce a new phase in between where $P_1$ and $P_2$ convince each other that they have not exceeded the rate limit.

The latter step is achieved as follows. Whenever one of the parties is going to use a "fresh" input, it transmits an encryption of "1" to the other party; otherwise, it sends an encryption of "0". The encryptions are obtained using a CPA-secure (homomorphic) PKE scheme $(\tilde{\mathsf{G}}, \tilde{\mathsf{E}}, \tilde{\mathsf{D}})$. Then, the party proves in ZK that "the last commitment transmitted hides an already used input *and* it encrypted 0, *or* the last commitment transmitted hides a fresh input *and* it encrypted 1 *and* the sum of all the plaintexts, encrypted until now, does not exceed the rate". A successful verification of this proof convinces the other party that the rate is not exceeded, leaking nothing more than this. We instantiate such ZK proofs for the OPE problem in Section 7. Notice that to generate such a proof each party needs to store all the ciphertexts transmitted to the other player, together with all the inputs and randomness used to generate the previous commitments. On the other hand, to verify the other party's proof, one needs to store the ciphertexts and the commitments received in all earlier executions. The remainder of the messages exchanged during each execution, however, can be discarded.

**Theorem 1.** *Let $\pi_f$ be a cf-SFE securely evaluating function $f$ and $(\tilde{\mathsf{G}}, \tilde{\mathsf{E}}, \tilde{\mathsf{D}})$ be a CPA-secure PKE scheme. Then $\widehat{\pi}_f \leftarrow \Psi_{\mathsf{RH}}(\pi_f, \hbar_1, \hbar_2, \ell)$ is a secure* rate-hiding *$(\hbar_1, \hbar_2)$-limited protocol for the function $f$.*

### 5.2 A Rate-Revealing Compiler

The Overview. Once again, we divide the cf-SFE protocol into a committing phase and a function evaluation phase and introduce a new phase in between where $P_1$ and $P_2$ convince each other that the current input has already been used in a previous execution. Note that the parties need to maintain a state variable $\Gamma$ collecting the input commitments sent and received in all earlier executions. During the $j$-th execution, given a list of input commitments (and the corresponding inputs and randomness) for all the previous executions, party $P_i$ can prove in ZK that the input commitment generated in the current execution is for the same value as one of the commitments collected previously. Party $P_{3-i}$ also needs to collect the same set of commitments in order to verify the statement proven by $P_i$. The remainder of the messages exchanged during each execution, however, can be discarded. We note that while in general efficient ZK proofs of repeated inputs might be hard to find, for discrete-logarithm based statements, there exist efficient techniques for proving such statements. We refer the reader to the full version for more details. We also instantiate such ZK proofs for the OPE problem in Section 7. A complete description of the compiler is depicted in Figure 1. We prove the following result:

**Theorem 2.** *Let $\pi_f$ be a cf-SFE securely evaluating function $f$. Then $\widehat{\pi}_f \leftarrow \Psi_{\mathsf{RR}}(\pi_f, \hbar_1, \hbar_2, \ell)$ is a secure* rate-revealing *$(\hbar_1, \hbar_2)$-limited protocol for $f$.*

### 5.3 A Pattern-Revealing Compiler

In this section, we introduce a more efficient compiler $\Psi_{\mathsf{PR}}$ for designing rate-limited SFE. Given as input a cf-SFE protocol, our compiler $\Psi_{\mathsf{PR}}$ outputs a

**Rate-Revealing Compiler $\Psi_{\mathsf{RR}}$:**

Given as input a commit-first protocol $\pi_f = (\pi_f^1, \pi_f^2)$, a rate $\nwarrow = (\nwarrow_1, \nwarrow_2)$, and a number of executions $\ell$, the compiled protocol $\widehat{\pi}_f$ is made of three phases, described below. Party $P_1$ and $P_2$ keep the state variables $\Gamma_1, \Gamma_2 := \emptyset$, respectively. For each execution $j \in [\ell]$, $\widehat{\pi}_f$ proceeds as follows.

**Committing Phase:** Parties $P_1$ and $P_2$, holding respectively inputs $x_1^j$ and $x_2^j$, run the protocol $\pi_f^1$ yielding the output $(\gamma_2^j = \mathsf{C}(\mathsf{pk}_2, x_2^j; r_2^j), \gamma_1^j = \mathsf{C}(\mathsf{pk}_1, x_1^j; r_1^j))$.

**Proof of Repeated-Input Phase:** Consider the following language: $\mathcal{L}_i = \{\gamma \in \mathcal{C}_{\mathsf{pk}_i} : \exists (x, r, r') \text{ s.t. } \gamma = \mathsf{C}(\mathsf{pk}_i, x; r) \wedge \mathsf{C}(\mathsf{pk}_i, x; r') \in \Gamma_{3-i}\}$, and let $(\mathsf{Pr}, \mathsf{Vr})$ be a ZK proof system for $\mathcal{L}_i$. The following is executed for all $i \in \{1, 2\}$. When the input $x_i^j$ of party $P_i$ is *not* fresh—i.e., it has already been used in a previous execution—$P_i$ plays the role of the prover in $(\mathsf{Pr}, \mathsf{Vr})$. When the input $x_i^j$ is fresh, $P_i$ just forwards the empty string $\varepsilon$. Also, party $P_i$ plays the role of the verifier in $(\mathsf{Pr}, \mathsf{Vr})$ (with $P_{3-i}$ being the prover and $\mathcal{L}_{3-i}$ being the underlying language). If the value $\varepsilon$ is received or if the verification of the proof fails, $P_i$ updates the rate by letting $\nwarrow_i := \nwarrow_i - 1$ and the state by letting $\Gamma_i := \Gamma_i \cup \{\gamma_{3-i}^j\}$. Otherwise, if the verification is successful, the state and rate information will not be modified.

**Protocol Emulation Phase:** $P_1$ and $P_2$ run the protocol $\pi_f^2$ on the same inputs as in the committing phase, yielding the output $(y_1^j, y_2^j)$.

Fig. 1: A compiler for rate-revealing rate-limited SFE.

weaker form of rate-limited SFE where each party not only learns the current rate for its counterpart during each execution, but also the pattern of already used inputs. The main advantage is that this new compiler adds very little overhead to the original cf-SFE.

The Overview. The idea is as follows. Besides their input, each party also stores a secret key for a PRF (a different key for each party). Before invoking the commit-first SFE protocol, each player generates the randomness it needs for the committing phase by applying the PRF on the *chosen input* for this execution. With this modification in place, the committing phase for each party becomes deterministic. If a party uses the same input in two executions, the two commitments its counterpart receives will be identical. As a result, to prove a repeated-input, each party can compare the commitment for the current execution with those used in the previous ones, and determine if the input is new or being repeated (hence also revealing the pattern). Note that the commitments still provide the required hiding and binding properties. The only overhead imposed by this compiler is the application of a PRF to generate the randomness for the committing phase.

**Theorem 3.** *Let $\pi_f$ be a cf-SFE securely evaluating function $f$. Then $\widehat{\pi}_f \leftarrow \Psi_{\mathsf{PR}}(\pi_f, \nwarrow_1, \nwarrow_2, \ell)$ is a secure* pattern-revealing $(\nwarrow_1, \nwarrow_2)$*-limited SFE for $f$.*

## 6 Making the Compilers Stateless

One drawback of the compilers described in the previous section is that both $P_1$ and $P_2$ need to maintain state. To some extent, this assumption is necessary. It is not too hard to see that RL-SFE is impossible to achieve if neither party is keeping any information about the previous executions (we omit a formal

argument of this statement). However, as discussed earlier, in many natural client-server applications of SFE in the real world, it is reasonable to assume that the servers keep state, while the clients typically do not.

In this section, we show how to modify the compilers from Section 5 in such a way that only one of the parties needs to keep state. Our solution is efficient and works for all three compilers we discussed earlier. Throughout this section, we assume $P_1$ is the client and $P_2$ is the server. Server $P_2$ receives no output (as it is usually the case in the client-server setting) and wants to enforce the rate limit $\wr$ for the client. Although $P_1$ does not maintain any state, it needs to make sure that $P_2$ handles the rate, honestly. On the other hand, the server also needs to be convinced that the client is not cheating, by exceeding the rate limit $\wr$.

THE OVERVIEW. Note that in the stateful versions of our compilers, $P_1$ needs to keep state in order to generate a ZK proof of repeated inputs, and verify the corresponding statement being proven by $P_2$. Since we are only enforcing the rate for $P_1$, we can eliminate the latter ZK proofs, and focus on the first one. Although our approach is general, for the sake of simplicity, we describe it in relation to our rate-revealing compiler from Section 5.2. The same idea can be applied to make the other compilers stateless. The basic idea is simple: We ask the server to store the list of all the commitments previously sent by $P_1$ sends the list to the client, during each run. For this simple approach to work, we need to address several important issues:

- For the client to learn the current rate and the previously queried inputs before each execution, it needs to store these values on the server side in a secure way. This can be easily addressed by having $P_1$ encrypt the message and randomness for each commitment (using a symmetric-key encryption) and send it along with the commitment itself. $P_1$ will just keep the private key for the encryption scheme.
- The client needs to verify that the list of commitments it receives from the server are the original commitments it sent in the previous executions. To do so, in each run $P_1$ computes a MAC $\phi$ of the string obtained by hashing all the commitments (i.e., the concatenation of the list it obtains from the server and the one it creates in the current execution) and sends it to the server.[4] In each execution, it requests this MAC, the list of commitments along with the ciphertext storing the inputs and random coins from the server. Due to the unforgeability of the MAC, the server will only be able to use a correct list of commitments, previously issued and MACed by the client itself.
- It may seem that the above solution still allows the server to cheat and only send a subset of the commitment list along with a tag generated for that subset in one of the earlier executions, to the client. This would potentially make the input to the current execution look "new" and allow the server to decrease the rate. The client would not be able to detect this attack since it does not keep state and does not know the total number of commitments.

[4] To save on computation, one could let the client obtain the previous hash value and compute the new one via *incremental* hashing [9,4].

However, a more careful inspection shows that the above does not really constitute an attack. In fact, the tag $\phi$ already *binds* the current rate to the current list of commitments, and prevents the server from decreasing the rate in this fashion. In particular, it is hard for the server to cook-up a state such that the verification of the tag is successful, and the client will think its rate is already exceeded when it is not. Essentially, coming up with such a state requires to find a collision in $H$ or forging a tag for a fake list of commitments.

A detailed description of the compiler $\Psi_{\mathsf{RR}}$ and a proof of the following theorem are given in our full version [11].

**Theorem 4.** *Let $\pi_f$ be a cf-SFE securely evaluating function $f = (f_1, -)$ and $(\mathsf{G}, \mathsf{T}, \mathsf{V})$ be a* UNF-CMA *MAC scheme, $(\tilde{\mathsf{G}}, \tilde{\mathsf{E}}, \tilde{\mathsf{D}})$ be a* CPA*-secure PKE scheme, and $H$ being picked from a family of CRHFs. Then, $\widehat{\pi}_f \leftarrow \Psi_{\mathsf{RR}}(\pi_f, \text{ɲ}, \ell)$ is a secure* rate-revealing *ɲ-limited protocol for $f$.*

## 7 Rate-Limited OPE

Hazay and Lindell [16] design an efficient two-party protocol for oblivious polynomial evaluation (OPE) with security against malicious adversaries. In an OPE protocol, the first party holds a value $t$ while the second party holds a polynomial $p$ of degree $d$. Their goal is to let the first party learn $p(t)$ without revealing anything else. The protocol takes advantage of an additively homomorphic encryption scheme (Paillier's encryption) and efficient ZK proofs of a few statements related to the encryption scheme. While the authors (only) prove security against malicious adversaries, we observe that, with a small modification, their construction is indeed a commit-first protocol for OPE as well.

First party's commitment. Consider an additively homomorphic encryption scheme $(\mathsf{G}, \mathsf{E}, \mathsf{D})$. The first few steps performed by the first party (the party holding the value $t$) are as follows: (i) it runs the key generation for the encryption scheme to generate a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{G}(1^\lambda)$, accompanied by a ZK proof of knowledge of the secret key; (ii) then, it encrypts powers of $t$, i.e. $\mathsf{E}(\mathsf{pk}, t), \mathsf{E}(\mathsf{pk}, t^2), \ldots, \mathsf{E}(\mathsf{pk}, t^d)$, and sends the resulting ciphertexts along with a ZK proof of the validity of the ciphertexts to the other party.

We observe that sending $\mathsf{E}(\mathsf{pk}, t)$ and a ZK proof of its validity constitutes a commitment by the first party to its input $t$. This commitment scheme realizes the ideal functionality of the first phase in our definition of commit-first protocols. (Recall that this means the simulator can extract both the input and the randomness used to generate the commitment.) In particular, a careful inspection of the security proof of [16] reveals that the simulator can extract both $t$ and the randomness used to encrypt it during the simulation. Extracting the randomness is possible since in Paillier's encryption scheme, given the secret key $\mathsf{sk}$ and a ciphertext $c$, one can recover both the randomness and the message.

Second party's commitment. The commitment of the second party to its input polynomial is slightly more subtle, and requires a small modification to the

original design. In the first few steps, the second party does the following: (i) it runs the key generation to generate a key pair $(\mathsf{pk}', \mathsf{sk}') \leftarrow \mathsf{G}(1^\lambda)$, accompanied by a ZK proof of knowledge of the secret key; (ii) it computes $((\mathsf{E}(\mathsf{pk}', q_1), \mathsf{E}(\mathsf{pk}', p - q_1)), \ldots, (\mathsf{E}(\mathsf{pk}', q_s), \mathsf{E}(\mathsf{pk}', p - q_s))$ where $q_i$'s are random polynomials of degree $d$ for some security parameter $s$; (iii) it sends all the ciphertext pairs along with ZK proofs of the fact that the homomorphic addition of every pair encrypts the same polynomial (i.e., $p$), to the first party. We need to slightly modify this step to realize our ideal commitment functionality: For the first pair of ciphertexts, the second party will also include a ZK proof of validity of $(\mathsf{E}(\mathsf{pk}', q_1), \mathsf{E}(\mathsf{pk}', p - q_1))$.

The pair of ciphertexts $(\mathsf{E}(\mathsf{pk}', q_1), \mathsf{E}(\mathsf{pk}', p - q_1))$ and the accompanied ZK proof of their validity, constitute the commitment by the second party to its input polynomial $p$. Once again, we note that the simulator in the proof is able to extract $q_1$, $p$, and the randomness used in the two encryptions, due to the randomness recovering property of Paillier's encryption. The proof of security provided in [16] can be easily modified to show the commit-first property of the above-mentioned variant of their OPE construction.

*Claim.* The modified oblivious polynomial evaluation protocol of [16] is a commit-first SFE with security against malicious adversaries.

### 7.1 ZK Proofs for Rate-Limited OPE

We now explain how to derive rate-limited OPE protocols from the scheme of [16], by giving concrete instantiation of our compilers from Section 5 and 6.

Rate-Revealing OPE. Consider first our rate-revealing compiler from Figure 1. A proof of repeated-input, here, is equivalent to proving a statement for the following language:

$$\mathcal{L}^{\mathrm{ope}}(n) = \left\{ \begin{array}{l} (\mathsf{pk}, \hat{c}, c_1, \ldots, c_n) : \ \exists \lambda, r \text{ s.t. } (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{G}(1^\lambda, r) \text{ and} \\ (\mathsf{D}(\mathsf{sk}, \hat{c}) = \mathsf{D}(\mathsf{sk}, c_1) \vee \ldots \vee \mathsf{D}(\mathsf{sk}, \hat{c}) = \mathsf{D}(\mathsf{sk}, c_n)) \end{array} \right\},$$

where the ciphertexts $c_1, \ldots, c_n$ are encryptions of the inputs for $n$ previous executions of the OPE protocol. The ciphertext $\hat{c}$ is the encryption of the input for the current execution.

Such a proof can be obtained by exploiting ZK proofs for the languages $\mathcal{L}^{\mathrm{zero}}$ and $\mathcal{L}^{\mathrm{mult}}$ defined in [16]. Informally, a valid proof of a statement in the language $\mathcal{L}^{\mathrm{zero}}$ says that a ciphertext is an encryption of 0. Language $\mathcal{L}^{\mathrm{mult}}$ allows us to prove that given three ciphertexts, one of them decrypts to the product of the other two underlying plaintexts. Denote the plaintext for each $c_i$ by $m_i$ and the one for $\hat{c}$ by $\hat{m}$. The high level idea is to have the prover compute $\mathsf{E}(\mathsf{pk}, (\hat{m} - m_1) \cdots (\hat{m} - m_n))$, prove correctness of this computation and show that the final ciphertext is an encryption of zero. This clearly ensures correctness when the current input equals one of the inputs used in previous executions. See the full version [11] for a complete description.

Rate-Hiding OPE. In the rate-hiding case, besides a standard proof of the statement "a ciphertext is a valid encryption of bit $b$", the prover also needs

to prove that: (i) "the current commitment corresponds to a fresh input"; (ii) "given a collection of ciphertexts, the sum of the corresponding plaintexts is below some threshold $\pi$".

Note that a proof for the first statement is equivalent to proving that an element is *not* in $\mathcal{L}^{\text{ope}}$ (denoted by $\mathcal{L}^{\overline{\text{ope}}}$). Moreover, we show that a proof for the second statement can also be reduced to a proof of membership in $\mathcal{L}^{\overline{\text{ope}}}$ by relying on the homomorphic properties of the underlying encryption scheme. It remains to show ZK proofs for $\mathcal{L}^{\overline{\text{ope}}}$. It is possible to do so using range proofs, but we show a simple and more efficient construction.

Using techniques of [10], the proofs discussed above can be combined (via conjuctive/disjunctive formulas) to generate a ZK proof of membership for the language used in our rate-hiding compiler. A detailed description of the compiler and the proof of the theorem below is given in this paper's full version [11].

**Acknowledgements.** Daniele Venturi acknowledges support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, and also from the CFEM research center (supported by the Danish Strategic Research Council) within which part of this work was performed. Özgür Dagdelen was supported by CASED (`http://www.cased.de`).

## References

1. M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.R. Sadeghi, and T. Schneider. Secure evaluation of private linear branching programs with medical applications. *Computer Security–ESORICS 2009*, pages 424–439, 2009.
2. D. Beaver and S. Goldwasser. Multiparty computation with faulty majority. In *CRYPTO*, pages 589–590, 1990.
3. A. Beimel, K. Nissim, and E. Omri. Distributed private data analysis: On simultaneously solving how and what. *CoRR*, abs/1103.2626, 2011.
4. M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *CRYPTO*, volume 839 of *LNCS*, pages 216–233, 1994.
5. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. *Computer Security-ESORICS 2008*, pages 192–206, 2008.
6. D. Bogdanov, R. Talviste, and J. Willemson. Deploying secure multiparty computation for financial data analysis. Technical report, Cryptology ePrint, 2011/662.
7. P. Bogetoft, D. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. Nielsen, J. Nielsen, K. Nielsen, J. Pagter, et al. Secure multiparty computation goes live. *FC*, pages 325–343, 2009.
8. D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *ACM symposium on Theory of computing*, pages 11–19. ACM, 1988.
9. D. Chaum, E. van Heijst, and B. Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. In *CRYPTO*, pages 470–484, 1991.
10. R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, pages 174–187, 1994.

11. Ö. Dagdelen, P. Mohassel, and D. Venturi. Rate-Limited Secure Function Evaluation: Definitions and Constructions. Cryptology ePrint, Report 201X/XXX.
12. I. Damgård, M. Geisler, M. Krøigaard, and J. Nielsen. Asynchronous multiparty computation: Theory and implementation. *PKC 2009*, pages 160–179, 2009.
13. M. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. *Theory of Cryptography*, pages 303–324, 2005.
14. R. Gennaro, C. Hazay, and F. Sorensen. Automata evaluation and text search protocols with simulation based security. Cryptology ePrint, Report 2010/484.
15. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.
16. C. Hazay and Y. Lindell. Efficient oblivious polynomial evaluation with simulation-based security. Technical report, Cryptology ePrint, Report 2009/459.
17. C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *Theory of Cryptography*, pages 155–175, 2008.
18. C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. *PKC 2010*, pages 312–331, 2010.
19. C. Hazay and T. Toft. Computationally secure pattern matching in the presence of malicious adversaries. *ASIACRYPT 2010*, pages 195–212, 2010.
20. W. Henecka, S. Kogl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *ACM CCS '07*, 2010.
21. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.
22. Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. *Theory of Cryptography*, pages 575–594, 2007.
23. S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114, 2007.
24. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.
25. P. Mardziel, M. Hicks, J. Katz, and M. Srivatsa. Knowledge-oriented secure multiparty computation. In *ACM SIGPLAN – PLAS*, June 2012.
26. A. McGregor, I. Mironov, T. Pitassi, O. Reingold, K. Talwar, and S. Vadhan. The limits of two-party differential privacy. *ECCC*, 18:106, 2011.
27. W. Ogata and K. Kurosawa. Oblivious keyword search. Technical report, Cryptology ePrint, Report 2002/182.
28. J. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik. Privacy preserving error resilient dna searching through oblivious automata. In *ACM CCS*, pages 519–528, 2007.
29. AC. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.
30. AC. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.