Enforcing ideal-world leakage bounds in real-world secret sharing MPC frameworks

José Bacelar Almeida[1,2], Manuel Barbosa[1,3], Gilles Barthe[4], Hugo Pacheco[1,2], Vitor Pereira[1,3], and Bernardo Portela[1,3]

[1] INESC TEC
[2] Universidade do Minho
[3] FC Universidade do Porto
[4] IMDEA Software Institute

Abstract. We give a language-based security treatment of domain-specific languages and compilers for secure multi-party computation, a cryptographic paradigm that enables collaborative computation over encrypted data. Computations are specified in a core imperative language, as if they were intended to be executed by a trusted-third party, and formally verified against an information-flow policy modelling (an upper bound to) their leakage. This allows non-experts to assess the impact of performance-driven authorized disclosure of intermediate values.

Specifications are then compiled to multi-party protocols. We formalize protocol security using (distributed) probabilistic information-flow and prove security-preserving compilation: protocols only leak what is allowed by the source policy. The proof exploits a natural but previously missing correspondence between simulation-based cryptographic proofs and (composable) probabilistic non-interference.

Finally, we extend our framework to justify leakage cancelling, a domain-specific optimization that allows to, first, write an efficiently computable specification that fails to meet the allowed leakage upper-bound, and then apply a probabilistic pre-processing that brings the overall leakage to within the acceptable range.

1   Introduction

Secure multi-party computation (MPC) is a powerful cryptographic paradigm. MPC protocols allow two or more mutually distrusting parties to collaboratively compute over their private data, revealing nothing more than the result of the computation. MPC eliminates the need for delegating secure computations to a TTP (trusted third party), significantly reducing logistical and trust management problems, as well as security risks inherent to having a TTP as a single point of failure. As a consequence (and after two decades of sustained breakthroughs in its underlying technology) MPC is increasingly used for practical applications [22, 31, 47].

One key element for the practical success of MPC has been the emergence of domain-specific languages and compilers [10, 15, 30, 34, 37, 38, 40, 44, 50]. These MPC software stacks give (non-expert) programmers the ability to develop applications in traditional (sequential) programming languages, as if the computation was to be run by one TTP. These programs are then compiled to (probabilistic) protocols that realize the computation in a distributed, multi-party, setting.

To achieve efficient realizations, MPC programs tend to avoid computations that are expensive in a distributed setting, such as accessing arrays with secret indexes or securely branching based on secret values. A common approach to expose these constraints is via a standard information flow type system, with MPC-specific public control-flow restrictions (control-flow guards and array access expressions for imperative languages [17], or conditionals, fixpoint recursion [40], sum types and higher-order functions [44] for functional languages).[5]

---

[5] It is possible to express oblivious control-flow by computing all possible outcomes and algebraically selecting the correct result, but the overhead can be prohibitive in practical applications. Most often, this process can be performed as a compilation step (cf. [40,44]), but we adopt the approach of [17] where programmers handle oblivious control-flow explicitly, which simplifies our presentation and formalization.

The type system does not constrain the expressivity of the language, thanks to *declassify* statements, which turn an arbitrary expression to public. This suggests that its goal is *not* to enforce a secure information flow policy—a programmer is always free to declassify information—but to make programmers aware that the MPC application will perform some otherwise expensive computations publicly—as a performance optimization technique—and to ensure that data is consistently translated between public and secret semantic domains. Hence, while the type system is useful, it fails to capture rigorously *how much* information is released via declassification.

The lack of a rigorous mechanism for analyzing leakage at source level is a serious hindrance for MPC technology, in particular because obtaining meaningful security guarantees has a significant impact on productivity. Even though high-level domain-specific source languages are tailored for non-experts, it is extremely hard to simultaneously achieve good performance, which implies declassifying intermediate results, and guarantee that leaked information is not harmful within a particular application, which usually calls for a MPC expert.

This paper demonstrates how to leverage language-based techniques to provide users of MPC domain-specific languages early and accurate feedback on the security of their programs. Technically, this is achieved in two steps: source-level analysis and secure compilation.

Source-level analysis We propose an automated method for proving security of source programs. Our notion of security is expressed as a variant of non-interference, and states that inputs related by a leakage specification yield equal leakage, where leakage is modeled using an instrumented source-level semantics. Verification relies on relational program verification techniques, and is performed with minimal overhead. Indeed, we observe that MPC source programs, through information flow types and declassify statements, expose sufficient information to adapt a technique developed for analysing timing leaks in assembly code [1]. Our main contribution at this level is to adapt this technique to the MPC setting, extending it to deal with a real-world MPC programming language, and to demonstrate its application to proving meaningful (not trace-based) leakage upper-bounds. Using our tool for source-level analysis, a programmer that is not a MPC expert can prove a leakage upper-bound that can be matched to the security requirements of the application.

Secure compilation We prove that low-level protocols do not leak more information than source programs from which they are generated. The central challenge here is to connect formally information flow-based notions of security for source programs and cryptographic simulation-based notions of security for protocols. Our solution is based on an alternative notion of protocol security, leveraging probabilistic information flow. We define a distributed probabilistic semantics that gives meaning to securely computing a functionality using a distributed protocol and introduce the notion of each party's view of the protocol. Our notion of protocol security states that parties executing the protocol correctly (a.k.a. honest-but-curious parties) cannot distinguish between two runs of the protocol on related inputs; precisely, the views—distributions over local execution traces—of each party are identical in the two runs.

We show that our notion of security composes to justify the guarantees provided by *secure multiparty compilation*, as used in MPC software stacks: generating MPC protocols for arbitrary source programs by plugging together very simple atomic cryptographic components. Our main theorem states that, for any correctly typed program, source-level security is preserved as distributed information flow security of the compiled protocol. We conclude by proving that, for correct executions of the program, this also implies the intended simulation-based notion of cryptographic security.

The challenge of secure compilation for MPC has been previously addressed by Mitchell et al. [40] and, indeed, our secure compilation theorem has a similar flavour. However, there are two main differences. i. We focus on secret sharing-based MPC, which allows us to give a unified probabilistic information-flow notion of protocol security that applies to both atomic and complex protocols. We prove that this property composes, greatly simplifying our secure compilation proof; we can work purely at the information-flow level, rather than reasoning inductively about the indistinguishability of distributions; and ii. We establish a natural but previously missing connection to standard security notions for information-theoretically secure MPC, by showing that our information-flow notion of protocol security is strong enough to imply the existence of a
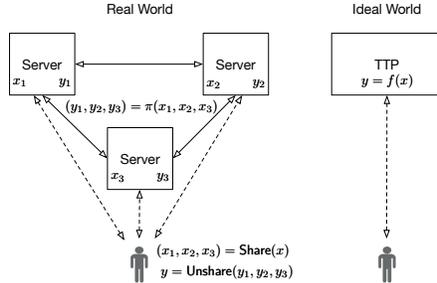
Fig. 1: Real world versus ideal world.

cryptographic simulator that requires only the leakage allowed by the source-level upper-bound to perfectly simulate real-world traces.

As an independent contribution related to performance, we leverage our framework to model *leakage cancelling*, a pattern usually performed by experts for optimizing MPC protocols in two steps: i. implement a specification p that leaks more than what is allowed, and then ii. use an efficient (oblivious) probabilistic preprocessing of inputs that renders leakage useless to an attacker. We give a sufficient condition (C) such that the following composition theorem holds: the sequential composition $p_0; p$ of a Ψ-secure program p_0 satisfying (C) and a Φ-secure program p is itself Φ-secure. Here, Φ-security means that two inputs satisfying a leakage specification Φ lead to identical leakage under program p.

Our main technical contributions are the following:
- an information flow-based definition of source-level leakage, and an automated method for proving that a program satisfies a leakage policy;
- an information flow-based definition of protocol security, and a proof that it entails the expected cryptographic notion of security;
- a proof (using a new technique) that compilation from programs to protocols preserves security;
- a formalization of leakage cancelling that attests its validity as a secure optimization technique;
- an implementation of our techniques for a real-world MPC language and an evaluation on challenging case studies from the literature.

Limitations Our language-based security framework is limited to passive security and private outputs. In the passive security model, honest-but-curious parties execute the protocol correctly, i.e., we do not consider adversaries that deviate from the protocol execution arbitrarily. Removing this assumption without compromising efficiency is an active area of research in cryptography, and extending our results to this setting is an important direction for further work. The standard security model for MPC protocols is universal composability, where attackers get to observe the raw protocol outputs. Our language-based security notion is slightly weaker than this, to practical gain, but it is well known [14] that a standard (and efficient) post-processing permits removing this caveat. We further discuss both limitations below.

2 Overview and motivating examples

The family of MPC protocols that we study in this paper is based on secret sharing, a cryptographic primitive that permits splitting secret data between multiple parties, in such a way that accessing an incomplete subset of these *shares* reveals nothing about the secrets.[6]

---

[6] We will describe these protocols in the deployment setting typically adopted by platforms such as Sharemind, where there are one or more clients providing inputs and receiving outputs from the computation and a set of workers that carry out the computation over secret share data. Still, our results are also applicable to scenarios where input/output parties participate in the computation.

Formally, a n-party secret sharing-scheme over a set S is defined by two algorithms: i. a probabilistic algorithm Share that takes a secret to be shared x and produces a distribution over n-tuples of shares $(x_1, \ldots, x_n) \in S^n$; and ii. a determistic algorithm Unshare that takes a tuple $(x_1, \ldots, x_n)$ and reconstructs the secret x. Correctness of secret sharing states that $\mathsf{Unshare}(\mathsf{Share}(x)) = x$ holds, for every x in S. For concreteness, we will consider an additive secret sharing scheme, where secrets are elements in a finite field $\mathbb{F}$. On input x, Share samples finite field elements $x_1, \ldots, x_n$ uniformly at random, conditioned on $x = x_1 + \ldots + x_n = \mathsf{Unshare}(x_1, \ldots, x_n)$.

The goal of MPC is then to perform (local or distributed) computations over shares, known as protocols, that are homomorphic to computations over the original secrets. This goal is naturally captured using the real-world vs. ideal world paradigm (see Figure 1 for a specialization to the 3-party case). On the right-hand side, one can see the ideal world that is emulated by the MPC platform: a user is able to offload some secret data x to a TTP, and outsource the computation of a function f over this data to obtain a result y. On the left-hand side, the end-user uses a secret sharing-scheme to offload shared data (x_1, x_2, x_3) to three servers in a secure way, and then obtains a secret-shared value (y_1, y_2, y_3) that results from the execution of a distributed protocol π between the three servers. We say that protocol π correctly implements f iff $\mathsf{Unshare}(y_1, y_2, y_3) = f(x)$, for every secret x and provided $(x_1, x_2, x_3) = \mathsf{Share}(x)$.

Designing MPC protocols requires trade-offs between efficiency and security. For instance, the following program, annotated with security types, avoids branching on secret values by using a declassify statement:[7]

```
secret minimum (secret xs) {
    secret min = x[0];
    for (i = 1; i < size(xs); i += 1)
        if declassify(x[i] < min) { min = x[i]; }
    return min; }
```

Even for this simple snippet, the first non-obvious question is how much information is leaked. Leakage at protocol level (messages exchanged among parties) is also rather different from source-level leakage (public values). The second question is whether a protocol π securely implements a program p, i.e. π does not leak more than p. In this paper we address both questions.

We provide a method for programmers to specify allowed leakage using annotations. For the above snippet, the programmer could declare that the function can leak comparisons between the vector elements. In the source annotation language proposed in this paper, this can be expressed as a pre-condition over an initial state xs:

```
forall uint i; 0<=i && i<size(xs)
    && 0<=j && j<size(xs) ==> public(xs[i] < xs[j])
```

Under the hood, this annotation is interpreted as a relational pre-condition on two initial states xs and xs′:

$$\begin{cases} \mathsf{size}(\mathsf{xs}) = \mathsf{size}(\mathsf{xs}') = k \\ \forall\, 0 \leq i, j < k.\mathsf{xs}[i] < \mathsf{xs}[j] \Leftrightarrow \mathsf{xs}'[i] < \mathsf{xs}'[j] \end{cases}$$

Our source-level verification can establish that this leakage bound is valid at source level using a notion of security that defines what source leakage is, and imposes that two executions of the above program on two related inputs states leak the same. A security preservation theorem then guarantees that the source-level bound is preserved by compilation to a distributed protocol.

Next, consider the following implementation of the partition operation for quick sort:

---

[7] Note that declassify statements can be *inferred*, but we follow the common practice of requiring that programmers manually insert them.

$$p ::= \mathsf{skip} \mid p_1; p_2 \mid \mathsf{while}\, e \,\mathsf{do}\, p$$
$$\mid lv := ae \mid \mathsf{if}\, e \,\mathsf{then}\, p_1 \,\mathsf{else}\, p_2$$
$$ae ::= e \mid e_1 \,\mathsf{sop}\, e_2 \mid \mathsf{declassify}\,(e)$$

$$lv ::= x \mid x[e]$$
$$e ::= v \mid x \mid x[e]$$
$$\mid e_1 \,\mathsf{pop}\, e_2$$

Fig. 2: Syntax of source-level language.

```
for (i = 0; i < size (xs); i=i+1) {
    secret y = xs[i];
    if (declassify(y <= p)) {ls = snoc(ls,y);}
    else {rs = snoc(rs,y);}
} return (ls,rs);  }
```

Our verification approach can show that the above leakage bound holds. Moreover, this leakage can be *cancelled* by probabilistic pre-processing. Intuitively, applying a random permutation to the quick sort input makes the sequence of comparisons look random, and useless to an attacker that does not know which permutation was applied. This yields a performance gain as random shuffling can be efficiently computed obliviously [29]. We leverage our formal framework to provide sound conditions for applying this optimization technique.

3   Source-level language

Syntax Our work considers SecreC [15], a commercial MPC language resembling C++, supporting high-level programming features such as procedures, templates or recursion, and used for writing secure applications in the Sharemind framework [17]. For our formal development, we will use a core imperative language extended with a declassify operator.[8] For clarity of presentation and w.l.o.g., we make a syntactic distinction between secure operations sop and public operations pop, and restrict the use of secure operations to top-level expressions. The syntax of programs appears in Figure 2.

Instrumented semantics The semantics of our source language gives meaning to evaluating a MPC specification as if a TTP would be computing directly over the data, with full knowledge of secret and public variables. Despite being agnostic to security, this semantics is instrumented to construct a leakage trace including all branching conditions and all declassified values.

The semantics is defined using the standard notion of transitions between configurations (Figure 3). A configuration $\langle p, m \rangle$ denotes a program p to be executed under a memory m. A memory m maps variables x or array elements $x[i]$ to values v. The evaluation of expression e under memory m is written $[\![e]\!](m)$. A transition from configuration c to c' is denoted by $\langle c \rangle \to_l \langle c' \rangle$. An execution of a program is then a sequence of configurations. Configuration $\langle p, m \rangle$ terminates in m' with leakage l, written $\langle p, m \rangle \Downarrow_l m'$, if $\langle p, m \rangle \to_l^* \langle \mathsf{skip}, m' \rangle$, where $\to^*$ forms the reflexive transitive closure of $\to$ and leakage is concatenated into a leakage trace.

Source-level semantics leaves as undefined the meaning of unsafe programs. A program is *safe* when its semantics is defined for every initial state. In particular, this entails that the program terminates on all inputs. This property can be checked using standard verification techniques supported by our tool mentioned further in the paper.

Source-level security Our notion of source-level security is an information flow policy that sets an upper bound on the leakage of a secure program.

---

[8] This makes our results more widely applicable and helps distinguishing them from language features that are orthogonal to security analysis, but would complicate the formalism without additional insight.

$$\frac{\langle p_1, m \rangle \rightarrow_l \langle \mathsf{skip}, m' \rangle \quad \langle p_2, m' \rangle \rightarrow_{l'} \langle p_2', m'' \rangle}{\langle p_1; p_2, m \rangle \rightarrow_{l \cdot l'} \langle p_2', m'' \rangle} \qquad \frac{\langle p_1, m \rangle \rightarrow_l \langle p_1', m' \rangle \quad p_1' \neq \mathsf{skip}}{\langle p_1; p_2, m \rangle \rightarrow_l \langle p_1'; p_2, m' \rangle}$$

$$\frac{p = \begin{cases} p_1 & \text{if } [\![e]\!](m) = \mathsf{tt} \\ p_2 & \text{if } [\![e]\!](m) = \mathsf{ff} \end{cases}}{\langle \mathsf{if}\ e\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2, m \rangle \rightarrow_{[\![e]\!](m)} \langle p, m \rangle}$$

$$\frac{p' = \begin{cases} p;\ \mathsf{while}\ e\ \mathsf{do}\ p & \text{if } [\![e]\!](m) = \mathsf{tt} \\ \mathsf{skip} & \text{if } [\![e]\!](m) = \mathsf{ff} \end{cases}}{\langle \mathsf{while}\ e\ \mathsf{do}\ p, m \rangle \rightarrow_{[\![e]\!](m)} \langle p', m \rangle}$$

$$\frac{[\![e]\!](m) = v}{\langle lv := e, m \rangle \rightarrow_\varepsilon \langle \mathsf{skip}, m[lv \mapsto v] \rangle}$$

$$\frac{\mathsf{sop}\ ([\![e_1]\!](m), [\![e_2]\!](m)) = v}{\langle lv := e_1\ \mathsf{sop}\ e_2, m \rangle \rightarrow_\varepsilon \langle \mathsf{skip}, m[lv \mapsto v] \rangle}$$

$$\frac{[\![e]\!](m) = v}{\langle lv := \mathsf{declassify}\ (e), m \rangle \rightarrow_v \langle \mathsf{skip}, m[lv \mapsto v] \rangle}$$

Fig. 3: Source-level instrumented semantics.

Definition 1 (Source-level security). *Let Φ be relations over memories. A program p is Φ-secure whenever:*

$$\left(\begin{matrix} \langle p, x_1 \rangle \Downarrow_{l_1} y_1 \\ \langle p, x_2 \rangle \Downarrow_{l_2} y_2 \end{matrix} \right) \Rightarrow \Phi(x_1, x_2) \Rightarrow l_1 = l_2$$

Note that every program is secure w.r.t. full leakage, i.e., the equality relation, since the instrumented semantics is deterministic. Further, we can assume w.l.o.g. that Φ is an equivalence relation, as Φ-security and Φ^*-security coincide—as usual, Φ^* denotes the reflexive, symmetric and transitive closure of Φ. Finally, note that every function ℓ mapping states to an arbitrary type L induces a leakage relation $\Phi_\ell(x, y)$ defined as $\ell(x) = \ell(y)$.

The definition is an instance of observational non-interference, where it is required that observation traces l_1 and l_2 (rather than program outputs) are equivalent under related program inputs. An implication of source-level security is that, when dealing with Φ-equivalent inputs, two executions of a source-secure program are guaranteed to run in lock-step, i.e., all Φ-equivalent inputs have identical control flow. This is because all branching conditions are included in the leakage traces.

4 Source-level security verification

Type system As we have seen in Section 2, MPC languages have an intrinsic notion of security domains, forming a security lattice $\mathcal{L}$ satisfying the ordering $\mathsf{public} \sqsubseteq \mathsf{private}$. We formalize a type system for our source-level language (Figure 4) that statically assigns security labels to intermediate variables. A security environment $\Gamma : V \rightarrow \mathcal{L}$ defines a mapping from variables to security labels, and $\Gamma(x)$ denotes the security label of variable x in environment Γ. We define typing judgments for programs p ($\Gamma \vdash p$), and auxiliarly for left-values lv ($\Gamma \vdash_{lv} lv : s$), expressions e ($\Gamma \vdash_e e : s$) and assignment expressions ae ($\Gamma \vdash_{ae} ae : s$) under security label s. Note that the type rules impose that branching conditions and array indices are public. As noted in the introduction, this is a design choice that we inherit from Sharemind [17].

Our type system is reminiscent of security type systems for information-flow with declassification [40, 49], which typically enforce trace-based notions of flow-insensitive non-interference or delimited release. Still, in this paper it only serves to syntactically restrict programs to a form—secret variables are only directly assigned to secret variables, or used as input to sop or declassify operations—compatible with our distributed

$$\frac{}{\Gamma \vdash \mathsf{skip}} \qquad \frac{\Gamma \vdash p_1 \quad \Gamma \vdash p_2}{\Gamma \vdash p_1 ; p_2} \qquad \frac{\Gamma \vdash_e e : \mathsf{public} \quad \Gamma \vdash p}{\Gamma \vdash \mathsf{while}\, e\, \mathsf{do}\, p}$$

$$\frac{\Gamma \vdash_{lv} lv : s \quad \Gamma \vdash_{ae} ae : s' \quad s' \sqsubseteq s}{\Gamma \vdash lv := ae} \qquad \frac{\Gamma \vdash_e e : \mathsf{public} \quad \Gamma \vdash p_1 \quad \Gamma \vdash p_2}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, p_1\, \mathsf{else}\, p_2}$$

$$\frac{\Gamma(x) = s}{\Gamma \vdash_{lv} x : s} \qquad \frac{\Gamma(x) = s \quad \Gamma \vdash_e e : \mathsf{public}}{\Gamma \vdash_{lv} x[e] : s}$$

$$\frac{}{\Gamma \vdash_e v : \mathsf{public}} \qquad \frac{\Gamma \vdash_{lv} lv : s}{\Gamma \vdash_e lv : s} \qquad \frac{\Gamma \vdash_e e_1 : \mathsf{public} \quad \Gamma \vdash_e e_2 : \mathsf{public}}{\Gamma \vdash_e e_1\, \mathsf{pop}\, e_2 : \mathsf{public}}$$

$$\frac{\Gamma \vdash_e e : s \quad s \sqsubseteq s'}{\Gamma \vdash_e e : s'} \qquad \frac{\Gamma \vdash_e e : \mathsf{private}}{\Gamma \vdash_{ae} \mathsf{declassify}\,(e) : \mathsf{public}}$$

$$\frac{\Gamma \vdash_e e : s}{\Gamma \vdash_{ae} e : s} \qquad \frac{\Gamma \vdash_e e_1 : \mathsf{private} \quad \Gamma \vdash_e e_2 : \mathsf{private}}{\Gamma \vdash_{ae} e_1\, \mathsf{sop}\, e_2 : \mathsf{private}}$$

Fig. 4: Type system of our source language.

semantic evaluation rules, where public values can be transparently used as secret ones, but the contrary is not true.[9]

Leakage analysis A security type system can serve as a preliminary security analysis. Indeed, if declassify statements are *not* used (the program has *no* leakage), it can enforce source-level security for all leakage relations that guarantee equality of public inputs, but says little about security in the presence of leakage. Thus, we introduce a Security Hoare Logic (SHL) that provides a more powerful way to reason compositionally about source-level security w.r.t. a general leakage relation. We start by considering a security post-condition.

Definition 2 (Composable source-level security). *A program p is (Φ, Ψ)-secure, written $\{\Phi\}\, p\, \{\Psi\}$, whenever:*

$$\begin{pmatrix} \langle p, x_1 \rangle \Downarrow_{l_1} y_1 \\ \langle p, x_2 \rangle \Downarrow_{l_2} y_2 \end{pmatrix} \Rightarrow \Phi(x_1, x_2) \Rightarrow \Psi(y_1, y_2) \wedge l_1 = l_2$$

The triple $\{\Phi\}\, p\, \{\Psi\}$ forms a security contract: the pre-condition Φ is an upper bound on the leakage of program p, and the post-condition Ψ evinces known leakage after running p. The rules of SHL are given in Figure 5, and are similar to those of standard Hoare Logic, with additional implications for leakage traces. Intuitively, it models the advice that a security verification system can give to a programmer who needs to justify leakage.

Formally, SHL has a relational interpretation consistent with source-level security, and reasons about pairs of executions of the same program running in lockstep.

Theorem 1. $\models \{\Phi\}\, p\, \{\Psi\}$ *is derivable iff* $\{\Phi\}\, p\, \{\Psi\}$.

A consequence of having public control-flow is that source-level security can be verified using deductive verification techniques over a self-composed program, with complexity in the same class as the original program.[10]

---

[9] Note that source-level security may allow the contrary if the secret values are publicly known from the leakage relation. This, in particular, is consistent to running a sop with public arguments; in practice, languages can offer a specialized sop relying on public information, e.g., instead of lifting a constant public value 3 and securely multiplying it by a secret variable y $((*)\, 3\, y)$, securely performing $(3*)\, y$.

[10] The verification techniques also support programs with secret control flow, but become less tractable. We could also apply our leakage verification after a secret-to-public control-flow compilation step.

$$\dfrac{}{\vDash \{\Phi\}\ \mathsf{skip}\ \{\Phi\}} \qquad \dfrac{\vDash \{\Phi\}\, p_1\, \{\Theta\} \quad \vDash \{\Theta\}\, p_2\, \{\Psi\}}{\vDash \{\Phi\}\, p_1; p_2\, \{\Psi\}}$$

$$\dfrac{\begin{array}{c}\vDash \{\Phi \wedge e\}\, p_1\, \{\Psi\} \\ \vDash \{\Phi \wedge \neg e\}\, p_2\, \{\Psi\} \\ \Phi \Rightarrow \mathsf{public}\,(e)\end{array}}{\vDash \{\Phi\}\ \mathsf{if}\ e\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2\, \{\Psi\}} \qquad \dfrac{\begin{array}{c}\Phi \Rightarrow \Phi' \\ \vDash \{\Phi'\}\, p\, \{\Psi'\} \\ \Psi' \Rightarrow \Psi\end{array}}{\vDash \{\Phi\}\, p\, \{\Psi\}}$$

$$\dfrac{\begin{array}{c}\vDash \{\Phi \wedge e\}\ \{\Phi\} \\ \Phi \Rightarrow \mathsf{public}\,(e)\end{array}}{\vDash \{\Phi\}\ \mathsf{while}\ e\ \mathsf{do}\ p\, \{\Phi \wedge \neg e\}} \qquad \dfrac{\Phi = \Psi[e/lv]}{\vDash \{\Phi\}\, lv := e\, \{\Psi\}}$$

$$\dfrac{\Phi = \Psi[e_1\, \mathsf{sop}\, e_2/lv]}{\vDash \{\Phi\}\, lv := e_1\, \mathsf{sop}\, e_2\, \{\Psi\}} \qquad \dfrac{\Phi = \Psi[e/lv] \Rightarrow \mathsf{public}\,(e)}{\vDash \{\Phi\}\, lv := \mathsf{declassify}\,(e)\, \{\Psi\}}$$

$$\mathsf{public}\,(e)(x_1, x_2) \triangleq [\![e]\!](x_1) = [\![e]\!](x_2)$$

Fig. 5: Inference system for Security Hoare Logic.

5 Low-level language

In this section we give a meaning to securely computing a source-level program using a distributed, secret sharing-based, cryptographic protocol. We will do this by providing a (low-level) distributed semantics for the specification language we introduced in the previous section. We start by introducing some notation.

Notation Our low-level distributed semantics keeps a state of n separate maps $\mathsf{M} = (\mathsf{M}_1, \ldots, \mathsf{M}_n)$, each corresponding to the local state of a different party. Together, they satisfy the (informal) invariant that $(\mathsf{M}_1, \ldots, \mathsf{M}_n)$ encodes the state of the source-level computation. Each $\mathsf{M}_i : V \mapsto \{0, 1\} * S$ maps variables to pairs, where each stored value holds an encoding bit and a share. The encoding bit is used to identify values stored in shared form from a special encoding of public values. A variable v holding a shared value $\bar{x}$ will be stored in the n maps as $\mathsf{M}_i[v] = (0, x_i)$, for $1 \leq i \leq n$. To simplify the encoding of public values and w.l.o.g., we assume that n is odd. A variable v holding a public value c will be stored as $\mathsf{M}_i[v] = (1, c)$, for odd $1 \leq i \leq n$ and $\mathsf{M}_2[v] = (1, -c)$ for the remaining parties.

This representation allows to locally reconstruct c without communication, and is consistent with its shared representation, as $c = (n/2 + 1)c - (n/2)c$. The decision to share public values was taken so that there would be a greater integration with MPC. The public values can thus always be used as shared (secret) values, but the converse is not true. We will use the notation $\|(b, a)\|_i$ to represent the decoding of a stored value:

$$\|(b, a)\|_i := \mathsf{if}\ b = 1 \wedge i \bmod 2 = 0\ \mathsf{then}\ (-a)\ \mathsf{else}\ a\,.$$

The low-level distributed semantics assumes the existence of basic cryptographic protocols for all secure operators sop in the source language. Each of these is a n-party protocol π, whose (distributed) execution is denoted $(\bar{y}, t, c) \twoheadleftarrow \pi_{\mathsf{sop}}(\bar{x}, \bar{x}')$, as short-hand for: i. sampling random coins $c = (c_1, \ldots, c_n)$ in the appropriate spaces, ii. running the n parties on inputs $((x_1, x'1), \ldots, (x_n, x'n)) = (\bar{x}, \bar{x}')$ and random coins, iii. recording the interaction between parties in trace t, and iv. collecting outputs $(y_1, \ldots, y_n) = \bar{y}$. We will use t_i, for $1 \leq i \leq n$ to denote the part of the communications trace that is within the view of each party (both sent and received messages). We note that the local behaviour of each participant i is fully determined by its local input shares x_i, x'_i its random coins c_i and its local trace t_i. We can therefore meaningfully refer to the recomputing of a local output as $y_i \leftarrow \pi_i(x_i, x'_i, t_i, c_i)$.

For illustrating the two semantic domains, we explicitly define a $\pi_{\mathsf{declassify}}$ protocol (Figure 6), that moves secret shared distributed values to public local values.

Distributed semantics The distributed semantics for the SecreC language is presented in Figure 7. It relies on

8

$\pi_{\text{declassify}}(u_1, \ldots, u_n; c_1, \ldots, c_n)$:

For $i = 1$ to $n - 1$: P_i sends c_i to P_{i+1}; P_n sends c_n to P_1
P_1 computes $u'_1 \leftarrow u_1 + c_n - c_1$, broadcasts u'_1
For $i = 2$ to n: P_i computes $u'_i \leftarrow u_i + c_{i-1} - c_i$, broadcasts u'_i
For $i = 1$ to n, s.t. i is odd:
  P_i computes $u = \mathsf{Unshare}(\bar{u}')$, locally returns u
For $i = 1$ to n, s.t. i is even:
  P_i computes $u = \mathsf{Unshare}(\bar{u}')$, locally returns $(-u)$

Fig. 6: Declassify protocol.

$$\frac{}{\langle\, \mathsf{skip}\,;p, \mathsf{M}_i\rangle \Rightarrow \langle p, \mathsf{M}_i\rangle} \qquad \frac{\langle p_1, \mathsf{M}_i\rangle \Rightarrow \langle p'_1, \mathsf{M}_i{'}\rangle}{\langle p_1;p_2, \mathsf{M}_i\rangle \Rightarrow \langle p'_1;p_2, \mathsf{M}_i{'}\rangle} \qquad \frac{[\![e]\!](\mathsf{M}_i) = (b,v)}{\langle lv := e, \mathsf{M}_i\rangle \Rightarrow \langle\, \mathsf{skip}\,, \mathsf{M}_i[lv \mapsto (b,v)]\rangle}$$

$$\frac{p_j = \begin{cases} p_1 & \text{if } \|[\![e]\!](\mathsf{M}_i)\|_i = \mathsf{tt} \\ p_2 & \text{if } \|[\![e]\!](\mathsf{M}_i)\|_i = \mathsf{ff} \end{cases}}{\langle\, \mathsf{if}\, e \,\mathsf{then}\, p_1 \,\mathsf{else}\, p_2, \mathsf{M}_i\rangle \Rightarrow \langle p_j, \mathsf{M}_i\rangle} \qquad \frac{p' = \begin{cases} p; \mathsf{while}\, e \,\mathsf{do}\, p & \|[\![e]\!](\mathsf{M}_i)\|_i = \mathsf{tt} \\ \mathsf{skip} & \|[\![e]\!](\mathsf{M}_i)\|_i = \mathsf{ff} \end{cases}}{\langle\, \mathsf{while}\, e \,\mathsf{do}\, p, \mathsf{M}_i\rangle \Rightarrow \langle p', \mathsf{M}_i\rangle}$$

$$\frac{[\![e_1 \,\mathsf{sop}\, e_2]\!]_{ae}(\mathsf{M}) = (\bar{v}', t, c)}{\langle lv := e_1 \,\mathsf{sop}\, e_2, \mathsf{M}\rangle \Rightarrow_{t,c} \langle\, \mathsf{skip}\,, \forall i : \mathsf{M}_i[lv \mapsto (0, v'_i)]\rangle} \qquad \frac{[\![\, \mathsf{declassify}\,(e)]\!]_{ae}(\mathsf{M}) = (\bar{v}', t, c) \quad b_{1,3,5\ldots} = 0 \quad b_{2,4,6\ldots} = 1}{\langle lv := \mathsf{declassify}\,(e), \mathsf{M}\rangle \Rightarrow_{t,c} \langle\, \mathsf{skip}\,, \forall i : \mathsf{M}_i[lv \mapsto (b_i, v'_i)]\rangle}$$

Fig. 7: Low-level distributed semantics.

the following expression evaluation rules.

$$[\![v]\!](\mathsf{M}_i) = \mathsf{M}_i[v]$$
$$[\![x]\!](\mathsf{M}_i) = \mathsf{M}_i(x)$$
$$[\![x[e]]\!](\mathsf{M}_i) = \mathsf{M}_i(x)[\|[\![e]\!](\mathsf{M}_i)\|_i]$$
$$[\![e_1 \,\mathsf{pop}\, e_2]\!](\mathsf{M}_i) = (1, v)$$

$$\text{where} \quad v_1 = \|[\![e_1]\!](\mathsf{M}_i)\|_i$$
$$v_2 = \|[\![e_2]\!](\mathsf{M}_i)\|_i$$
$$\|(1, v)\|_i = \mathsf{pop}\,(v_1, v_2)$$

We present the semantics from a local evaluation perspective, as transitions do not require interaction between the parties, except for the evaluation rules of sop or $\mathsf{declassify}$ operators that have explicit communication:

$$[\![e_1 \,\mathsf{sop}\, e_2]\!]_{ae}((\mathsf{M}_1, \ldots, \mathsf{M}_n)) = (\bar{v}', t, c)$$
$$\text{where} \quad \forall 1 \le i \le n,\ [\![e_1]\!](\mathsf{M}_i) = (\cdot, v_{1,i})$$
$$\forall 1 \le i \le n,\ [\![e_2]\!](\mathsf{M}_i) = (\cdot, v_{2,i})$$
$$(\bar{v}', t, c) \twoheadleftarrow \pi_{\mathsf{sop}}\,(\bar{v}_1, \bar{v}_2)$$
$$[\![\, \mathsf{declassify}\,(e)]\!]_{ae}((\mathsf{M}_1, \ldots, \mathsf{M}_n)) = (\bar{v}', t, c)$$
$$\text{where} \quad \forall i,\ [\![e]\!](\mathsf{M}_i) = (\cdot, v_i)$$
$$(\bar{v}', t, c) \twoheadleftarrow \pi_{\mathsf{declassify}}\,(\bar{v})$$

The pre-requisites in these two transition rules will block the progress of all local evaluations until all parties synchronously execute them. Furthermore, only these rules contribute to the global execution trace.

As before, $\Rightarrow^*$ forms the reflexive transitive closure of $\Rightarrow$. The execution trace for the distributed evaluation of the program is the concatenation of rule traces. An execution of a program is then a sequence of distributed configurations. Configuration $\langle p, (\mathsf{M}_1, \ldots, \mathsf{M}_n)\rangle$ terminates execution in configuration $(\mathsf{M}'_1, \ldots, \mathsf{M}'_n)$ with trace

(t, c), written $\langle p, (\mathsf{M}_1, \ldots, \mathsf{M}_n) \rangle \Downarrow_{(t,c)} (\mathsf{M}_1', \ldots, \mathsf{M}_n')$, if

$$\forall 1 \leq i \leq n : \langle p, (\mathsf{M}_i) \rangle \Rightarrow^*_{(t,c)} \langle (\,\mathsf{skip}\,), (\mathsf{M}_i') \rangle \,.$$

We slightly abuse notation and refer to the distributed evaluation of secure operators sop and $\mathsf{declassify}$ as

$$\langle\,\mathsf{sop}\,, \bar{x} \rangle \Downarrow_{(t,c)} \bar{x}' \qquad \langle\,\mathsf{declassify}\,, \bar{x} \rangle \Downarrow_{(t,c)} \bar{x}' \,.$$

We have intentionally written our distributed semantics so that it can be seen as a high-level cryptographic protocol that relies on lower level ones to evaluate a program p. Put differently, our distributed semantics describes a compiler that takes a high level MPC specification and produces a composite protocol π_p. We will prove strong cryptographic security properties for this protocol in the style of certified compilation: if p is source-level secure, then π_p will guarantee that this security is translated into standard cryptographic security guarantees.

Low-level correctness and security To reason about the guarantees provided by our MPC software stack, we introduce correctness and security notions for low-level evaluations. The definitions apply to high-level cryptographic protocols π_p and also to low-level cryptographic protocols π_{sop} and $\pi_{\mathsf{declassify}}$.

Intuitively, correctness states that whatever behaviours are observable at the source level will be also observable at target level, modulo the sharing relation.

Definition 3 (Low-level correctness). *A protocol π is low correct for specification $s \in \{p, \mathsf{sop}\,, \mathsf{declassify}\,\}$ if, for all sharings $\bar{x}$, we have*

$$\langle s, \mathrm{Unshare}(\bar{x}) \rangle \Downarrow \mathrm{Unshare}(\bar{y}) \Rightarrow \langle \pi, \bar{x} \rangle \Downarrow_{t,c} \bar{y}$$

We define security by means of a probabilistic non-interference notion.

Definition 4 (Low-level security). *Let Φ be a relation over unshared inputs. A protocol π is secure for Φ if, for all $1 \leq i \leq n$ and all sharings $\bar{x}, \bar{x}'$ such that $x = \mathrm{Unshare}(\bar{x})$ and $x' = \mathrm{Unshare}(\bar{x}')$ we have*

$$\Phi(x, x') \wedge \left(\begin{array}{c} \langle \pi, \bar{x} \rangle \Downarrow_{t,c} \bar{y}_{\_} \\ \langle \pi, \bar{x}' \rangle \Downarrow_{t',c'} \bar{y}' \end{array} \right) \Rightarrow (t_i, c_i) = (t_i', c_i')$$

Informally, this definition states that messages exchanged (traces t) and randomness (coins c) as seen by each party throughout two distinct executions of the same protocol will have identical distributions and thus leak no information about the (unshared) inputs in addition to that revealed by the leakage relation; two distributions are equal iff they assign the same probability to every element in their support.

Note that leakage relations are expressed over (unshared) values, and therefore low-secure protocols guarantee that no information about specific shares is revealed. This is extremely important when relating our security notion to cryptographic security definitions, where the attacker is able to see part of the shares. We illustrate these definitions with the $\pi_{\mathsf{declassify}}$ protocol.

Lemma 1. *The $\pi_{\mathsf{declassify}}$ is correct and is low-secure for the trivial leakage relation that reveals the full (unshared) input, i.e., $\Phi(x, x') := x = x'$, but hides all additional information about the input shares.*

Proof. For correctness, observe that the declassify operator preserves the input value: it just changes its security domain. The protocol guarantees this, as the final shares returned to all the parties are simply the public storage of a *resharing* of the input value. In the resharing step, all parties add and remove some randomness from their input shares, in such a way that the reconstructed reshared value will coincide with the original one by cancelling these operations.

For security, observe that the leakage relation imposes that the broadcast operations will be identically distributed for both inputs, as they are the same reconstructed value. Furthermore, the coins and messages exchanged in the resharing step are uniformly distributed and independent of everything else in each party's view, as this view does not have enough information to bind these values to the subsequent broadcast information (at least one share will be missing).

Compositional reasoning Compositionality is an important property of the correctness and security definitions for low-level distributed executions.

Lemma 2 (Low-level composability). *Let protocols π_1, π_2 be correct w.r.t. programs p_1, p_2 and secure w.r.t. leakage relations Φ_1, Φ_2. Then, $\pi(\bar{x}) = \pi_2(\pi_1(\bar{x}))$ is correct for $p_2 \circ p_1$ and secure for $\Phi = \Phi_1 \wedge \Phi_2 \circ p_1$. Similarly, $\pi(\bar{x}) = (\pi_1(\bar{x}), \pi_2(\bar{x}))$ is correct for $p_2 \times p_1$ and secure for $\Phi_1 \wedge \Phi_2$.*

Proof. Correctness of both sequential and parallel composition follows immediately from the fact that both π_1 and π_2 are correct by hypothesis and the observation that, by construction, the composed protocols match the structure of the composed source level programs.

For security, observe that the leakage relations in both cases will only admit pairs of inputs to the composed protocol for which *both* protocols are guaranteed not to leak any information. In sequential composition one excludes inputs that could leak under π_1 and also those that would lead to intermediate $p_1(x)$ and $p_1(x')$ that might leak under π_2. (Here the fact that π_1 is correct is used to assert that π_2 is fed with values that satisfy this condition.) For parallel composition one simply excludes inputs that might leak under either protocol.

6   Security-preserving compilation

Our main theorem shows that our MPC software stack preserves source-level information-flow security to probabilistic distributed non-interference at the low-level. As a result, we will show in the next section that such systems guarantee security in the cryptographic sense.

Theorem 2 (Main Theorem). *Assume by hypothesis that all π_{sop} are correct and secure for the leakage relation that accepts all inputs (nothing leaks). Let p be a program, such that $\Gamma \vdash p$ for some security context Γ, and let Φ be a leakage relation. Then, if p is source-level secure for Φ, we have that π_p is correct for p and secure for Φ.*

Proof. The proof proceeds in two steps that we formalize using two lemmas presented below. We first prove in Lemma 3 correctness of the low-level execution by relying on the composability of low-level correctness and the fact that the program type-checks (this guarantees that no secret-encoded value is used in a local public computation). We then prove in Lemma 4 that, for all MPC specifications p, the composability of low-security implies that the distributed execution will leak no more than the source-level traces. In other words, we get low-level security for the low-level leakage function that imposes equality of source-level traces (the leakage function is based on the source-level instrumented semantics). By transitivity, we can therefore conclude that π_p is Φ-low-secure, as source-level security guarantees source-level trace equality over inputs satisfying Φ.

We now provide the proofs for Lemmas 3 and 4.

Lemma 3. *Assume by hypothesis that all π_{sop} are sop-low-correct. Let p be a program such that $\Gamma \vdash p$ for some security context Γ. Then π_p is correct for p.*

Proof. We note that the definition of correctness says nothing about unsafe programs at source level, so we consider only safe programs in the proof. For such programs there is a one-to-one matching between source-level and distributed semantics rules. For public values, the two semantics match modulo the correctness of the encodings of inputs to public operators. This is guaranteed by the fact that the source program typechecks and that the declassify protocol correctly encodes its public outputs. Furthermore, this guarantees that the control flow of the protocol exactly matches the control flow of the source program and that distributed evaluations of sop operations are executed synchronously by all parties. For secret operations, the two semantics match modulo the correctness of secret sharing encodings and the fact that protocol correctness is composable.

Lemma 4. *Assume by hypothesis that all π_{sop} are low-secure for the empty leakage relation. Let p be a program such that $\Gamma \vdash p$ for some security context Γ. Let also Φ_p be the leakage relation that imposes source-level leakage equality. Then, we have that π_p is Φ_p-secure.*

Proof. We first observe that unsafe programs are compiled to unsafe protocols, for which both source-level and low-level security definitions say nothing. (This is because we assume sop operators are total at source level and all π_{sop} protocols are correct, and hence all sources of unsafety match in the source-level and low-level evaluations.) We therefore only need to address safe programs in the proof, which are compiled to safe protocols. Our correctness proof above states that, when evaluating such programs, there is a one-to-one matching between source-level and low-level transitions in the semantics (including synchronous execution of low-level distributed protocols). Looking at the distributed semantics, we see that execution traces correspond to the execution of declassify and sop protocols, where the latter guarantee identical traces for *all* inputs. Composability of low-level security therefore implies that we will have identical low-level traces, provided that the Φ_p leakage relation implies that all pairs of inputs considered in low-level security will lead to identical sequences of declassified (unshared) values, as this is the leakage required by declassify operations. This is indeed the case, as Φ_p guarantees lock-step execution of evaluation for any such inputs, and low-level correctness guarantees equality of declassified values.

7   Cryptographic security

Cryptographic security of MPC protocols is typically defined using the simulation paradigm. Intuitively, the definition states that no attacker can distinguish a real world from an ideal world. In the real world, an attacker $\mathcal{A}$ interacts with the cryptographic protocol π directly, according to a set of rules that define an attack model In the ideal world, the parties relying on the protocol have access to an ideal functionality $\mathcal{F}$ representing a TTP.

The ideal functionality also defines what information from the participant's inputs can be leaked to the attacker. Then, a simulator $\mathcal{S}$ must be able to fool the attacker into thinking it is actually in the real world based on this leakage, even when the attacker sees the outputs produced by the ideal functionality on inputs of its choosing. The existence of $\mathcal{S}$ shows that whatever the adversary sees in its attack can contain no more information about the inputs than what is specified by the functionality in the ideal world. Furthermore, since the attacker can observe the protocol outputs, and compare them to the ideal functionality output, it also implies that the protocol must be correct. The simulation-based definition that we adopt, and discuss later in this section, is called *privacy* [14]. Our notion of simulation-based security implies standard (computational) MPC security, since our simulators are polynomial time. This means that the resulting protocols can be directly used by cryptographers in combination with other constructions satisfying computational security in a compositional setting such as the Universal Composability framework.

We will consider a simple class of ideal functionalities. We denote such functionalities as $\mathcal{F}\langle p, \ell \rangle$, to indicate that they are parameterized by a source program p and a leakage function ℓ. The functionality specifies what a cryptographic protocol should achieve when executed on some shared initial state $\bar{x} = (\mathsf{M}_1, \ldots, \mathsf{M}_n)$:

  i. produce a result y such that $y = p(\mathrm{Unshare}(\bar{x}))$, where the meaning of this evaluation is given by the source-level semantics (in cryptography all specifications are total, and so p must be safe);
 ii. Leak at most $\ell(\mathrm{Unshare}(\bar{x}))$ information about the unshared input.

The security definition is given in Figure 8.

The attacker gets to pick the shared input and the identity of one party that will be corrupted.[11] This means that the attacker will know whatever this party knows, and still the protocol must leak only what the ideal functionality specifies. In practice this ensures that, if there is an honest majority that does not collude to break the protocol, security is guaranteed.

In the real world the attacker observes the unshared protocol output and the part of the trace corresponding to the corrupted party. In the ideal world, the attacker sees the ideal functionality output and a simulated trace. We note that the ideal world simulator gets to see the input share of the corrupt party and the allowed leakage, and it must simulate the rest of the corrupt party's view.

---

[11] The corruption of a single party is made for clarity of presentation. The same model and results can trivially be generalised to allow for an arbitrary set of corrupt parties under some adversarial threshold.

game **Real**$_{\Pi,\mathcal{A}}(\,)$:	game **Ideal**$_{\mathcal{F}\langle p,\ell\rangle,\mathcal{S},\mathcal{A}}(\,)$:
$(\bar{x},i,\mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$	$(\bar{x},i,\mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$
	$x \leftarrow \mathrm{Unshare}(\bar{x})$
	$y \leftarrow p(x)$
$(\bar{y},t,c) \twoheadleftarrow \Pi(\bar{x})$	$l \leftarrow \ell(x)$
$y \leftarrow \mathrm{Unshare}(\bar{y})$	$(y_i,t,c) \twoheadleftarrow \mathcal{S}(i,l,x_i)$
Return $\mathcal{A}_2(\bar{x},y,y_i,t_i,c_i,\mathsf{st})$	Return $\mathcal{A}_2(\bar{x},y,y_i,t_i,c_i,\mathsf{st})$

Fig. 8: Cryptographic privacy.

Definition 5 (Cryptographic privacy). *We say a protocol π is (ℓ,p)-private if there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that, for all adversaries $\mathcal{A}$, the following definition of advantage is 0:*

$$\mathbf{Adv}_{\Pi,\mathcal{F},\mathcal{A},\mathcal{S}} = \Pr[\mathsf{Real}_{\Pi,\mathcal{A}}] - \Pr[\mathsf{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{A}}],$$

where games Real *and* Ideal *are described in Figure 8.*

We do not impose a bound on the attacker's computational power, meaning that we obtain information-theoretic security. The following theorem states that low-level security and correctness as defined in Section 5 imply cryptographic privacy, which allows us to rely only on language-based security techniques to reason about the security of our MPC software stack. Our result requires leakage functions to be efficiently invertible, in the sense that there exists a polynomial-time algorithm that, given l, computes some input x' such that $\ell(x') = l$. If this is not the case, then we get a weaker notion of security where the simulator must also be unbounded (the proof is the same except for this detail).

Theorem 3 (Cryptographic privacy from probabilistic non-interference). *Let π be a protocol, p a safe program in our source language and ℓ a leakage function. Assume also that ℓ is efficiently invertible.[12] Then, if π is low-correct for p and it is low-secure for leakage relation Φ_ℓ, we have that π is (ℓ,p)-private.*

A full game-based version of the proof can be found in Appendix D. Combined with the main theorem in the previous section, this result yields the following corollary.

Corollary 1 (Privacy-preserving compilation). *Assume that all π_{sop} are correct and secure for the empty leakage relation. Let p be a safe program such that $\Gamma \vdash p$ for some security context Γ and let ℓ be a leakage function. Then, if p is Φ_ℓ-secure, we have that π_p is (p,ℓ)-private.*

Relation to Universal Composability The standard model for describing security of secure computation protocols is the Universal Composability (UC) framework [18]. The notion of privacy that we consider in this paper is weaker than UC-security, because the attacker does not see the raw output of the protocol in its shared form. However, this is not a significant limitation. Indeed, our results readily extend to a UC-realization of an arithmetic black box (ABB) [24] using standard techniques. ABBs are a common abstraction of secure computation applications when the goal is to design a system that performs several basic operations before producing an output [19, 20, 33]. Appendix B discusses this in further detail. Furthermore, the presented model considers static corruptions, while results in [14] consider adaptive corruptions. In Appendix C we discuss how our results can easily be extended to adaptive corruptions under the same assumptions.

8 Leakage cancelling

The resolution of the security versus performance contention in MPC has led to interesting optimization techniques. One such technique consists of composing a program p that is secure for a leakage relation Φ with

---

[12] Requiring leakage functions to be efficiently invertible is not a significant caveat. This is true for the empty leakage function, and for all practical examples we have encountered, except if p is computing a cryptographic function for which efficient inverters are not known.

a probabilistic pre-processing step p_0, resulting in a (probabilistic, yet) functionally equivalent program p' that satisfies a weaker leakage relation Ψ. The requirement that Φ implies Ψ reflects the natural information-theoretic interpretation of relations, and ensures that the program p' leaks less information than p. A typical example is sorting: p leaks the length of the array and the sequence of the comparison, as modeled by the relation Φ_{cmp} defined in Section 2, and we want p' to only leak the length of the array. Leakage cancelling can be achieved in this case by obliviously randomly shuffling the input array. We provide a rigorous justification of this technique.

We model probabilistic behaviors by extending the expression language with probabilistic operators. The instrumented semantics of programs is modified accordingly: $\langle p, x \rangle \Downarrow_{\tilde{l}} \tilde{y}$ now states that executing program p with initial memory x terminates with distribution $\tilde{y}$ on output memories and a distribution on leakage traces.[13] When programs have deterministic leakage, i.e., all their guards and declassified expressions do not depend on values computed by probabilistic operators, we write $\langle p, x \rangle \Downarrow_l \tilde{y}$ to state that executing program p with initial memory x terminates with distribution $\tilde{y}$ on output memories and leakage trace l.

The following composition theorem provides sufficient conditions for leakage cancelling. The theorem is stated in terms of a lifting of source-level Φ-security for arbitrary probabilistic programs.

Definition 6 (Probabilistic source-level security). *A probabilistic program p is Φ-secure whenever:*

$$\Phi(x_1, x_2) \,\wedge\, \left(\begin{array}{l} \langle p, x_1 \rangle \Downarrow_{\tilde{l_1}} \tilde{y_1} \\ \langle p, x_2 \rangle \Downarrow_{\tilde{l_2}} \tilde{y_2} \end{array} \right) \,\Rightarrow\, \tilde{l_1} = \tilde{l_2}$$

Theorem 4 (Secure pre-processing). *Let p_0 and p be Ψ-secure and Φ-secure programs, respectively, such that p_0 has deterministic leakage. Then, for every input states x_1, x_2 such that $\Psi(x_1, x_2)$, $\langle p_0, x_1 \rangle \Downarrow_{l_1} \tilde{y_1}$, $\langle p_0, x_2 \rangle \Downarrow_{l_2} \tilde{y_2}$ and all output states y:*

$$\Pr_{y_1 \leftarrow \tilde{y_1}}[\Phi(y_1, y)] = \Pr_{y_2 \leftarrow \tilde{y_2}}[\Phi(y_2, y)]$$

Proof. Let x_1 and x_2 be such that $\Psi(x_1, x_2)$, and $\delta_i(l)$ be the probability that executing $p_0; p$ on initial memory x_i yields leakage trace l. We want to show that $\delta_1(l) = \delta_2(l)$ for every leakage trace l. Since p_0 has deterministic leakage and is Ψ-secure, there exists a unique l_0 such that $\langle p_0, x_1 \rangle \Downarrow_{l_0} \tilde{y_1}$ and $\langle p_0, x_2 \rangle \Downarrow_{l_0} \tilde{y_2}$. Trivially, $\delta_1(l) = \delta_2(l) = 0$ if l_0 is not a prefix of l so we assume that $l = l_0 \cdot l'$. By unfolding the instrumented semantics of program composition, we get:

$$\delta_i(l_0 \cdot l') = \sum_{\{y|\ \langle p, y \rangle \Downarrow_{l'}\}} \Pr[\langle p_0, x_i \rangle \Downarrow_{l_0} y]$$

where we write $\langle p, y \rangle \Downarrow_l$ as shorthand for $\exists z.\ \langle p, y \rangle \Downarrow_l z$.

Let $\{y_1, \ldots, y_k\}$ be an enumeration of the equivalence classes of Φ whose elements satisfy $\langle p, y \rangle \Downarrow_{l'}$. Formally, $\{y_1, \ldots, y_k\}$ is a maximal set such that $\neg\Phi(y_i, y_j)$ for every $1 \leq i, j \leq k$ with $i \neq j$ and $\langle p, y_j \rangle \Downarrow_{l'}$ holds for every $1 \leq j \leq k$. Since p is Φ-secure, the above identity can be rewritten as:

$$\delta_i(l_0 \cdot l') = \sum_{1 \leq j \leq k} \sum_{\{y|\ \Phi(y, y_j)\}} \Pr[\langle p_0, x_i \rangle \Downarrow_{l_0} y]$$

$$= \sum_{1 \leq j \leq k} \sum_{\{y|\ \Phi(y, y_j)\}} \Pr_{y \leftarrow \tilde{y_i}}[\Phi(y, y_j)]$$

where $\tilde{y_i}$ satisfies $\langle p_0, x_i \rangle \Downarrow_{l_0} \tilde{y_i}$. By assumption,

$$\Pr_{y_1 \leftarrow \tilde{y_1}}[\Phi(y, y_1)] = \Pr_{y_2 \leftarrow \tilde{y_2}}[\Phi(y, y_2)]$$

for every y, from which we immediately conclude.

---

[13] We implicitly assume that programs are safe, so that we consider distributions rather than sub-distributions, and leakage traces have bounded length. A more precise semantics would consider the joint distribution of $\tilde{y}$ and $\tilde{l}$, but this is not required for our purposes.
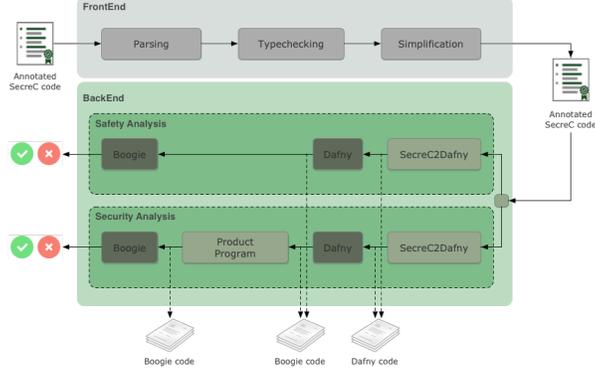
Fig. 9: SecreC verification infrastructure.

In particular, Theorem 4 holds for all pre-processing functions that yield uniformly distributed outputs. Technically, for every x, $\tilde{y}$, y_1 and y_2 [7]:

$$\langle p_0, x \rangle \Downarrow_l \tilde{y} \quad \Rightarrow \quad \Pr_{y \leftarrow \tilde{y}}[\Phi(y_1, y)] = \Pr_{y \leftarrow \tilde{y}}[\Phi(y_2, y)]$$

For completeness, we also provide a correctness criterion for leakage cancelling.

Definition 7 (Correct pre-processing). *A probabilistic program p_0 is a correct pre-processing for a deterministic program p iff for every initial memory x*

$$\langle p, x \rangle \Downarrow_l y \wedge \langle p_0; p, x \rangle \Downarrow_{l'} \tilde{y} \Longrightarrow \tilde{y} = \mathbf{1}_y$$

where $\mathbf{1}_y$ is the Dirac distribution assigning probability 1 to y and 0 to all other elements.

Finding a preprocessing program p_0 that satisfies the leakage cancelling conditions is often simple, and the practical benefits have been extensively compared in [13, 16, 29, 51]. For example, for sorting and all algorithms based on array comparisons, leaking the results of these comparisons can be cancelled by randomly pre-shuffling the array (assuming that all elements are distinct). For this optimization, it is critical that there exist (relatively) efficient oblivious shuffling protocols for the random shuffle operator (without any leakage). In platforms such as Sharemind [17], these protocols are offered as probabilistic instruction extensions at the source level, of the otherwise deterministic source language.

Automated verification of leakage cancelling is left for future work. Deterministic leakage can be enforced by an adaptation of the information flow type system for deterministic programs and the remaining conditions can be established using probabilistic relational Hoare logic [8], using tools like EasyCrypt [6].

We conclude with a discussion of how to extend our secure compilation to encompass leakage cancelling. Lifting this result for probabilistic programs is out of the scope of this paper. However, there is a simple extension for probabilistic preprocessing scenarios. Since the preprocessing stage has no leakage, by leakage cancelling this is also the case for the final composed protocol. Assuming the existence of an atomic preprocessing protocol, the extended compiler simply prepends it to the compiled protocol. The leakage cancelling theorem then implies the existence of a cryptographic simulator that works as follows: sample any input and execute the program p to obtain some leakage; then run the simulator described in Theorem 3 to obtain a simulated trace. Intuitively, by leakage cancelling any input will lead to a leakage trace that is distributed exactly as in the real-world, and hence the simulator for the compiled protocol suffices.

9 Implementation

This section describes our verification infrastructure for source-level analysis of deterministic MPC specifications: the *frontend* translates SecreC programs into an intermediate language close to Figure 2; the

backend deploys our source-level analysis on top of the Dafny-Boogie verification toolchain. The architecture of our verification infrastructure is illustrated in Figure 9. The development is available at `https://github.com/haslab/SecreC`.

Using our infrastructure, A non-MPC expert is able to construct a SecreC program, annotate it and prove a leakage upper-bound for the program that justifies all the declassify statements as a function of the input. MPC experts typically come into play in the case of leakage cancelling, which at the moment is not supported by our verification infrastructure. For code without declassify statements, the type system trivially suffices for guaranteeing security. Security verification is required for snippets involving declassify statements. Nevertheless, guaranteeing cryptographic security of compiled protocols programs requires proving safety and termination of SecreC programs, which is a classical deductive verification process.

9.1 Frontend

Our verification infrastructure supports the SecreC language bundled with the Sharemind SDK. This enables programmers to use our verification tools together with the Sharemind interpreter, compiler, secure execution engine and cloud deployment services. The internal operation of the Sharemind system is much more complex than the formal view of the compiler presented in this paper, as it was designed to allow for a high-degree of generality and flexibility with respect to low-level protocols, data types and operations. For instance, it allows linking external protocols that are secure according to Definition 5. The main performance distinction between our formal language and Sharemind-bundled SecreC lies in the fact that we do not capture an optimization that groups operations in a SIMD style in order to save communication rounds. Even so, the general principles of the Sharemind operation match the compilation strategy we have described in our formalization, as can be seen by the cryptographic security arguments that support the system [17], which are given at a comparable (if not higher) level of abstraction.

Much like C++, SecreC supports high-level programming features such as procedures, arrays, templates or recursion, and domain-specific support for array programming and security type polymorphism. Seeing SecreC programs as specifications of ideal secure functionalities, it becomes natural to express the security properties directly in the SecreC language. For that purpose, we have extended SecreC with an annotation language inspired by Dafny [35], a general-purpose verification language with support for procedures, loops, arrays, user-defined datatypes and native collection theories.

We have implemented a parser and a typechecker for our extended SecreC language in Haskell, and the typechecking algorithm for security type polymorphism and templates greatly resembles the treatment of ad-hoc polymorphism and type classes in Haskell. After typechecking, we apply a series of SecreC-to-SecreC simplification steps, such as removing implicit (subtyping) coercions or inlining template applications.

9.2 Backend

The backend translates an annotated SecreC intermediate program into two complementary Dafny programs: the first encodes functional correctness; the second assumes functional correctness and encodes security.

Functional embedding As we have seen above, cryptographic MPC specifications must be (by definition) total, and our notions of source-level security are termination-insensitive. Therefore, our functional embedding of SecreC into Dafny always checks that a SecreC program is safe. It preserves the original program structure and is almost one-to-one, reducing the functional correctness of SecreC programs to that of the Dafny embedding. Under the hood, the Dafny verifier checks for functional correctness by translating to Boogie code.

Security embedding Our SecreC specification language allows programmers to express leakage upper bounds and their flow through a program as annotations using the `public` keyword. These look like standard assertions but have a relational interpretation in the Security Hoare Logic from Section 3.

The security analysis explores the existing translation from Dafny to Boogie to propagate security properties from SecreC to Boogie programs. We adapt the constant-time verification approach from [1] and implement a Boogie-to-Boogie transformation that computes a product program (for a SecreC program with

public control flow). In the Boogie input language, procedures are defined as a sequence of basic blocks that start with a label, contain straight-line statements, and may jump at the end. For each procedure, we make shadow copies of program variables and duplicate all statements inside basic blocks to mention shadow variables instead, with two exceptions: i. procedure call statements are converted to single statements calling the product procedure with twice as many inputs and outputs; and ii. security assertions are translated to relational assertions expressing first-order logic formulas that relate original and shadowed variables, by translating `public(e)` expressions to equality expressions `e == e.shadow`. Proving that the product program is safe ensures that the original program is secure, as detailed in Appendix A.

10   Experiments

We have evaluated our infrastructure by analyzing existing SecreC specifications that are publicly available as part of the Sharemind SDK.[14] Some of these examples leak information that is subsequently cancelled using oblivious shuffling as described in Section 8. Here, the source analyser plays an important role, as it permits checking that the specification satisfies a leakage upper-bound that is compatible with the leakage cancelling theorem statement. In the remaining examples, proving a leakage bound permits matching the specification to the application requirements. We now demonstrate how to verify leakage bounds using our tool. We also discuss how the leakage cancelling steps can be performed manually, at the moment with no tool support.

Quick sort Comparison-based sorting is a very heavy operation to execute obliviously in a naive way, due to the high number of oblivious branches that it involves. However any sorting specification that declassifies the results of comparing vector elements, but nothing more, gives rise to leakage that can be cancelled using oblivious shuffling of the vector prior to sorting [29]. We have proven a deterministic quick sort [16] (Section 2) safe and secure with the Φ_{cmp} security policy from Section 3 that leaks all comparisons between array elements. For cancelling this leakage, it suffices to show that an oblivious shuffle for an array of distinct elements induces a uniform distribution on Φ_{cmp} (Theorem 4), and that the final sorted array is the same regardless of the relative ordering of the inputs (Definition 7). The algorithm can be generalized to arbitrary arrays by using the index of each element in the input list as a tie-breaker [29].

Gaussian elimination Our more intricate case-study is an implementation of Gaussian elimination [13]:

```
secret uint maxFirstLoc(secret float[[1]] vec) {
    secret float best = vec[0]; secret uint idx = 0;
    for (uint i = 1; i < size(vec); i=i+1) {
        secret bool c = vec[i] > best;
        best = choose(c,vec[i],best);
        idx = choose(c,i,idx);
    }
    return declassify(idx); }
```

The algorithm receives a $k * k$ square matrix and an array of k coefficients, for $k > 0$, and solves a system of k linear equations by iterating over the columns of the matrix. For each column j, it finds the row i of the pivot (the first maximum absolute value) using a procedure that performs all comparisons obliviously, and declassifies the output. Then, it shuffles the rows i and j, performs standard matrix arithmetic on the values of the underlying rows, repeating this process until the $k - 1$-th column. We first proved that the source-level declassification trace, constructed by instrumenting the source program with ghost code, consists of a permutation of the row indices. We then proved that the vector value indexed by the output of the `maxFirstLoc` function is constant for all possible permutations of the input vector. By combining these results we can easily derive that the oblivious shuffle pre-processing is sufficient to cancel the leakage under the results presented in Section 8.

---

[14] https://github.com/sharemind-sdk/secrec

Radix sort As an alternative to comparison-based sorting, we have proven the safety and security of an oblivious radix sort [16]. The implementation has an outer loop that iterates over the bit representation of the vector elements and, for each bit, operates as follows: it randomly shuffles the vector, computes a permutation that sorts the array according to the i-th bit, declassifies the permutation, and applies it in public to the vector. We prove a leakage upper-bound for the loop body that exactly matches the leakage of the permutation induced by the i-th bit. The leakage cancelling theorem then implies that the random shuffling preprocessing is sufficient to cancel this leakage in each iteration. By composition of the loop body, we get security for the entire algorithm.

Frequent itemsets Finally, we have analyzed a frequent itemset algorithm that searches for co-occurring items in transactional data. Given a boolean matrix encoding of items occurring in transactions, it computes all the itemsets up to a given size k whose frequency is above a certain threshold f, revealing those itemsets. Concretely, we studied the apriori algorithm from [12], which is based on level-wise search. The algorithm computes all itemsets of size 1 up to k, using the itemsets of size $k-1$ and cached numbers of occurrences thereof to compute the itemsets of size k. For efficiency, it declassifies all the comparison tests of wether itemsets of size up to k are above the frequency. We have proven that its SecreC implementation is secure with a leakage upper bound that releases only whether every itemset up to k is frequent. Although not exercised, we could additionally prove a functional correctness property that the algorithm publishes in declassified form exactly all the frequent itemsets up to k. This match between declassified output and leakage upper bound means that the leakage of intermediate computations is benign and hence the performance benefit comes at no additional security cost.

Benchmarks To give an idea of the complexity of the algorithms and the required verification effort, we have measured the number of lines of code (LOC) and the number of proof obligations (PO) for the Boogie code generated from a user-annotated SecreC program (Table 1).[15] The verification experience is in all similar to deductive verification environments such as Dafny, and requires typical programmer-supplied annotations for procedure contracts and loop invariants. Since security properties often depend on auxiliary functional correctness properties, we distinguish the verification effort for security annotations (S) from their required functional correctness annotations (F). We have also measured the average execution time for discharging the proof obligations over series of 10 runs on a standard MacBook Pro 2016 clocked at 2,9 GHz.

SecreC	LOC	PO		Time (s)	
		F	S	F	S
quick-sort	101	161	464	2.182	3.484
radix-sort	135	322	911	1.618	3.223
apriori	414	3877	6104	100.335	19.692
gaussian	178	610	-	2.789	-

Table 1: Verification results.

The number of security POs tends to be twice as large as the functional correctness ones, because our Boogie product program transformation duplicates all functions and statements without security annotations. For apriori, we have placed a significant effort in proving the soundness of the caching process and the observation that all frequent itemsets are reachable by adding single items to discovered frequent itemsets,

---

[15] Our tool generates a single Boogie file, including encodings of standard SecreC functions and Dafny builtin theories, that are implicitly imported and replicated for all examples. Thus, we measure only proof obligations originating from each SecreC example file.

as these are crucial for justifying in the security analysis that leaked information indeed corresponds to itemsets from the original database. To prove security of our gaussian example, it remains to show that our manually-instrumented leakage trace exactly matches the program's leakage trace. We could then elegantly tie source-level leakage as the permutation of the input defined by our instrumented trace. To support this, we are currently extending our implementation to handle output-dependent reasoning as in [1].

11 Related work

Some authors use language-based verification methods for optimizing MPCs. [32] relies on epistemic modal logic to infer public intermediate values. [45] proved a similar approach sound and complete for a simple functional language. Others focus on the specification and security enforcement of MPC protocols. [34, 43], develop a domain-specific language for writing low-level SecreC protocols and a sound data flow analysis to prove the security of generated protocols w.r.t. Definition 5. [44, 46] develop the Wysteria domain-specific language to write mixed-mode MPC protocols, and give an embedding into F*. Their approach is similar to ours in that the computational behavior of programs is given by a single-threaded and a multi-threaded semantics, which are formally related. However, their notion of security is left implicit and unrelated to the simulation-based notions. We also remark that our language works at a different level of abstraction. It specifies a MPC program as a composition of high-level MPC operations (that compute an ideal functionality typically agnostic of parties). Orthogonally, the languages of [34] and [44] describe lower-level MPC protocols that give meaning to executing a MPC operation in a distributed cryptographic environment (they have notions of shares and parties). These can be integrated as primitive operations in our language. Recently, Haagh et. al [28] have provided a machine-checked proof of a concrete MPC protocol against active adversaries. They present non-interference definitions that adapt our low-level definitions to the active case, but do not consider the problem of compilation from high-level specifications.

Our approach is based on the same idea of secure compilation explored in [40]. On top of the differences highlighted in the introduction, they consider a slightly weaker security model where the adversary cannot select the initial shares. Moreover, they do not address the problem of verifying leakage of source programs. In our approach, leakage is made explicit and verified at source-level, providing early feedback to developers.

Leakage cancelling is a standard technique for optimizing MPC programs [13, 16, 29, 51]. However, these works are cast in the cryptographic setting, not supported by language-based verification methods. Leakage cancelling has also been considered for oblivious RAM [52] and secure hardware [42]. Both works yield provable guarantees; however, their setting is different.

We now provide a more detailed review of less directly related work.

11.1 Programming Languages

Several works have studied the problem of securely compiling information flow secure programs into a distributed setting [27, 53]. However, their security models are rather different from those standard to MPC and require a simpler information flow analysis of source programs.

In addition, language-based techniques have been developed for related problems, including authenticated data structures [39], memory obliviousness [36], differential power analysis [3, 4], cache-based side-channel analysis [1], resource-based side-channel analysis [41]. However, the usual notions of security in these settings do not require reasoning about arbitrary leakage.

Finally, there is a significant body of work on language-based accounts of declassification; the survey [49] summarizes some key developments until 2009. [2] developed a relational logic for information flow (without declassification). However, these works are not directly applicable to our setting.

11.2 Cryptography

The development of secure computation protocols for real-world usability has long been an ongoing challenge in cryptography. We now present relevant work on frameworks enabling the deployment of applications relying on SC.

FairplayMP [10] is a multiparty computation system where C-like programs are translated to garbled boolean circuits. The evaluation of these circuits follows the BMR protocol [9], in which private inputs are secret shared, and garbled circuit computation is performed over the shared values.

Tools such as TASTY [30] and ABY [26] are tool suites addressing secure computation in the passive adversary model. These frameworks leverage a hybrid approach, combining garbled circuits, boolean secret sharing and arithmetic secret sharing, to take advantage of the best aspects of each of these approaches.

FRESCO [23] is an open-source Java framework allowing secure computation, where functions are described as abstract circuits. The actual gate-by-gate evaluation can be instantiated at runtime by specifying MPC protocols to be used for each gate. The existing protocol suites of FRESCO consist of two arithmetic secret sharing based protocols: the BGW suite [11], and the on-line phase of SPDZ [25].

12 Conclusion

We gave a language-based security treatment of secret sharing-based MPC software stacks. We showed that our notion of source-level security is propagated, via security-aware compilation, to cryptographically secure protocols. We also provided both a formalization and an implementation of a verification technique for source-level leakage analysis in real-world examples.

Our results leave room for a number of future research directions. One interesting open problem is to investigate novel language-based approaches to deal with active adversaries. Another promising direction (following our formalization) is to implement a certified compiler or to prove an existing compiler correct. Finally, another interesting line of work is to extend our theoretical framework and tools to fully cover probabilistic specifications.

References

1. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, pages 53–70, 2016.
2. Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Proceedings of the 33rd Symposium on Principles of Programming Languages*, pages 91–102. ACM, 2006.
3. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Proceedings, Part I of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9056, pages 457–485. Springer, 2015.
4. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 23rd Conference on Computer and Communications Security*, pages 116–129. ACM, 2016.
5. Gilles Barthe, Juan Manuel Crespo, and César Kunz. Product programs and relational program logics. *Journal of Logical and Algebraic Methods in Programming*, 85(5):847–859, 2016.
6. Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604, pages 146–166. Springer, 2013.
7. Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving uniformity and independence by self-composition and coupling. In *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. In print, 2017.
8. Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Symposium on Principles of Programming Languages*, pages 90–101. ACM, 2009.
9. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the 22nd Annual Symposium on Theory of Computing*, pages 503–513. ACM, 1990.
10. Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *Proceedings of the 15th conference on Computer and Communications Security*, pages 257–266. ACM, 2008.
11. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual Symposium on Theory of Computing*, pages 1–10. ACM, 1988.

12. Dan Bogdanov, Roman Jagomägis, and Sven Laur. Privacy-preserving histogram computation and frequent itemset mining with sharemind. Technical report, Cybernetica research report T-4-8, 2009.

13. Dan Bogdanov, Liina Kamm, Swen Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. *IEEE Transactions on Dependable and Secure Computing*, 2016.

14. Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From input private to universally composable secure multi-party computation primitives. In *Proceedings of the 27th Computer Security Foundations Symposium*, pages 184–198. IEEE, 2014.

15. Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-polymorphic programming of privacy-preserving applications. In *Proceedings of the 9th Workshop on Programming Languages and Analysis for Security*, page 53. ACM, 2014.

16. Dan Bogdanov, Sven Laur, and Riivo Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *Proceedings of the 19th Nordic Conference on Secure IT Systems*, pages 59–74. Springer, 2014.

17. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

18. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

19. Geoffroy Couteau. Efficient secure comparison protocols. *IACR Cryptology ePrint Archive*, 2016:544, 2016.

20. Geoffroy Couteau. Revisiting covert multiparty computation. *IACR Cryptology ePrint Archive*, 2016:951, 2016.

21. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation, an introduction. *Contemporary cryptology*, pages 41–87, 2005.

22. Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, pages 169–187. Springer, 2016.

23. Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, pages 169–187. Springer, 2016.

24. Ivan Damgård and Jesper Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *Proceedings of the 23rd Annual International Cryptology Conference*, pages 247–264. Springer, 2003.

25. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proceedings of the 32nd International Cryptology Conference*, pages 643–662. Springer, 2012.

26. Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Proceedings of the 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.

27. Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In *Proceedings of the 16th Conference on Computer and Communications Security*, pages 432–441. ACM, 2009.

28. Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-aided proofs for multiparty computation with active security. In *Proceedings of the 31st Computer Security Foundations Symposium*, page In print. IEEE.

29. Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *Proceedings of the 15th International Conference on Information Security and Cryptology*, pages 202–216. Springer, 2012.

30. Wilko Henecka, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *Proceedings of the 17th Conference on Computer and Communications Security*, pages 451–462. ACM, 2010.

31. Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, 14(6):531–548, 2015.

32. Florian Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th Conference on Computer and Communications Security*, pages 703–714. ACM, 2011.

33. Varsha Bhat Kukkala, Jaspal Singh Saini, and SRS Iyengar. Privacy preserving network analysis of distributed social networks. In *Information Systems Security*, pages 336–355. Springer, 2016.

34. Peeter Laud and Jaak Randmets. A domain-specific language for low-level secure multiparty computation protocols. In *Proceedings of the 22nd Conference on Computer and Communications Security*, pages 1492–1503. ACM, 2015.

35. K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 17th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

36. Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *Proceedings of the 26th Computer Security Foundations Symposium*, pages 51–65. IEEE, 2013.

37. Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient ram-model secure computation. In *Proceedings of the 35th Symposium on Security and Privacy*, pages 623–638. IEEE, 2014.

38. Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *Proceedings of the 36th Symposium on Security and Privacy*, pages 359–376. IEEE, 2015.

39. Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *Proceedings of the 41st Symposium on Principles of Programming Languages*, pages 411–423. ACM, 2014.

40. John C Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. Information-flow control for programming on encrypted data. In *Proceedings of the 25th Computer Security Foundations Symposium*, pages 45–60. IEEE, 2012.

41. Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *Proceedings of the 38th Symposium on Security and Privacy*, pages 710–728. IEEE, 2017.

42. Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *Proceedings of the 22nd Conference on Computer and Communications Security*, pages 1570–1581. ACM, 2015.

43. Martin Pettai and Peeter Laud. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In *Proceedings of the 28th Computer Security Foundations Symposium*, pages 75–89. IEEE, 2015.

44. Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proceedings of the 35th Symposium on Security and Privacy*, pages 655–670. IEEE, 2014.

45. Aseem Rastogi, Piotr Mardziel, Michael Hicks, and Matthew A Hammer. Knowledge inference for optimizing secure multi-party computation. In *Proceedings of the 8th Workshop on Programming Languages and Analysis for Security*, pages 3–14. ACM, 2013.

46. Aseem Rastogi, Nikhil Swamy, and Michael Hicks. Wys*: A verified language extension for secure multi-party computations. *arXiv preprint arXiv:1711.06467*, 2017.

47. M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. *arXiv preprint arXiv:1801.03239*, 2018.

48. Andrei Sabelfeld and Andrew C Myers. A model for delimited information release. In *Software Security-Theories and Systems*, pages 174–191. Springer, 2004.

49. Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

50. Axel Schropfer, Florian Kerschbaum, and Gunter Muller. L1-an intermediate language for mixed-protocol secure computation. In *Proceedings of the 35th Annual Computer Software and Applications Conference*, pages 298–307. IEEE, 2011.

51. Shruti Tople, Hung Dang, Prateek Saxena, and Ee-Chien Chang. PermuteRam: Optimizing oblivious computation for efficiency, 2015.

52. Shruti Tople, Hung Dang, Prateek Saxena, and Ee-Chien Chang. Permuteram: Optimizing oblivious computation for efficiency. Cryptology ePrint Archive, Report 2017/885, 2017. https://eprint.iacr.org/2017/885.

53. Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th Symposium on Operating System Principles*, pages 1–14. ACM, 2001.

Appendix A Source-level security verification

This section gathers additional definitions and observations that complete our formalization of source-level security.

Appendix A.1 Source-level language

The source-level semantics of expressions, with no leakage, is defined as:

$$[\![v]\!](m) = v$$
$$[\![x]\!](m) = m(x)$$
$$[\![x[e]]\!](m) = m(x)[[\![e]\!](m)]$$
$$[\![e_1 \, \mathsf{pop} \, e_2]\!](m) = \mathsf{pop}\,([\![e_1]\!](m), [\![e_2]\!](m))$$

As shorthand for the evaluation of programs, $\langle p, m \rangle \Downarrow m'$ ignores leakage.

Appendix A.2 Security type system

We recall the standard notion of non-interference security [49], which states that public values do not depend on secret values.

Definition 8 (Non-interference). *Let V_{public} denote a set of variables that are public in the (implicit) security environment Γ. A program is (termination-insensitive)* non-interferent *whenever:*

$$m_1 \stackrel{\mathsf{public}}{=} m_2 \,\wedge\, \left(\begin{array}{c} \langle p, m_1 \rangle \Downarrow m_1' \\ \langle p, m_2 \rangle \Downarrow m_2' \end{array} \right) \Rightarrow m_1' \stackrel{\mathsf{public}}{=} m_2'$$

We define public-*equivalence of memories extensionally:*

$$m_1 \stackrel{\mathsf{public}}{=} m_2 \triangleq \bigwedge_{x \in V_{\mathsf{public}}} m_1(x) = m_2(x)$$

Note that this notion of non-interference does not consider leakage. Our SecreC type satisfies non-interference for programs without declassification:

Theorem 5. *If $\Gamma \vdash p$ and p has no declassifies, then p is non-interferent.*

For programs that declassify intermediate values, the standard notion of leakage-aware non-interference is known as delimited release [48]:

Definition 9 (Delimited release). *A program satisfies delimited release whenever:*

$$\left(\begin{array}{c} m_1 \stackrel{\mathsf{public}}{=} m_2 \\ l_1 = l_2 \end{array} \right) \,\wedge\, \left(\begin{array}{c} \langle p, m_1 \rangle \Downarrow_{l_1} m_1' \\ \langle p, m_2 \rangle \Downarrow_{l_2} m_2' \end{array} \right) \Rightarrow m_1' \stackrel{\mathsf{public}}{=} m_2'$$

The intuition is that information leakage is allowed through all declassify "escape hatches", by only talking about the security of programs having the same leakage traces. Both our type system and that of [40] guarantee this form of delimited release.

Theorem 6. *If $\Gamma \vdash p$, then p satisfies delimited release.*

Proof. By induction on the typing derivation for p. For if and while programs, since branch conditions are in the leakage trace, both computations are guaranteed to always take the same branch, and we proceed by induction. In the case of assignments to public lvalues, the right side is either the result of a declassify, and is in the leakage trace, or the security levels of all variables occurring in the right-side expression is public, guaranteeing that the result of the expression is also public.

Note that secret information is only allowed to leak in declassify expressions, since public control flow (that is also in the leakage trace) is required for all well-typed programs.

Our notion of source-level security is strictly stronger than delimited release, in the sense that it can establish an upper bound on the amount of leakage.

Theorem 7. *A (Φ, Ψ)-source-secure, with $\Phi \Rightarrow \overset{\mathsf{public}}{=}$ and $\overset{\mathsf{public}}{=} \Rightarrow \Psi$, satisfies delimited release.*

Proof. From the fact that source security implies equal traces.

Appendix A.3 Product programs

Source-level security can be proven in two steps: reduce the relational properties of two executions of the program to the properties of the program composed with itself; and then use standard verification tools to prove the properties of the self-composed program.

For the purpose of verification of the corresponding product programs, we add regular assert and assume statements to our source language:

$$\frac{\llbracket \phi \rrbracket(m) = \mathsf{true}}{\langle \mathsf{assume}\, \phi, m \rangle \to_\varepsilon \langle \mathsf{skip}, m \rangle}$$

$$\frac{\llbracket \phi \rrbracket(m) = \mathsf{true}}{\langle \mathsf{assert}\, \phi, m \rangle \to_\varepsilon \langle \mathsf{skip}, m \rangle}$$

$$\frac{\llbracket \phi \rrbracket(m) = \mathsf{false}}{\langle \mathsf{assert}\, \phi, m \rangle \to_\varepsilon \langle \mathsf{skip}, \bot \rangle}$$

and define an axiomatic semantics as a standard Hoare Logic (Figure 10). We introduce an error state in our source-level semantics to model assertion failure; assume failure is undefined. Configurations in the small-step semantics now take memories m to extended memories $m'_\bot$, that can be a valid memory m' or an error state $\bot$. All our semantic rules then become implicitly guarded by the rule $m \neq \bot$.

Definition 10 (Hoare Triple). *A hoare triple $\{\psi\}\, p\, \{\psi\}$ is valid iff for all states m, m'*

$$\phi(m) \wedge \langle p, m \rangle \Downarrow m' \Rightarrow \psi(m')$$

$$\frac{}{\vdash \{\phi\}\ \mathsf{skip}\ \{\phi\}} \qquad \frac{\vdash \{\phi\}\, p_1\, \{\psi\} \quad \vdash \{\psi\}\, p_2\, \{\theta\}}{\vdash \{\phi\}\, p_1; p_2\, \{\theta\}}$$

$$\frac{\vdash \{\phi \wedge e\}\, p_1\, \{\psi\} \quad \vdash \{\phi \wedge \neg e\}\, p_2\, \{\psi\}}{\vdash \{\phi\}\ \mathsf{if}\ e\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2\ \{\psi\}}$$

$$\frac{\vdash \{\phi \wedge e\}\, p\, \{\phi\}}{\vdash \{\phi\}\ \mathsf{while}\ e\ \mathsf{do}\ p\ \{\phi \wedge \neg e\}}$$

$$\frac{}{\vdash \{\psi[ae/lv]\}\ lv := ae\ \{\psi\}}$$

$$\frac{}{\vdash \{\phi \Rightarrow \psi\}\ \mathsf{assume}\, \phi\, \{\psi\}} \qquad \frac{}{\vdash \{\phi \wedge \psi\}\ \mathsf{assert}\, \phi\, \{\psi\}}$$

$$\frac{\vdash \{\phi\}\, p\, \{\psi\} \quad \phi' \Rightarrow \phi \quad \psi \Rightarrow \psi'}{\vdash \{\phi'\}\, p\, \{\psi'\}}$$

Fig. 10: Inference system for Hoare logic.

Remember Theorem 1 that attests the soundness and completeness of Security Hoare Logic:

$$\text{product}(\text{skip}) = \text{skip}$$
$$\text{product}(p_1; p_2) = \text{product}(p_1); \text{product}(p_2)$$
$$\text{product}(\text{if } e \text{ then } p_1 \text{ else } p_2) =$$
$$\quad \text{assert } e = \hat{e};$$
$$\quad \text{if } e \text{ then product}(p_1) \text{ else product}(p_2)$$
$$\text{product}(\text{while } e \text{ do } p) =$$
$$\quad \text{assert } e = \hat{e};$$
$$\quad \text{while } e \text{ do } (\text{product}(p); \text{assert } e = \hat{e})$$
$$\text{product}(lv := e) =$$
$$\quad lv := e; \quad \hat{lv} := \hat{e}$$

$$\text{product}(lv := e_1 \text{ sop } e_2) =$$
$$\quad lv := e_1 \text{ sop } e_2;$$
$$\quad \hat{lv} := \hat{e_1} \text{ sop } \hat{e_2}$$
$$\text{product}(lv := \text{declassify}(e)) =$$
$$\quad \text{assert } e = \hat{e};$$
$$\quad lv := \text{declassify}(e);$$
$$\quad \hat{lv} := \text{declassify}(\hat{e})$$
$$\text{product}(\text{assume } \phi) = \text{assume } \phi \wedge \hat{\phi}$$
$$\text{product}(\text{assert } \phi) = \text{assert } \phi \wedge \hat{\phi}$$

Fig. 11: Product programs for SecreC.

Theorem 8. $\models \{\Phi\} \, p \, \{\Psi\}$ *is derivable iff* $\{\Phi\} \, p \, \{\Psi\}$.

Proof. SHL is equivalent to minimal Relational Hoare Logic [5], plus declassification. Soundness is trivial and relative completeness w.r.t. the expressiveness of the assertion language comes from source-level security enforcing that both copies execute in lockstep.

Figure 11 shows how to construct a product program for any source-level SecreC program. The notation $\hat{x}$ is used to denote the copy of variable x in the product program. Relational security triples are also in one-to-one correspondence with product programs:

Theorem 9. $\models \{\Phi\} \, p \, \{\Psi\}$ *iff* $\vdash \{\Phi\} \, \text{product}(p) \, \{\Psi\}$

Using the product program representation, we can use standard deductive program verification tools for verifying source-level security:

Lemma 5. *A program* $\Gamma \vdash p$ *is* (Φ, Ψ)-*secure iff* p' *is safe, with* p' *defined as:*

$$\text{assume } \Phi; \ \text{product}(p); \ \text{assert } \Psi$$

The soundness and practical efficiency of this technique from hinge on the assumption that program paths are independent from program secrets, guaranteeing that the product program always has as many paths as the original program. Note that this is always true for the SecreC language that, to enable the generation of efficient MPC protocols, already requires all branching conditions to be public.

Appendix B   Universally composable security

We will show that our protocols UC-realize the ABB functionality, but our assumption on low level protocols is that they satisfy a privacy notion. We therefore begin by formalizing functionalities $\mathcal{F}\langle \text{op}, \ell \rangle$ and $\mathcal{F}\langle p, \ell \rangle$, denoted as $\mathcal{F}_{\text{op}}$ and $\mathcal{F}_p$ for short. $\mathcal{F}_{\text{op}}$ will denote intermediate operations. These functionalities execute over values and perform exactly as in Figure 8. $\mathcal{F}_p$ will denote ABB functionalities, and will be used to demonstrate security of protocols.

We present a stronger simulation-based notion for secure computation protocols that matches a restricted setting of UC security, considering honest-but-curious adversaries, static corruptions and a threshold of a single corrupt party. Figure 13 presents the experiment that captures this property. Its behaviour is as follows. On both cases, the adversary begins by selecting input shares $\bar{x}$, the corrupt participant i, and some auxiliary state st. Then, either the protocol is executed (the real world), or $\bar{x}$ is given to the functionality $\mathcal{F}\langle p, \ell \rangle$, and

$$
\begin{array}{l|l}
\underline{\mathcal{F}\langle \mathsf{op}, \ell \rangle(x):} & \underline{\mathcal{F}\langle p, \ell \rangle(\bar{x}):} \\
 & x \leftarrow \mathrm{Unshare}(\bar{x}) \\
y \leftarrow \mathsf{op}(x) & y \leftarrow p(x) \\
l \leftarrow \ell(x) & l \leftarrow \ell(x, y) \\
 & \bar{y} \leftarrowtail \mathrm{Share}(y) \\
\mathrm{Return}\ (y, l) & \mathrm{Return}\ (\bar{y}, l)
\end{array}
$$

Fig. 12: Private and secure functionalities.

the simulator is executed with corrupt input x_i, corrupt output y_i and computation leakage l to emulate the corresponding trace and coins (the ideal world). Finally, the adversary receives the input shares $\bar{x}$, the output shares $\bar{y}$, the trace and coins of the corrupt participant t_i, c_i, and state st, and must produce a guessing bit.

Definition 11 (Cryptographic security). *We say a protocol Π is (ℓ, p)-crypto-secure if there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that, for all adversaries $\mathcal{A}$, the following definition of advantage is 0.*

$$
\mathbf{Adv}_{\Pi, \mathcal{F}_p, \mathcal{A}, \mathcal{S}} = \Pr[\mathsf{Real}_{\Pi, \mathcal{A}}] - \Pr[\mathsf{Ideal}_{\mathcal{F}_p, \mathcal{S}, \mathcal{A}}].
$$

$$
\begin{array}{l|l}
\textbf{game Real}_{\Pi, \mathcal{A}}(\,): & \textbf{game Ideal}_{\mathcal{F}\langle p, \ell \rangle, \mathcal{S}, \mathcal{A}}(\,): \\
\hline
(\bar{x}, i, \mathsf{st}) \leftarrowtail \mathcal{A}_1(\,) & (\bar{x}, i, \mathsf{st}) \leftarrowtail \mathcal{A}_1(\,) \\
 & (\bar{y}, l) \leftarrowtail \mathcal{F}\langle p, \ell \rangle(\bar{x}) \\
(\bar{y}, t, c) \leftarrowtail \Pi(\bar{x}) & (t, c) \leftarrowtail \mathcal{S}(i, l, x_i, y_i) \\
\mathrm{Return}\ \mathcal{A}_2(\bar{x}, \bar{y}, t_i, c_i, \mathsf{st}) & \mathrm{Return}\ \mathcal{A}_2(\bar{x}, \bar{y}, t_i, c_i, \mathsf{st})
\end{array}
$$

Fig. 13: Simulation-based security.

Following the approach of ABB functionalities, we must demonstrate that the composition of any (ℓ_1, op)-crypto-private protocol with a (ℓ_2, p)-crypto-secure protocol produces a $((\ell_1, \ell_2), p)$-crypto-secure protocol. This is a very similar result to the one presented in [14] and captures the notion that we are using private protocols as intermediate arithmetic steps, and before the computation output is provided, a secure protocol will ensure independence and freshness of output shares. Given that a resharing protocol such as π_{resh} (Figure 14) is both efficient and trivially secure for an identity functionality with zero leakage, we can easily convert our originally private protocols into secure ones.

$$
\begin{array}{l}
\underline{\pi_{\mathsf{resharing}}(\bar{u}):} \\
\quad \text{For } i = 1 \text{ to } n - 1 \\
\qquad \pi_i(u_i, \cdot, c_i): \text{Send } c_i \text{ to } P_{i+1} \\
\quad \pi_n(u_n, \cdot, c_n): \text{Send } c_n \text{ to } P_1 \\
\quad \pi_1(u_1, \cdot, c_1): u_1' \leftarrow u_1 + c_1 - c_n \\
\quad \text{For } i = 2 \text{ to } n \\
\qquad \pi_i(u_i, \cdot, c_i): u_i' \leftarrow u_i + c_i - c_{i-1} \\
\quad \mathrm{Return}\ (u_1, \dots, u_n)
\end{array}
$$

Fig. 14: Resharing protocol.

More specifically, we want to say that.

Theorem 10. *Let Π_1 be a protocol ensuring (ℓ_1, op)-crypto-privacy, and let Π_2 be a protocol ensuring (ℓ_2, p)-crypto-security. The protocol Π, sequentially executing Π_1, Π_2 provides $((\ell_1, \ell_2), (\mathsf{op}, p))$-crypto-security.*

Proof. Our proof is a sequence of two game hops, represented in Figure 15 and described as follows.

G0 is the real world, where Π is extended to sequentially execute Π_1 and Π_2. G1 replaces the behaviour of Π_2 with the associated functionality $\mathcal{F}_p$. We upper bound the difference between these two experiments by constructing an adversary $\mathcal{B}$ against the (l_2, p)-crypto-security of Π_2 such that

$$|\Pr[\mathsf{G1}(\,) \Rightarrow \mathsf{T}] - \Pr[\mathsf{G0}(\,) \Rightarrow \mathsf{T}]| = \mathsf{Adv}_{\Pi_2, \mathcal{F}_p, \mathcal{B}, \mathcal{S}_2}^{(\ell_2, p)-\sec}$$

Adversary $\mathcal{B}$ executes as follows. $\mathcal{B}_1$ follows the description of G1 until it produces $(\bar{y}, t_1, c_1)$, and selects $(\bar{y}, i, (\bar{x}, t_1, c_1))$ as input for the security experiment of Π_2. This will produce a tuple $(\bar{y}, \bar{z}, t_2, c_2)$, which $\mathcal{B}_2$ combines with the state $(\bar{x}, t_1, c_1)$ to run $\mathcal{A}_2$ and return the same result.

G2 replaces the behaviour of Π_1 with the associated functionality $\mathcal{F}_{\mathsf{op}}$. We upper bound the difference between these two experiments by constructing an adversary $\mathcal{C}$ against the (l_1, op)-crypto-privacy of Π_1 such that

$$|\Pr[\mathsf{G2}(\,) \Rightarrow \mathsf{T}] - \Pr[\mathsf{G1}(\,) \Rightarrow \mathsf{T}]| = \mathsf{Adv}_{\Pi_1, \mathcal{F}_{\mathsf{op}}, \mathcal{C}, \mathcal{S}_1}^{(\ell_1, \mathsf{op})-\mathrm{priv}}$$

Adversary $\mathcal{C}$ executes as follows. $\mathcal{C}_1$ runs $\mathcal{A}_1$ to construct $(\bar{x}, i, \mathsf{st})$ and select it as input for the privacy experiment of Π_1. This will produce a tuple $(\bar{x}, y, y_i, t_1, c_1)$, and $\mathcal{B}_2$ proceeds with the description of G1, computing the remaining values for $(z, l_2, \bar{z}, t_2, c_2)$, runs $\mathcal{A}_2$ and returns the same result.

Finally, we can rearrange G3 to capture a functionality $\mathcal{F}_p$ that executes op followed by p, and a simulator $\mathcal{S}$ that makes employs $(\mathcal{S}_1, \mathcal{S}_2)$. This experiment exactly matches the Ideal world.

game $\mathsf{G0}_{\Pi_1, \Pi_2, \mathcal{A}}(\,)$:	**game** $\mathsf{G1}_{\Pi_1, \ell_2, p, \mathcal{S}_2, \mathcal{A}}(\,)$:
$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$	$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$
$(\bar{y}, t_1, c_1) \twoheadleftarrow \Pi_1(\bar{x})$	$(\bar{y}, t_1, c_1) \twoheadleftarrow \Pi_1(\bar{x})$
	$y \leftarrow \mathrm{Unshare}(\bar{y})$
$(\bar{z}, t_2, c_2) \twoheadleftarrow \Pi_2(\bar{y})$	$z \twoheadleftarrow p(y)$
	$l_2 \leftarrow \ell_2(y, z)$
	$\bar{z} \twoheadleftarrow \mathrm{Share}(z)$
	$(t_2, c_2) \twoheadleftarrow \mathcal{S}_2(i, l_2, y_i, z_i)$
$t \leftarrow (t_1, t_2); \ c \leftarrow (c_1, c_2)$	$t \leftarrow (t_1, t_2); \ c \leftarrow (c_1, c_2)$
Return $\mathcal{A}_2(\bar{x}, \bar{z}, t_i, c_i, \mathsf{st})$	Return $\mathcal{A}_2(\bar{x}, \bar{z}, t_i, c_i, \mathsf{st})$

game $\mathsf{G1}_{\Pi_1, \ell_2, p, \mathcal{S}_2, \mathcal{A}}(\,)$:	**game** $\mathsf{G2}_{\ell_1, \mathsf{op}, \mathcal{S}_1, \ell_2, p, \mathcal{S}_2, \mathcal{A}}(\,)$:
$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$	$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$
$(\bar{y}, t_1, c_1) \twoheadleftarrow \Pi_1(\bar{x})$	$x \leftarrow \mathrm{Unshare}(\bar{x})$
$y \leftarrow \mathrm{Unshare}(\bar{y})$	$y \twoheadleftarrow \mathsf{op}(x)$
	$l_1 \leftarrow \ell_1(x)$
	$(y_i, t_1, c_1) \twoheadleftarrow \mathcal{S}_1(i, l_1, x_i)$
$z \twoheadleftarrow p(y)$	$z \twoheadleftarrow p(y)$
$l_2 \leftarrow \ell_2(y, z)$	$l_2 \leftarrow \ell_2(y, z)$
$\bar{z} \twoheadleftarrow \mathrm{Share}(z)$	$\bar{z} \twoheadleftarrow \mathrm{Share}(z)$
$(t_2, c_2) \twoheadleftarrow \mathcal{S}_2(i, l_2, y_i, z_i)$	$(t_2, c_2) \twoheadleftarrow \mathcal{S}_2(i, l_2, y_i, z_i)$
$t \leftarrow (t_1, t_2); \ c \leftarrow (c_1, c_2)$	$t \leftarrow (t_1, t_2); \ c \leftarrow (c_1, c_2)$
Return $\mathcal{A}_2(\bar{x}, \bar{z}, t_i, c_i, \mathsf{st})$	Return $\mathcal{A}_2(\bar{x}, \bar{z}, t_i, c_i, \mathsf{st})$

Fig. 15: Composition hops.

To conclude, we have that

$$\mathbf{Adv}_{\Pi,\mathcal{F}_p,\mathcal{A},\mathcal{S}}() = \mathsf{Adv}^{(\ell_2,p)-\sec}_{\Pi_2,\mathcal{F}_p,\mathcal{B},\mathcal{S}_2} + \mathsf{Adv}^{(\ell_1,\mathsf{op})-\mathrm{priv}}_{\Pi_1,\mathcal{F}_{\mathsf{op}},\mathcal{C},\mathcal{S}_1}$$
$$= 0$$

and Theorem 10 follows.

Appendix C   From static to adaptive adversaries

Our model considers static corruptions, which means that an adversary must fix which participant it is willing to corrupt before protocol execution. This is a more restrictive model that what is considered for adaptive corruptions, where the adversary can select the corrupt participant at any time during protocol execution [21]. The work in [14] discussed how privacy extends to adaptive adversaries. We now explain how our results also extend to this setting.

Since $\mathcal{A}$ can perform corruptions at any point of the execution, we refine the protocol description to capture every possible moment of corruption. More specifically, we denote Π to be an n-step protocol that maintains state $\mathsf{st}_\Pi = (\bar{x}, \bar{y}, t, c)$, containing the input shares, output shares, communication trace and random coins of all participants. For convenience, we also make use of st^i_Π to refer to the projection of this state corresponding to participant i. If no participant is given, the projection returns the empty state: $\mathsf{st}^\epsilon_\Pi = (\epsilon, \epsilon, [\,], [\,])$.

Figure 16 presents an alternative version of our privacy definition to consider adaptive corruptions. For clarity in presentation, we exclude all executions where $\mathcal{A}$ attempts to corrupt more than one participant (i.e. if $\mathcal{A}_2$ produces $i \neq \epsilon$, then it must always provide the same i from then onwards). Its behaviour is as follows. $\mathcal{A}_1$ begins by selecting input shares $\bar{x}$ and initializing some state $\mathsf{st}_\mathcal{A}$. Then, $\mathcal{A}_2$ will have $n + 1$ opportunities to select a corrupt participant i, which will allow him to receive the state projection st^i_Π. This projection is either the real protocol state for that party (the real world), or a simulated view (the ideal world). Observe that $\mathcal{S}$ will only have access to the execution leakage $l_\mathcal{S}$ and corrupt input x_i after the corruption has occurred (If $i \neq \epsilon$). At the end of the experiment $\mathcal{A}_3$ receives the same information as the privacy definition $(\bar{x}, y, \mathsf{st}^i_\Pi, \mathsf{st}_\mathcal{A})$, and must produce a guessing bit.

game $\mathbf{Real}_{\Pi,\mathcal{A}}()$:	**game $\mathbf{Ideal}_{\mathcal{F}\langle p,\ell\rangle,\mathcal{S},\mathcal{A}}()$:**
$(\bar{x}, \mathsf{st}_\mathcal{A}) \twoheadleftarrow \mathcal{A}_1()$	$(\bar{x}, \mathsf{st}_\mathcal{A}) \twoheadleftarrow \mathcal{A}_1()$
	$x \leftarrow \mathrm{Unshare}(\bar{x})$
$\mathsf{st}_\Pi \leftarrow (\bar{x}, \epsilon, [\,], [\,])$	$(\mathsf{st}_\mathcal{S}, \mathsf{st}_\Pi) \leftarrow \mathcal{S}_1()$
	$(l, y) \twoheadleftarrow \mathcal{F}\langle \ell, p\rangle(x)$
$i \leftarrow \epsilon$	$i \leftarrow \epsilon; l_\mathcal{S} \leftarrow \epsilon$
For $s \in [1..n]$:	For $s \in [1..n]$:
$\quad (\mathsf{st}_\mathcal{A}, i) \twoheadleftarrow \mathcal{A}_2(\mathsf{st}_\mathcal{A}, \mathsf{st}^i_\Pi)$	$\quad (\mathsf{st}_\mathcal{A}, i) \twoheadleftarrow \mathcal{A}_2(\mathsf{st}_\mathcal{A}, \mathsf{st}^i_\Pi)$
	$\quad$ If $i \neq \epsilon$: $l_\mathcal{S} \leftarrow l$
$\quad \mathsf{st}_\Pi \twoheadleftarrow \Pi(s, \mathsf{st}_\Pi)$	$\quad (\mathsf{st}_\mathcal{S}, \mathsf{st}_\Pi) \twoheadleftarrow \mathcal{S}_2(i, \mathsf{st}_\mathcal{S}, l_\mathcal{S}, x_i, \mathsf{st}_\Pi)$
$(\mathsf{st}_\mathcal{A}, i) \twoheadleftarrow \mathcal{A}_2(\mathsf{st}_\mathcal{A}, \mathsf{st}^i_\Pi)$	$(\mathsf{st}_\mathcal{A}, i) \twoheadleftarrow \mathcal{A}_2(\mathsf{st}_\mathcal{A}, \mathsf{st}^i_\Pi)$
$(\cdot, \bar{y}, \cdot, \cdot) \leftarrow \mathsf{st}_\Pi$	If $i \neq \epsilon$: $l_\mathcal{S} \leftarrow l$
$y \leftarrow \mathrm{Unshare}(\bar{y})$	$(\mathsf{st}_\mathcal{S}, \mathsf{st}_\Pi) \twoheadleftarrow \mathcal{S}_2(i, \mathsf{st}_\mathcal{S}, l_\mathcal{S}, x_i, \mathsf{st}_\Pi)$
Return $\mathcal{A}_3(\bar{x}, y, \mathsf{st}^i_\Pi, \mathsf{st}_\mathcal{A})$	Return $\mathcal{A}_3(\bar{x}, y, \mathsf{st}^i_\Pi, \mathsf{st}_\mathcal{A})$

Fig. 16: Cryptographic privacy in the presence of adaptive corruptions.

Definition 12 (Adaptive cryptographic privacy). *We say a protocol Π is (ℓ, p)-crypto-adaptive-private if there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that, for all adversaries $\mathcal{A}$, the following definition of advantage is 0.*

$$\mathbf{Adv}_{\Pi,\mathcal{F}_p,\mathcal{A},\mathcal{S}} = \Pr[\mathsf{Real}_{\Pi,\mathcal{A}}] - \Pr[\mathsf{Ideal}_{\mathcal{F}_p,\mathcal{S},\mathcal{A}}].$$

Despite our privacy results considering static corruptions, every Π ensuring cryptographic privacy will also do so in the presence of adaptive adversaries. More specifically, we say that

Theorem 11. *If Π is a protocol ensuring (ℓ, p)-crypto-privacy, then the same protocol Π ensures (ℓ, p)-crypto-adaptive-privacy.*

Proof. Let $\mathcal{S}_{\mathsf{priv}}$ be the (ℓ, p)-crypto-privacy simulator associated with Π, and let $\mathcal{S}$ be the (ℓ, p)-crypto-adaptive-privacy simulator associated with Π. We begin by describing how one can construct $\mathcal{S}$ from $\mathcal{S}_{\mathsf{priv}}$, and then argue how $\mathcal{S}$ is able to always present a consistent view to $\mathcal{A}$ for the experiment on Figure 16.

We assume that, given the description of Π and a full communication trace and coins (t, c), it is possible to infer which messages/coins were generated up to a certain protocol step s, and whether the participant is allowed to display its output share. We denote this selection process as $\mathsf{Sel}(\Pi, s, t, c)$, producing $(t', c') \subseteq (t, c)$ and boolean b, which is set as T if output is to be made available, and F otherwise. The simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ is described in Figure 17 and executes as follows.

$\mathcal{S}$ will maintain a counter for the protocol stage, and must present a protocol state consistent with the execution so far. If there is yet to be a corruption ($i = \epsilon$), the protocol state remains empty. Otherwise, the simulator will make use of the privacy simulator $\mathcal{S}_{\mathsf{priv}}$ to generate an output share y_i and full trace and coins (t, c). Then, $\mathsf{Sel}(\Pi, s, t, c)$ is used to ensure that every step produces a view consistent with the current protocol step.

$$
\begin{array}{l|l}
\underline{\mathcal{S}_1()}: & \underline{\mathcal{S}_2\langle\mathcal{S}_{\mathsf{priv}}\rangle(i, \mathsf{st}_\mathcal{S}, l_\mathcal{S}, x_i, \mathsf{st}_\Pi)}: \\
\quad \mathsf{st}_\mathcal{S} \leftarrow (1, \epsilon, [\,], [\,]) & \quad (s, y_i, t, c) \leftarrow \mathsf{st}_\mathcal{S} \\
\quad \mathsf{st}_\Pi \leftarrow (\epsilon, \epsilon, [\,], [\,]) & \quad \text{If } i \neq \epsilon: \\
\quad \text{Return } (\mathsf{st}_\mathcal{S}, \mathsf{st}_\Pi) & \qquad \text{If } y_i = \epsilon: \\
& \qquad\quad (y_i, t, c) \twoheadleftarrow \mathcal{S}_{\mathsf{priv}}(i, l_\mathcal{S}, x_i) \\
& \qquad\quad (b, t', c') \leftarrow \mathsf{Sel}(\Pi, s, t, c) \\
& \qquad\quad \text{If } b = \mathsf{T}: \mathsf{st}_\Pi \leftarrow (x_i, y_i, t', c') \\
& \qquad\quad \text{Else: } \mathsf{st}_\Pi \leftarrow (x_i, \epsilon, t', c') \\
& \qquad \mathsf{st}_\mathcal{S} \leftarrow (s + 1, y_i, t, c) \\
& \qquad \text{Return } (\mathsf{st}_\Pi, \mathsf{st}_\mathcal{S})
\end{array}
$$

Fig. 17: Description of the simulator for adaptive corruptions.

Following the assumption on [14], we establish that the adversary must see no messages of uncorrupted participants. As such, until a corruption occurs, the adversary sees either a projection of the protocol state producing an empty state (real world) or an actual empty state (ideal world), and thus

$$\mathsf{st}_\Pi^\epsilon = (\epsilon, \epsilon, [\,], [\,])$$

The adversary can corrupt a participant in any of the $n + 1$ protocol steps. It remains to show that, from that point onwards, the view is indistinguishable at all times.

Observe that (ℓ, p)-crypto-privacy ensures that $\mathcal{S}_{\mathsf{priv}}$ produces an output share, trace and coins indistinguishable from an adversary that observes the full protocol execution. This means that a corruption after the final protocol step (n) produces a trivially indistinguishable view, since $(\mathsf{T}, t, c) \leftarrow \mathsf{Sel}(\Pi, n, t, c)$. For all other corruptions in step $s \in [1..n - 1]$, and given that (y_i, t, c) is an indistinguishable protocol trace for Π, the call to $\mathsf{Sel}(\Pi, s, t, c)$ ensures that the resulting (b, t', c') will provide an indistinguishable protocol execution for protocol step s. Our theorem follows.

Appendix D   Full proof of theorem 3

To prove (ℓ, p)-privacy, we must show that, for all $\bar{x}$ and $1 \leq i \leq n$, the distributions of $(\bar{x}, y, y_i, t_i, c_i, \mathsf{st})$ are identical in the real and ideal worlds. The correct distribution of y follows from low-correctness of π. We will

next establish the correct distributions of t_i and c_i. Low-security states that all shared values that satisfy the leakage relation will result in identical traces. So the simulator simply inverts the leakage function to get some value x' with the correct leakage and computes any sharing $\bar{x}'$. It then executes π on $\bar{x}'$ to get the simulated trace for party i and share y_i. The remaining step that y_i is identically distributed in both worlds follows from the fact that c_i and t_i are identically distributed in both worlds, and uniquely determine y_i.

For clarity in presentation, we present the three main definitions that are necessary for the proof in a game-based form.

Figure 18 presents correctness of local execution, i.e. that y_i is uniquely defined by (x_i, t_i, c_i). Figure 19 describes low-level correctness of Π according to Definition 3, namely the correctness of the output value of Π with respect to its idealized version p. Figure 20 presents low-level security of Π according to Definition 4, namely the distributional equivalence of two runs of Π for inputs of the same leakage.

game $\mathbf{Real}_{\Pi,\mathcal{A}}()$:	**game $\mathbf{Ideal}_{\Pi,\mathcal{A}}()$:**
$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$	$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$
$(\bar{y}, t, c) \twoheadleftarrow \Pi(\bar{x})$	$(\bar{y}, t, c) \twoheadleftarrow \Pi(\bar{x})$
	$y_i' \leftarrow \Pi_i(x_i, t_i, c_i)$
$y \leftarrow \mathrm{Unshare}(\bar{y})$	$y \leftarrow \mathrm{Unshare}(\bar{y})$
Return $\mathcal{A}_2(\bar{x}, y_i, y, t_i, c_i, \mathsf{st})$	Return $\mathcal{A}_2(\bar{x}, y_i', y, t_i, c_i, \mathsf{st})$

Fig. 18: Correctness of local execution.

game $\mathbf{Real}_{\Pi,\mathcal{A}}()$:	**game $\mathbf{Ideal}_{p,\mathcal{A}}()$:**
$(\bar{x}, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$	$(\bar{x}, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$
	$x \leftarrow \mathrm{Unshare}(\bar{x})$
$(\bar{y}, \cdot, \cdot) \twoheadleftarrow \Pi(\bar{x})$	$y \leftarrow p(x)$
$y \leftarrow \mathrm{Unshare}(\bar{y})$	
Return $\mathcal{A}_2(\bar{x}, y, \mathsf{st})$	Return $\mathcal{A}_2(\bar{x}, y, \mathsf{st})$

Fig. 19: Protocol correctness.

game $\mathsf{L}_{\Pi,\mathcal{A}}()$:	**game $\mathsf{R}_{\Pi,\mathcal{A}}()$:**
$(\bar{x}, \bar{x}', i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$	$(\bar{x}, \bar{x}', i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$
$x \leftarrow \mathrm{Unshare}(\bar{x})$	$x \leftarrow \mathrm{Unshare}(\bar{x})$
$x' \leftarrow \mathrm{Unshare}(\bar{x}')$	$x' \leftarrow \mathrm{Unshare}(\bar{x}')$
If $\ell(x) \neq \ell(x')$:	If $\ell(x) \neq \ell(x')$:
$\quad$ Return $(b \twoheadleftarrow \{0, 1\})$	$\quad$ Return $(b \twoheadleftarrow \{0, 1\})$
$(\cdot, t, c) \twoheadleftarrow \Pi(\bar{x})$	$(\cdot, t', c') \twoheadleftarrow \Pi(\bar{x}')$
Return $\mathcal{A}_2(\bar{x}, \bar{x}', t_i, c_i, \mathsf{st})$	Return $\mathcal{A}_2(\bar{x}, \bar{x}', t_i', c_i', \mathsf{st})$

Fig. 20: Probabilistic non-interference.

Proof. Our proof is a sequence of three game hops, represented in Figure 21 and described as follows.

$\mathsf{G0}$ exactly matches the real world in Figure 8. $\mathsf{G1}$ replaces the value of the received y_i by an alternative y_i', calculated from (x_i, t_i, c_i) via Π_i. We upper bound the difference between these two experiments by

constructing an adversary $\mathcal{B}$ against Π_i of Π such that

$$|\Pr[\mathsf{G1}(\,) \Rightarrow \mathsf{T}] - \Pr[\mathsf{G0}(\,) \Rightarrow \mathsf{T}]| = \mathsf{Adv}_{\Pi,\mathcal{B}}^{\Pi_i}$$

Adversary $\mathcal{B}$ executes as follows. $\mathcal{B}_1$ runs $\mathcal{A}_1$ to construct $(\bar{x}, i, \mathsf{st})$ and selects it as input for the experiment of Figure 18. This will produce a tuple $(\bar{x}, y_i, y, t_i, c_i, \mathsf{st})$, exactly matching the one that must be provided to $\mathcal{A}_2$. $\mathcal{B}_2$ returns the result obtained from $\mathcal{A}_2$.

G2 replaces the value of the received y by an alternative y', computed via p. We upper bound the difference between the two experiments by constructing an adversary $\mathcal{C}$ against p-low-correctness of Π so that

$$|\Pr[\mathsf{G2}(\,) \Rightarrow \mathsf{T}] - \Pr[\mathsf{G1}(\,) \Rightarrow \mathsf{T}]| = \mathsf{Adv}_{\Pi,p,\mathcal{C}}^{p\text{-ll-corr}}$$

Adversary $\mathcal{C}$ executes as follows. $\mathcal{C}_1$ runs $\mathcal{A}_1$ to construct $(\bar{x}, i, \mathsf{st})$, selecting $(\bar{x}, \mathsf{st})$ as the input for the low-level correctness experiment of Figure 19. This will produce a tuple $(\bar{x}, y, \mathsf{st})$. $\mathcal{C}_2$ will then run

$$(\cdot, t, c) \twoheadleftarrow \Pi(\bar{x})$$
$$y_i' \leftarrow \Pi_i(x_i, t_i, c_i)$$

to produce the tuple $(\bar{x}, y_i', y', t_i, c_i, m)$ to be given to $\mathcal{A}_2$. $\mathcal{C}_2$ returns the result obtained from $\mathcal{A}_2$.

game $\mathsf{G0}_{\Pi,\mathcal{A}}(\,)$:	**game** $\mathsf{G1}_{\Pi,\mathcal{A}}(\,)$:
$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$	$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$
$(\bar{y}, t, c) \twoheadleftarrow \Pi(\bar{x})$	$(\bar{y}, t, c) \twoheadleftarrow \Pi(\bar{x})$
	$y_i' \leftarrow \Pi_i(x_i, t_i, c_i)$
$y \leftarrow \mathrm{Unshare}(\bar{y})$	$y \leftarrow \mathrm{Unshare}(\bar{y})$
Return $\mathcal{A}_2(\bar{x}, y_i, y, t_i, c_i, \mathsf{st})$	Return $\mathcal{A}_2(\bar{x}, y_i', y, t_i, c_i, \mathsf{st})$
game $\mathsf{G1}_{\Pi,\mathcal{A}}(\,)$:	**game** $\mathsf{G2}_{\Pi,p,\mathcal{A}}(\,)$:
$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$	$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$
$(\bar{y}, t, c) \twoheadleftarrow \Pi(\bar{x})$	$(\bar{y}, t, c) \twoheadleftarrow \Pi(\bar{x})$
$y_i' \leftarrow \Pi_i(x_i, t_i, c_i)$	$y_i' \leftarrow \Pi_i(x_i, t_i, c_i)$
$y \leftarrow \mathrm{Unshare}(\bar{y})$	$x \leftarrow \mathrm{Unshare}(\bar{x})$
	$y' \leftarrow p(x)$
Return $\mathcal{A}_2(\bar{x}, y_i', y, t_i, c_i, \mathsf{st})$	Return $\mathcal{A}_2(\bar{x}, y_i', y', t_i, c_i, \mathsf{st})$
game $\mathsf{G2}_{\Pi,p,\mathcal{A}}(\,)$:	**game** $\mathsf{G3}_{\Pi,p,\mathcal{A}}(\,)$:
$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$	$(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1(\,)$
$x \leftarrow \mathrm{Unshare}(\bar{x})$	$x \leftarrow \mathrm{Unshare}(\bar{x})$
	$x' \twoheadleftarrow \mathsf{Find}(\ell(x))$
	$\bar{x}' \twoheadleftarrow \mathrm{Share}(x')$
$(\bar{y}, t, c) \twoheadleftarrow \Pi(\bar{x})$	$(\bar{y}, t', c') \twoheadleftarrow \Pi(\bar{x}')$
$y_i' \leftarrow \Pi_i(x_i, t_i, c_i)$	$y_i' \leftarrow \Pi_i(x_i, t_i', c_i')$
$y' \leftarrow p(x)$	$y' \leftarrow p(x)$
Return $\mathcal{A}_2(\bar{x}, y_i', y', t_i, c_i, \mathsf{st})$	Return $\mathcal{A}_2(\bar{x}, y_i', y', t_i', c_i', \mathsf{st})$

Fig. 21: Proof hops for Theorem 3.

G3 replaces the values of (t, c) by those produced by an alternate protocol execution, over a set of shares whose leakage is the same as that of the original input. This makes use of $\mathsf{Find}(l)$, which we assume to be an efficient computation of a value x', whose leakage is l. We upper bound the difference between the two experiments by constructing an adversary $\mathcal{D}$ against ℓ-low-security of Π such that

$$|\Pr[\mathsf{G3}(\,) \Rightarrow \mathsf{T}] - \Pr[\mathsf{G2}(\,) \Rightarrow \mathsf{T}]| = \mathsf{Adv}_{\Pi,\ell,\mathcal{D}}^{\ell\text{-ll-sec}}$$

Adversary $\mathcal{D}$ executes as follows. $\mathcal{D}_1$ runs $\mathcal{A}_1$ to construct $(\bar{x}, i, \mathsf{st})$. Then, it recovers x, runs $\mathsf{Find}(\ell(x))$ to obtain an alternative x' with the same leakage (which, by assumption, can be done efficiently), generates shares $\bar{x}'$ and selects $(\bar{x}, \bar{x}', i, x')$ as the input for the low-level security experiment of Figure 20. This will produce a tuple $(\bar{x}, \bar{x}', t_i, c_i, x')$, and $\mathcal{D}_2$ will run

$$y_i' \leftarrow \Pi_i(x_i, t_i', c_i')$$
$$y' \leftarrow p(x')$$

to produce the tuple $(\bar{x}, y_i', y', t_i, c_i, \mathsf{st})$ to be given to $\mathcal{A}_2$. $\mathcal{D}_2$ returns the result obtained from $\mathcal{A}_2$.

To conclude, we have that

$$\mathbf{Adv}_{\Pi, \mathcal{F}, \mathcal{A}, \mathcal{S}}(\,) = \mathsf{Adv}_{\Pi, \mathcal{B}}^{\Pi_i} + \mathsf{Adv}_{\Pi, p, \mathcal{C}}^{p\text{-ll-corr}} + \mathsf{Adv}_{\Pi, \ell, \mathcal{D}}^{\ell\text{-ll-sec}}$$
$$= 0$$

and Theorem 3 follows.