Verified implementations for secure and verifiable computation

José Bacelar Almeida
Manuel Barbosa
INESC TEC and Universidade do Minho
{jba,mbb}@di.uminho.pt

Gilles Barthe
IMDEA Software Institute
gjbarthe@gmail.com

Guillaume Davy
ENS Cachan and
IMDEA Software Institute
davyg@davyg.fr

François Dupressoir
IMDEA Software Institute
fdupress@gmail.com

Benjamin Grégoire
INRIA Sophia-Antipolis
Benjamin.Gregoire@inria.fr

Pierre-Yves Strub
IMDEA Software Institute
pierre-yves@strub.nu

Abstract

Formal verification of the security of software systems is gradually moving from the traditional focus on idealized models, to the more ambitious goal of producing *verified implementations*. This trend is also present in recent work targeting the verification of cryptographic software, but the reach of existing tools has so far been limited to cryptographic primitives, such as RSA-OAEP encryption, or standalone protocols, such as SSH. This paper presents a scalable approach to formally verifying implementations of higher-level cryptographic systems, directly in the computational model.

We consider circuit-based cloud-oriented cryptographic protocols for secure and verifiable computation over encrypted data. Our examples share as central component Yao's celebrated transformation of a boolean circuit into an equivalent "garbled" form that can be evaluated securely in an untrusted environment. We leverage the foundations of garbled circuits set forth by Bellare, Hoang, and Rogaway (CCS 2012, ASIACRYPT 2012) to build *verified implementations* of garbling schemes, a verified implementation of Yao's secure function evaluation protocol, and a verified (albeit partial) implementation of the verifiable computation protocol by Gennaro, Gentry, and Parno (CRYPTO 2010). The implementations are formally verified using EasyCrypt, a tool-assisted framework for building high-confidence cryptographic proofs, and critically rely on two novel features: a module and theory system that supports compositional reasoning, and a code extraction mechanism for generating implementations from formalizations.

Keywords. Garbled Circuits, Secure Function Evaluation, Verifiable Computation, Verified Implementations, EasyCrypt.

1. Introduction

Research on formally verifying the security of software systems was initiated almost 40 years ago, and has established a number of significant landmarks, including security proofs for separation kernels, hypervisors, virtual machines, and web browsers. For their most part, these works have focused on idealized models. However, a recent trend is to build *verified implementations*. Transposing the guarantees from models to implementations offers a much higher degree of assurance, and prevents the possibility of introducing attack vectors via implementation mistakes or abstraction gaps.

The recent security controversy has placed cryptographic software under the spotlight and has raised justified concerns with respect to the potential exploitation of *backdoors* at two different levels: i. those that undermine the theoretical security guarantees offered by protocols included in cryptographic standards; ii. those that thwart the practical security of cryptographic software implementations. One potential way to rebuild trust in cryptographic software is to employ formal methods to rule out the existence of such backdoors, by building and deploying formally verified cryptographic implementations.

One application area where the adoption of this strategy is both particularly pressing *and* challenging is that of cloud computing. Here, the need to reduce the level of trust placed in service providers can only be addressed by deploying a new generation of advanced cryptographic protocols that allow end-users to protect their data, whilst taking advantage of the technical and economic benefits of the cloud paradigm. These protocols enable secure and verifiable computation over encrypted data and can be seen as cryptographic virtual machines supporting arbitrary computations; to highlight this fact, we will refer to them as *cryptographic systems*.

Developing verified implementations of cryptographic systems seems unfathomable because their security hinges on rather sophisticated cryptographic constructions, whose security must itself be verified formally. We identify at least three significant challenges towards achieving this goal, and argue below that these challenges have not yet been met satistactorily by existing tools for verifying the security of cryptographic constructions such as CryptoVerif [1] and earlier versions (v0.2) of EasyCrypt [2]; we discuss F7 [3] at the end of the introduction.

The first challenge is to provide support for a broader scopus of cryptographic proof techniques. EasyCrypt v0.2 and CryptoVerif excel at modelling basic game-playing techniques such as equivalences, failure events, reductions, eager/lazy sampling. However, a number of standard concepts and techniques from provable security, such as hybrid arguments or rewinding, have resisted thus far formalization in these systems. In addition, comparatively little work has been done in formalising computational security proofs for simulation-based notions of security. Indeed, simulation-based proofs are markedly distinct from the reductionistic arguments typically addressed by CryptoVerif or EasyCrypt v0.2; they intrinsically require existential quantification over adversarial algorithms and the ability to instantiate security models with concrete algorithms (the simulators) that serve as witnesses as to the validity of the security claims. Notable exceptions include formal accounts of zero-knowledge protocols [4] and computational differential privacy [5]; however, these simulation-based proofs are developed in simplified settings, as further elaborated in the related work section.

The second challenge is to ensure that the verification tools permit taming the complexity of cryptographic systems. CryptoVerif and EasyCrypt v0.2 have primarily been used to verify crypto-

graphic primitives, such as Full-Domain Hash signatures or OAEP, or standalone protocols, such as Kerberos or SSH. While these examples can be intricate to verify, there is a difference of scale with cryptographic systems, which typically involve several layers of cryptographic constructions. Scalability is a well-known bottleneck for formal verification, that has been addressed with varying degrees of success over the last 40 years. However, CryptoVerif and EasyCrypt v0.2 do not offer any mechanism that specifically addresses this issue, and it is not even clear how (and which) mechanisms used elsewhere could be added in order for formalizations to scale to larger systems.

The third challenge is to enable a new workflow of specification, validation, and prototyping for building verified implementations of cryptographic systems. To ensure that these implementations can have a practical impact, one must develop integrated environments in which cryptographers can specify cryptographic algorithms, perform lightwheight validation (e.g., using a type-checking mechanism), formalize security proofs, and generate reasonably efficient implementations. What we envision here is a formal counterpart to existing prototyping systems for cryptography [6–9]. Although CryptoVerif and EasyCrypt v0.2 have made preliminary steps in this direction, they fall short of providing the functionalities that are expected by practice-oriented cryptographers (see section 7).

These challenges can be summarised as an important open question: can we realistically hope to build practical verified implementations of cryptographic systems in the near future?

CONTRIBUTIONS. In this paper we answer this open question positively, by presenting a new version of EasyCrypt and an associated methodology that enables us to obtain the first verified implementations of cryptographic systems. More specifically, we provide:

1. verified implementations of two variants of Yao's garbled circuits, following the foundational framework put forth by Bellare, Hoang and Rogaway [10, 11], and an n-fold extension of the oblivious transfer protocol by Bellare and Micali [12], in the hashed version presented by Naor and Pinkas [13] (n being the size of the selection string). These are generic components that can be used in many cryptographic systems;
2. a verified implementation of Yao's two-party secure function evaluation (SFE) protocol based on garbled circuits and oblivious transfer. This protocol allows two parties holding private inputs x_1 and x_2, to jointly evaluate any function $f(x_1, x_2)$ and learn its result, whilst being assured that no additional information about their respective inputs is revealed. Two-party SFE provides a general distributed solution to the problem of computing over encrypted data in cloud scenarios;
3. a verified (albeit partial) implementation of a verifiable computation (VC) protocol due to Gennaro, Gentry, and Parno (GGP). This protocol allows a client to delegate the computation of an arbitrary function f to an untrusted worker, while keeping its data secret *and* being able to verify that the result is correct. It is based on Yao's garbled circuits and fully homomorphic encryption (FHE),[1] and it was one of the first to address verifiability in the specific scenario of cloud computing.

In addition to their practical relevance, these examples are appealing because they crystallize many facets of the issues that arise in the formalization of cryptographic systems.

We adopt the latest version[2] (v1.0) of EasyCrypt, which enhances the initial prototype described in [2] in multiple directions. In particular, it provides a lightweight module system that is able to reflect the logical design of cryptographic systems and the modular structure of their proofs. This version also incorporates a new code generation mechanism to produce reasonably efficient functional implementations of cryptographic algorithms from EasyCrypt descriptions. Using the new features of EasyCrypt and our own extensions, we are able to formalize complex reasonings that arise in our examples and were previously out of reach.

Highlights of our formalization include:

- a formally verified library of generic hybrid arguments. The library is based on formalizing security games as parametrisable modules, and critically uses quantification over modules. We use hybrid arguments in the security proofs for oblivious transfer and for garbled circuits. However, the library can be used for other purposes, and will contribute to build a broad scopus of formally verified cryptographic techniques;
- simulation-based proofs where simulators and attackers can be seen as interchangeable adversarial algorithms, which can either be concrete, existentially quantified, or universally quantified. In particular, the generic statements of security for the SFE protocol involve two alternations of quantifiers, which would be challenging to handle even in a hand-written proof.
- layered security proofs in which general compositional theorems can be proven using abstract views of cryptographic primitives, and instantiations of these primitives can be proven secure down to one or more computational assumptions. This type of modular reasoning, which is not necessary when performing reductionistic proofs (even complex ones), is essential to permit tackling security proofs of high-level protocols.

Moreover, using the new code generation mechanism provided by EasyCrypt we are able to derive a verified implementation for the SFE protocol, and a verified (but partial) implementation of the VC protocol. We include experimental results showing the practicality of our verified implementation of Yao's SFE protocol.

COMPARISON WITH $F7$. Bhargavan et al. [3] develop a type-based approach for the verification of cryptographic implementations written in a dialect of ML. Their approach is based on refinement types, which are counterparts to assertions in typed functional languages, and allows proving security in a symbolic setting where cryptographic primitives are idealized. Subsequently, Fournet et al. [16] extend this approach towards proving security in the computational setting, using an intricate combination of type abstraction and refinement types. They then use their system to build verified implementations of TLS [17]. In particular, they have currently verified the TLS handshake protocol.

However, $F7$ does not provide any support for relational nor probabilistic reasonings, both of which are central to cryptographic proofs. As a consequence, large and critical parts of cryptographic proofs cannot be stated (let alone verified) using $F7$ and must instead be carried out using pen and paper. Accordingly, there is a significant gap between $F7$ statements and the provable security guarantees that are commonly sought by cryptographers. We believe that this gap is not only hard to apprehend, but might also be problematic to close. As evidence, their most recent work on TLS uses EasyCrypt (v1.0) to carry out some intricate parts of the proof.

[1] The fact that it relies on FHE explains why we only have a partial implementation: there does not yet exist a practical instantiation for this component. The modularity of our approach implies that a working implementation can be derived if and when a suitable FHE scheme becomes available. We also note that that many exciting developments [14, 15] have taken place in VC since the original GGP paper. However, this protocol uses Yao's garbled circuits in a totally different way than that required by Yao's SFE protocol, which makes it ideal for our demonstration purposes.

[2] http://www.easycrypt.info

IMPLEMENTATION-LEVEL DESCRIPTIONS. It is common practice in cryptographic implementations to separate the randomness sampling operations from the rest of the implementation. There are two reasons for this: on one hand, general-purpose programming languages are not probabilistic, and so randomness must invariably be obtained via some sort of system/library call and handled explicitly; on the other hand, developers (particularly in open-source scenarios) often give the end-users the possibility (and responsibility) to choose their preferred randomness sampling procedure (see, for example, the NaCl cryptographic library[3]).

We take the same approach, describing various cryptographic algorithms as deterministic functional operators, so that we immediately get for free a one-to-one mapping between the EasyCrypt functional definitions and ML code. The randomness generation procedures, which we see as ideal specifications of the required distributions that must be available in practical applications, are described separately as EasyCrypt modules.

2. Two-party protocols and oblivious transfer

We present a variant of a classic oblivious transfer protocol [12, 13] and discuss its security proof. Its small size and relative simplicity make it a good introductory example to EasyCrypt formalization.

TWO-PARTY PROTOCOLS. We first start by generically defining two-party protocols, that generalize both Secure Function Evaluation and Oblivious Transfer, and their security. In EasyCrypt, declarations pertaining to abstract concepts meant to later be refined can be grouped into named theories such as the one shown in Figure 1. Any lemma proved in such a theory is also a lemma of any implementation (or instantiation) where the theory axioms hold.

```
theory Protocol.
 type input1, output1.    type input2, output2.
 op validInputs: input1 → input2 → bool.
 op f: input1 → input2 → output1 * output2.

 type rand1, rand2, conv.
 op prot: input1→ rand1→ input2→ rand2→ conv * output1 * output2.
  ...
end Protocol.
```

Figure 1: Abstract Two-Party Protocol.

Two parties want to compute a *functionality* f on their joint inputs, each obtaining their share of the output. This may be done interactively via a *protocol* prot that may make use of additional randomness (passed in explicitly for each of the parties) and produces, in addition to the result, a *conversation trace* of type conv that describes the messages publicly exchanged by the parties during the protocol execution. In addition, the input space may be restricted by a validity predicate validInputs. This predicate expresses restrictions on the adversary-provided values, typically used to exclude trivial attacks not encompassed by the security definition.

Following the standard approach for secure multi-party computation protocols, security is defined using simulation-based definitions. In this case we capture honest-but-curious (or semi-honest, or passive) adversaries. We consider each party's *view* of the protocol (typically containing its randomness and the list of messages exchanged during a run), and a notion of *leakage* for each party, modelling how much of that party's input may be leaked by the protocol execution (for example, its length). Informally, we say that such a protocol is secure if each party's view can be efficiently simulated using only its inputs, its outputs and the other party's leakage. Formally, we express this security notion using two games (one for each party). We display one of them in Figure 2, in the form of an EasyCrypt *module*. Note that modules are used to model games and experiments, but also schemes, oracles and adversaries.

Module type $\mathcal{Adv}_i^{\mathsf{Prot}}$ ($i \in \{1,2\}$) tells us that an adversary impersonating Party i is defined by two procedures: i. choose that takes no argument and chooses a full input pair for the functionality, and ii. distinguish, that uses Party i's view of the protocol execution to produce a boolean guess as to whether it was produced by the real system or the simulator. Since the module type is not parameterized, the adversary is not given access to any oracles (modelling a non-adaptive adversary). We later show how oracle access can be given to both abstract and concrete modules. Note that procedures in the same module may share state, and it is therefore not necessary to explicitly add state to the module signature. A module that implements all procedures in a given module type is said to *implement* that type. Note that, for example, any module implementing Sim also implements Sim_1. We omit module types for the randomness generators R_1 and R_2, as they only provide a single procedure gen taking some leakage and producing some randomness. We also omit the dual security game for Party 2.

The security game, modelled as module Sec_1, is explicitly parameterized by two randomness-producing modules R_1 and R_2, a simulator S_1 and an adversary $\mathcal{A}_1$. This enables the code of procedures defined in Sec_1 to make queries to any procedure that appears in the module types of its parameters. However, they may not directly access the internal state or procedures that are implemented by concrete instances of the module parameters, when these are hidden by the module type. The game implements, in a single experiment, both the real and ideal worlds. In the real world, the protocol prot is used with adversary-provided inputs to construct the adversary's view of the protocol execution. In the ideal world, the functionality is used to compute Party 1's output, which is then passed along with Party 1's input and Party 2's leakage to the simulator, which produces the adversary's view of the system. We prevent the adversary from trivially winning by denying it any advantage when it chooses invalid inputs.

A two-party protocol prot (parameterized by its randomness-producing modules) is said to be secure with leakage $\Phi = (\phi_1, \phi_2)$ whenever, for any adversary $\mathcal{A}_i$ implementing $\mathcal{Adv}_i^{\mathsf{Prot}}$ ($i \in \{1,2\}$), there exists a simulator S_i implementing Sim_i such that

$$\mathsf{Adv}^{\mathsf{Prot}^i_\Phi}_{\mathsf{prot},\mathsf{S}_i,\mathsf{R}_1,\mathsf{R}_2}(\mathcal{A}_i) = |2 \cdot \Pr[\mathsf{Sec}_i(\mathsf{R}_1,\mathsf{R}_2,\mathsf{S}_i,\mathcal{A}_i) : \mathsf{res}] - 1|$$

is small, where res denotes the Boolean output of procedure main.

Intuitively, the existence of such a simulator S_i implies that the protocol conversation and output cannot reveal any more information than the information revealed by the simulator's input.

Throughout the paper we omit the indices representing randomness generators whenever they are clear from the context.

OBLIVIOUS TRANSFER PROTOCOLS. We can now define oblivious transfer, restricting our attention to a specific notion useful for constructing general SFE functionalities. To do so, we *clone* the Protocol theory, which makes a literal copy of it and allows us to instantiate its abstract declarations with concrete definitions. When cloning a theory, everything it declares or defines is part of the clone, including axioms and lemmas. Note that lemmas proved in the original theory are also lemmas in the clone. The partial instantiation is shown in Figure 3. We restrict the input, output and leakage types for the parties, as well as the leakage functions and the functionality f. The chooser (Party 1) takes as input a list of Boolean values (i.e., a bit-string) she needs to encode, and the sender (Party 2), takes as input a list of pairs of messages (which can also be seen as alternative encodings for the Boolean values in Party 1's inputs). Together, they compute the array encoding the chooser's input, revealing only the lengths of each other's inputs. We declare an abstract constant n that bounds the size of the chooser's input.

[3] http://nacl.cr.yp.to

```
type leak_1, leak_2.   op φ_1 : input_1 → leak_1.   op φ_2 : input_2 → leak_2.
type view_1 = rand_1 * conv.   type view_2 = rand_2 * conv.

module type Sim = {
  proc sim_1(i_1: input_1, o_1: output_1, l_2: leak_2) : view_1
  proc sim_2(i_2: input_2, o_2: output_2, l_1: leak_1) : view_2
}.

module type Sim_i = {
  proc sim_i(i_i: input_i, o_i: output_i, l_{3-i}: leak_{3-i}) : view_i
}.

module type Adv_i^Prot = {
  proc choose(): input_1 * input_2
  proc distinguish(v: view_i) : bool
}.

module Sec_1(R_1: Rand_1, R_2: Rand_2, S: Sim_1, A_1: Adv_1^Prot) = {
  proc main() : bool = {
    var real, adv, view_1, o_1, r_1, r_2, i_1, i_2;
    (i_1,i_2) = A_1.choose();
    real <$- {0,1};
    if (!validInputs i_1 i_2)
      adv <$- {0,1};
    else {
      if (real) {
        r_1 = R_1.gen(φ_1 i_1);
        r_2 = R_2.gen(φ_2 i_2);
        (conv,_) = prot i_1 r_1 i_2 r_2;
        view_1 = (r_1, conv);
      } else {
        (o1,_) = f i_1 i_2;
        view_1 = S.sim_1(i_1, o_1, φ_2 i_2);
      }
      adv = A_1.distinguish(view_1);
    }
    return (adv = real);
  }
}.
```

Figure 2: Security of a two-party protocol protocol.

```
clone Protocol as OT with
  type input_1 = bool array,
  type output_1 = msg array,
  type leak_1 = int,
  type input_2 = (msg * msg) array,
  type output_2 = unit,
  type leak_2 = int,
  op φ_1 (i_1: bool array) = length i_1,
  op φ_2 (i_2: (msg * msg) array) = length i_2,
  op f (i_1: bool array) (i_2: (msg * msg) array) = i_1 i_2.
  op validInputs(i_1: bool array) (i_2: (msg * msg) array) =
      0 < length i_1 ≤ n_max ∧ length i_1 = length i_2,
...
```

Figure 3: Instantiating Two-Party Protocols into Abstract SFE.

This introduces an implicit quantification on the bound n in all results we prove. Defining OT security is then simply a matter of instantiating the general notion of security for two-party protocols via cloning. Looking ahead, we use $\mathsf{Adv}^{\mathsf{OT}^i}$ to denote the resulting instance of $\mathsf{Adv}^{\mathsf{Prot}^i_{(\mathsf{length},\mathsf{length})}}$, and similarly, we write $\mathcal{Adv}_i^{\mathsf{OT}}$ the types for adversaries against the OT instantiation.

AN OBLIVIOUS TRANSFER PROTOCOL. To define a concrete two-party OT protocol, one now only has to define the types of randomness and conversations and the protocol itself, with its individual computation and message exchange steps.

In the following, we abuse notation and denote operators, their lifting to arrays of the same length, and their lifting to one array argument and one scalar argument in the same way, writing $\overline{\mathsf{v}}$ to single out array variables. For an array of pairs $\overline{\mathsf{a}}$, we write $\overline{\mathsf{a}}_0$ for the array of its first components (i.e. $\mathsf{fst}\ \overline{\mathsf{a}}$), and $\overline{\mathsf{a}}_1$ for the array of its second components. We also denote multiplicatively operations in a group $\mathcal{G}$ of prime order q, using a given generator g.

In Figure 4, we describe our OT protocol in a purely functional manner, making any local state shared between the various stages of a given party explicit. For example, step_1 outputs the sender's local state st_s, for later use by step_3.

```
op step_1 (m:(msg * msg) array) (r:int array * G) =
  let (c,hkey) = r in
  let st_s = (m,g^c,hkey) in
  let m_1 = (hkey,g^c) in
  (st_s,m_1).

op step_2 (b:bool array) (r:G array) m_1 =
  let (hkey,gc) = m_1 in
  let st_c = (b,hkey,r) in
  let m_2 = if b then gc / g^r else g^r in
  (st_c,m_2).

op step3 st_s (r:G) m_2 =
  let (m,gc,hkey) = st_s in
  let e = (H(hkey,m_2^r) ⊕ m_0,H(hkey,(gc/m_2)^r) ⊕ m_1) in
  let m_3 = (g^r,e) in
  m_3.

op finalize st_c m_3 =
  let (b,hkey,x) = st_c in
  let (gr,e) = m_3 in
  let res = H(hkey,gr^x) ⊕ e_b in
  res.

clone OTProt as SomeOT with
  type rand_1 = G array,
  type rand_2 = (G array * G) * G,
  op prot (b:input_1) (r_c:rand_1) (m:input_2) (r_s:rand_2) =
    let (st_s,m_1) = step1 m (fst r_s) in
    let (st_c,m_2) = step2 b r_c m_1 in
    let m_3 = step3 st_s (snd r_s) m_2 in
    let res = finalize st_c m_3 in
    let conv = (m_1,m_2,m_3) in
    (conv,(res,())).
```

Figure 4: Our Concrete Oblivious Transfer Protocol.

We prove this protocol secure in the standard model via a reduction to the decisional Diffie-Hellman assumption and an entropy-smoothing assumption on the hash function. We let $\mathsf{Adv}^{\mathsf{DDH}}(\mathcal{A})$ and $\mathsf{Adv}^{\mathsf{ES}}(\mathcal{A})$ be the advantage of an adversary $\mathcal{A}$ breaking the DDH and the Entropy Smoothing assumptions, respectively.

Theorem 1 (OT-security of SomeOT)**.** *For all $i \in \{1,2\}$ and OT^i adversary $\mathcal{A}_i$ of type $\mathcal{Adv}_i^{\mathsf{OT}}$ against the SomeOT protocol, we can construct two efficient adversaries $\mathcal{D}^{\mathsf{DDH}}$ and $\mathcal{D}^{\mathsf{ES}}$, and a efficient simulator S such that*

$$\mathsf{Adv}^{\mathsf{OT}^i}_{\mathsf{SomeOT},\mathsf{S}}(\mathcal{A}_i) \leq \mathsf{n} \cdot \mathsf{Adv}^{\mathsf{DDH}}(\mathcal{D}^{\mathsf{DDH}}) + \mathsf{n} \cdot \mathsf{Adv}^{\mathsf{ES}}(\mathcal{D}^{\mathsf{ES}}).$$

Proof (Sketch). We first reduce to n-ary variants of the assumptions, constructing adversaries $\mathcal{D}^{\mathsf{DDHn}}$ and $\mathcal{D}^{\mathsf{ESn}}$. Security against malicious senders ($i = 1$) is information theoretic, since the sender's view of a protocol execution is statistically indistinguishable from a random view. To prove security against malicious choosers ($i = 2$), we consider a simulator S that replaces the chosen components of the ciphertexts e_0 and e_1 (in step_2) by random bitstrings. The proof that the real world is computationally close to the ideal world follows the following strategy: $\mathcal{D}^{\mathsf{DDHn}}$ is used to justify the replacement of the public keys corresponding to chosen messages by random group elements, and $\mathcal{D}^{\mathsf{ESn}}$ to replace the hash

of random group elements by random bitstrings. We further reduce both n-ary variants to the corresponding standard assumption using a single generic lemma described below. □

USING GENERIC LEMMAS. In the proof of Theorem 1, both reductions first go to n-ary versions of the DDH and Entropy-Smoothing hypotheses before reducing these further to standard assumptions. Both of these two "n-ary to unary" reductions are in fact proved by simply applying a generic lemma, which we formalize independently of these particular applications as part of EasyCrypt's library of verified transformations. The objective of this library is to formalize often-used proof techniques once and for all, enabling the user to perform proofs "by a hybrid argument", or "by eager sampling", whilst formally checking that all side conditions are fulfilled at the time the lemma is applied. We now describe the generic hybrid argument used in the proof of Theorem 1 and others.

```
type input, output, inleaks, outleaks.

module type Orcl = { proc o(_:input) : output }.

module type Orcl_b = {
  proc leaks(_:inleaks): outleaks
  proc o_L(_:input) : output
  proc o_R(_:input) : output
}.

module type Adv^Hy (O_b:Orcl_b, O:Orcl) = { proc main () : bool }.

module L_n (O_b:Orcl_b, A:Adv^Hy) = {
  module O: Orcl = { ... } (* increment C.c and call O_b.o_L *)
  module A' = A(O_b, O);
  proc main () : bool = { C.c = 0; return A'.main(); }
}.
module R_n (O_b:Orcl_b, A:Adv) = { ... (* Same as L_n but use O_b.o_R *) }.

op q : int.

module B(A:Adv^Hy, O_b:Orclb, O:Orcl) = {
  module LR = {
    var l, l_0 : int
    proc orcl(m:input):output = {
      var r : output;
      if (l_0 < l) r = O_b.o_L(m);
      else if (l_0 = l) r = O.orcl(m);
      else r = O_b.o_R(m);
      l = l + 1; return r;
    }
  }
  module A' = A(O_b,LR)
  proc main():outputA = {
    var r:outputA;
    LRB.l_0 <$- [0..q-1]; LRB.l = 0; return A'.main();
  }
}.

lemma Hybrid: ∀ (O_b:Orcl_b{C,B}) (A:Adv^Hy {C,B,O_b}),
  Pr[Ln(O_b,A): res∧ C.c ≤ n] − Pr[Rn(O_b,A): res∧ C.c ≤ n]
  = q ∗ (Pr[Ln(O_b,B(A)): res∧ B.l ≤ n ∧ C.c ≤ 1]
       − Pr[Rn(O_b,B(A)): res∧ B.l ≤ n ∧ C.c ≤ 1]).
```

Figure 5: Abstract Definitions for Hybrid Argument.

As described in Figure 5, consider an adversary parametrized by two modules. The first parameter O_b, implementing the module type Orcl_b, provides a leakage oracle, a left oracle o_L and right o_R. The second parameter O, implementing module type Orcl, provides a single oracle o. The goal of an adversary implementing type $\mathcal{Adv}^{\mathsf{Hy}}$ is to guess in at most q queries to O.o if it is the left oracle $\mathsf{O}_b.\mathsf{o}_L$ or the right oracle $\mathsf{O}_b.\mathsf{o}_R$. To express the advantage of such an adversary, we write two modules: the first one, L_n, defines a game where the adversary is called with O.o equal to $\mathsf{O}_b.\mathsf{o}_L$, the second one, R_n, uses $\mathsf{O}_b.\mathsf{o}_R$ instead. Both L_n and R_n use a variable C.c to count the number of queries made to their oracle by the adversary. We define the advantage of an adversary $\mathcal{A}$ in distinguishing $\mathsf{O}_b.\mathsf{o}_L$ from $\mathsf{O}_b.\mathsf{o}_R$ as the difference of the probability of games $\mathsf{L}_n(\mathsf{O}_b,\mathcal{A})$ and $\mathsf{R}_n(\mathsf{O}_b,\mathcal{A})$ returning 0. Given any distinguishing adversary $\mathcal{A}$, we construct a distinguishing adversary $\mathcal{B}$ that may use $\mathcal{A}$ but always makes at most one query to oracle O.o.

The Hybrid lemma relates the advantages of any adversary $\mathcal{A}$ with the advantage of its constructed adversary $\mathcal{B}$ when $\mathcal{A}$ is known to make at most q queries to O.o. Note that the validity of the Hybrid lemma is restricted to adversaries that do not have a direct access to the counter C.c, or to the memories of B and O_b, this is denoted by the notation $\mathcal{Adv}^{\mathsf{Hy}}\{\mathsf{C},\mathsf{B},\mathsf{O}_b\}$ in the EasyCrypt code. Other lemmas shown in this paper also have such restrictions in their formalizations, but they are as expected (that is, they simply enforce a strict separation of the various protocols', simulators' and adversaries' memory spaces) and we omit them for clarity. The construction of $\mathcal{B}$ is generic in the underlying adversary $\mathcal{A}$, which can remain completely abstract. We underline that, for all $\mathcal{A}$ implementing module type $\mathcal{Adv}^{\mathsf{Hy}}$, the partially-applied module $\mathcal{B}(\mathcal{A})$ implements $\mathcal{Adv}^{\mathsf{Hy}}$ as well and can therefore be plugged in anywhere a module of type $\mathcal{Adv}^{\mathsf{Hy}}$ is expected. This ability to generically construct over abstract schemes or adversaries is central to handling modularity in EasyCrypt.

Finally, we observe that the Hybrid lemma applies even to an adversary that may place queries to the individual $\mathsf{O}_b.\mathsf{o}_L$ and $\mathsf{O}_b.\mathsf{o}_R$ oracles. It is of course applicable (and is in fact often applied) to adversaries that do not place such queries.

3. Garbling schemes

Garbling schemes [10] (Figure 6) are operators on *functionalities* of type func. Such functionalities can be evaluated on some input using an eval operator. In addition, a functionality can be *garbled* using three operators (all of which may consume randomness). funG produces the garbled functionality, inputK produces an input-encoding key, and outputK produces an output-encoding key. The garbled evaluation evalG takes a garbled functionality and some encoded input and produces the corresponding encoded output. The input-encoding and output-decoding functions are self-explanatory. In practice, we are interested in garbling functionalities encoded as

```
type func, input, output.
op eval : func → input → output.
op valid: func → input → bool.

type rand, funcG, inputK, outputK.
op funcG : func → rand → funcG.
op inputK : func → rand → inputK.
op outputK: func → rand → outputK.

type inputG, outputG.
op evalG : funcG → inputG → outputG.
op encode: inputK → input → inputG.
op decode: outputK → outputG → output.
```

Figure 6: Abstract Garbling Scheme.

Boolean circuits and therefore fix the func and input types and the eval function. Circuits themselves are represented by their topology and their gates. A topology is a tuple $(n, m, q, \mathbb{A}, \mathbb{B})$, where n is the number of input wires, m is the number of output wires, q is the number of gates, and $\mathbb{A}$ and $\mathbb{B}$ map to each gate its first and second input wire respectively. A circuit's gates are modelled as a map $\mathbb{G}$ associating output values to a triple containing a gate number and the values of the input wires. Gates are modelled polymorphically, allowing us to use the same notion of circuit for Boolean circuits and their garbled counterparts. We only consider

projective schemes [10], where Boolean values on each wire are encoded using a fixed-length random *token*. This fixes the type funcG of garbling schemes, and the outputK and decode operators.

Following the Garble1 construction of Bellare et al. [10], we construct our garbling scheme using a variant of Yao's garbled circuits based on a pseudo-random permutation, via an intermediate Dual-Key Cipher (DKC) construction. We denote the DKC encryption with E, and DKC decryption with D. Both take four tokens as argument: a tweak that we generate with an injective function and use as unique IV, two keys, and a plaintext (or ciphertext). Some intuition about the construction is given in Appendix A.1. We give functional specifications to the garbling algorithms in Figure 7. For clarity, we denote functional folds using stateful for loops.

```
type topo = int * int * int array * int array.
type α circuit = topo * (int * α * α ,α ) map.

type leak = topo.

type input, output = bool array.
type func = bool circuit.

type funcG = token circuit.
type inputG, outputG = token array.
op evalG f i =
  let ((n,m,q,𝔸,𝔹),𝔾) = f in
  let evalGate = λ g x1 x2,
    let x1,0 = lsb x1 and x2,0 = lsb x2 in
    D (tweak g x1,0 x2,0) x1 x2 𝔾[g,x1,0,x2,0] in
  let wires = extend i q in (* extend the array with q zeroes *)
  let wires = map (λ g, evalGate g 𝔸[g] 𝔹[g]) wires in (* decrypt wires *)
  sub wires (n + q − m) m.

type rand, inputK = ((int * bool),token) map.
op encode iK x = init (length x) (λ k, iK[k,x[k]]).

op inputK (f:func) (r:((int * bool),token) map) =
  let ((n,_,_,_,_),_) = f in filter (λ x y, 0 ≤fst x < n) r.

op funcG (f:func) (r:rand) =
  let ((n,m,q,𝔸,𝔹),𝔾) = f in
  for (g,xa,xb) ∈ [0..q] * bool * bool
    let a = 𝔸[g] and b = 𝔹[g] in
    let ta = r[a,xa] and tb = r[b,xb] in
    𝔾̃[g,ta,tb] = E (tweak g ta tb) ta tb r[g,𝔾[g,xa,xb]]
  ((n,m,q,𝔸,𝔹),𝔾̃).
```

Figure 7: SomeGarble: our Concrete Garbling Scheme.

SECURITY OF GARBLING SCHEMES. The privacy property of garbling schemes required by Yao's SFE protocol is more conveniently captured using a simulation-based definition. Like the security notions for protocols, the privacy definition for garbling schemes is parameterized by a leakage function upper-bounding the information about the functionality that may be leaked to the adversary. (We consider only schemes that leak at most the topology of the circuit.) Consider efficient non-adaptive adversaries that provide two procedures: i. choose takes no input and outputs a pair (f,x) composed of a functionality and some input to that functionality; ii. on input a garbled circuit and garbled input pair (F,X), distinguish outputs a bit b representing the adversary's guess as to whether he is interacting with the real or ideal functionality. Formally, we define the SIM-CPA$_\Phi$ advantage of an adversary $\mathcal{A}$ of type $\mathcal{Adv}^{\mathsf{Gb}}$ against garbling scheme Gb = (funcG,inputK,outputK) and simulator S as

$$\mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_\Phi}_{\mathsf{Gb},\mathsf{R},\mathsf{S}}(\mathcal{A}) = |2 \cdot \Pr[\mathsf{SIM}(\mathsf{R},\mathsf{S},\mathcal{A}) : \mathsf{res}] - 1| \,.$$

A garbling scheme Gb using randomness generator R is SIM-CPA$_\Phi$-secure if, for all adversary $\mathcal{A}$ of type $\mathcal{Adv}^{\mathsf{Gb}}$, there exists an efficient simulator S of type Sim such that $\mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_\Phi}_{\mathsf{Gb},\mathsf{R},\mathsf{S}}(\mathcal{A})$ is small.

```
type leak.
op Φ: func → leak.

module type Sim = {
  fun sim(x: output, l: leak): funcG * inputG
}.

module type 𝒜dv^Gb = {
  fun choose(): func * input
  fun distinguish(F: funcG, X: inputG) : bool
}.

module SIM(R: Rand, S: Sim, 𝒜: 𝒜dv^Gb) = {
  fun main() : bool = {
    var real, adv, f, x, F, X;
    (f,x) = 𝒜.gen_query();
    real ←$ {0,1};
    if (!valid f x)
      adv ←$ {0,1};
    else {
      if (real) {
        r = R.gen(Φ f);
        F = funcG f r;
        X = encode (inputK f r) x;
      } else {
        (F,X) = S.sim(f(x),Φ f);
      }
      adv = 𝒜.dist(F,X);
    }
    return (adv = real);
  }
}.
```

Figure 8: Security of garbling schemes.

Following [10], we establish simulation-based security via a general result that leverages a more convenient indistinguishability-based security notion denoted IND-CPA$_{\Phi_{\mathsf{topo}}}$: we formalize a general theorem stating that, under certain restrictions on the leakage function Φ, IND-CPA$_\Phi$-security implies SIM-CPA$_\Phi$ security. This result is discussed below as Lemma 1.

A MODULAR PROOF. The general lemma stating that IND-CPA-security implies SIM-CPA-security is easily proved in a very abstract model, and is then as easily instantiated to our concrete garbling setting. We describe the abstract setting to illustrate the proof methodology enabled by EasyCrypt modules on this easy example.

```
module type 𝒜dv^IND = {
  fun choose(): ptxt * ptxt
  fun distinguish(c:ctxt): bool
}.

module IND (R:Rand, 𝒜:𝒜dv^IND) = {
  fun main(): bool = {
    var p0, p1, p, c, b, b', ret, r;
    (p0,p1) = 𝒜.choose();
    if (valid p0 ∧ valid p1 ∧ Φ p0 = Φ p1) {
      b ←$ {0,1};
      p = if b then p1 else p0;
      r = R.gen(|p|);
      c = enc p r;
      b' = 𝒜.distinguish(c);
      ret = (b = adv);
    }
    else ret ←$ {0,1};
    return ret;
  }
}.
```

Figure 9: Indistinguishability-based Security for Garbling Schemes.

The module shown in Figure 9 is a slight generalization of the standard IND-CPA security notions for symmetric encryption, where some abstract leakage operator Φ replaces the more usual check that the two adversary-provided plaintexts have the same length.We formally prove an abstract result that is applicable to any circumstances where indistinguishability-based and simulation-based notions of security interact. We define the IND-CPA advantage of an adversary $\mathcal{A}$ of type $\mathcal{Adv}^{\mathsf{IND}}$ against the encryption operator enc using randomness generator R with leakage Φ as

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}_\Phi}_{\mathsf{enc},\mathsf{R}}(\mathcal{A}) = |2 \cdot \Pr[\mathsf{Game\_IND}(\mathsf{R},\mathcal{A}).\mathsf{res}] - 1|$$

where R is the randomness generator used in the concrete theory.

In the rest of this subsection, we use the following notion of invertibility. A leakage function Φ on plaintexts (when we instantiate this notion on garbling schemes these plaintexts are circuits and their inputs) is *efficiently invertible* if there exists an efficient algorithm that, given the leakage corresponding to a given plaintext, can find a plaintext consistent with that leakage.

Lemma 1 (IND-CPA-security implies SIM-CPA-security)**.** *If Φ is efficiently invertible, then for every efficient* SIM-CPA *adversary $\mathcal{A}$ of type $\mathcal{Adv}^{\mathsf{Gb}}$, one can build an efficient* IND-CPA *adversary $\mathcal{B}$ and an efficient simulator* S *such that*

$$\mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_\Phi}_{\mathsf{enc},\mathsf{S}}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{IND\text{-}CPA}_\Phi}_{\mathsf{enc}}(\mathcal{B}).$$

Proof (Sketch). Using the inverter for Φ, $\mathcal{B}$ computes a second plaintext from the leakage of the one provided by $\mathcal{A}$ and uses this as the second part of her query in the IND-CPA game. Similarly, simulator S generates a simulated view by taking the leakage it receives and computing a plaintext consistent with it using the Φ-inverter. The proof consists in establishing that $\mathcal{A}$ is called by $\mathcal{B}$ in a way that coincides with the SIM-CPA experiment when S is used in the ideal world, and is performed by code motion. □

FINISHING THE PROOF. We reduce the $\mathsf{IND\text{-}CPA}_{\Phi_{\mathsf{topo}}}$-security of SomeGarble to the DKC-security of the underlying DKC primitive (see [10]). In the lemma statement, c is an abstract upper bound on the size of circuits (in number of gates) that are considered valid. The lemma holds for all values of c that can be encoded in a token minus two bits.

Lemma 2 (SomeGarble is $\mathsf{IND\text{-}CPA}_{\Phi_{\mathsf{topo}}}$-secure)**.** *For every efficient* IND-CPA *adversary $\mathcal{A}$ of type $\mathcal{Adv}^{\mathsf{Gb-IND}}$, we can construct a efficient* DKC *adversary $\mathcal{B}$ such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}_{\Phi_{\mathsf{topo}}}}_{\mathsf{SomeGarble}}(\mathcal{A}) \le (\mathsf{c}+1) \cdot \mathsf{Adv}^{\mathsf{DKC}}_{\mathsf{SomeGarble}}(\mathcal{B}).$$

Proof (Sketch). The constructed adversary $\mathcal{B}$, to simulate the garbling scheme's oracle, samples a wire ℓ_0 which is used as pivot in a hybrid construction where: i. all tokens that are revealed by the garbled evaluation on the adversary-chosen inputs are garbled normally, using the real DKC scheme; otherwise ii. all tokens for wires less than ℓ_0 are garbled using encryptions of random tokens (instead of the real tokens representing the gates' outputs); iii. tokens for wire ℓ_0 uses the real-or-random DKC oracle; and iv. all tokens for wires greater than ℓ_0 are garbled normally.

Here again, the generic hybrid argument (Figure 5) can be instantiated and applied without having to be proved again, yielding a reduction to an adaptive DKC adversary. A further reduction allows us to then build a non-adaptive DKC adversary, since all DKC queries made by $\mathcal{B}$ are in fact random and independent. □

From Lemmas 1 and 2, we can conclude with a security theorem for our garbling scheme.

Theorem 2 (SomeGarble is $\mathsf{SIM\text{-}CPA}_{\Phi_{\mathsf{topo}}}$-secure)**.** *For every* SIM-CPA *adversary $\mathcal{A}$ that implements $\mathcal{Adv}^{\mathsf{Gb}}$, one can construct an efficient simulator* S *and a DKC adversary $\mathcal{B}$ such that*

$$\mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_{\Phi_{\mathsf{topo}}}}_{\mathsf{SomeGarble},\mathsf{S}}(\mathcal{A}) \le (\mathsf{c}+1) \cdot \mathsf{Adv}^{\mathsf{DKC}}_{\mathsf{SomeGarble}}(\mathcal{B}).$$

Proof (Sketch). Lemma 1 allows us to construct from $\mathcal{A}$ the simulator S and an IND-CPA adversary $\mathcal{C}$. From $\mathcal{C}$, Lemma 2 allows us to construct $\mathcal{B}$ and conclude. □

4. Constructing a Secure SFE Protocol

We now explain how garbled circuits and oblivious transfer can be combined to provide a general secure function evaluation protocol, and formalize a generic security proof, parameterized by a secure garbling scheme and a secure oblivious transfer protocol. We then instantiate the abstract argument with our concrete oblivious transfer protocol (Section 2) and our concrete garbling scheme (Section 3) and conclude.

YAO'S SFE CONSTRUCTION. As discussed briefly in Section 2, we model SFE as a two-party protocol. We consider the functionality to be evaluated, encoded as a circuit, as part of Party 2's input and set up the leakage function to let it become public. We denote with $\mathsf{Adv}^{\mathsf{SFE}^i}$ and $\mathcal{Adv}^{\mathsf{SFE}}_i$ the instantiations of $\mathsf{Adv}^{\mathsf{Prot}^i}$ and $\mathcal{Adv}^{\mathsf{Prot}}_i$ to the SFE construction. We preface the definition of our generic SFE construction with two named clones of the abstract garbling scheme and oblivious transfer theories. This allows us to essentially parameterize the SFE protocol with a garbling scheme and an oblivious transfer protocol. Instantiating these parameters is simply done by instantiating the Gb and OT theories with concrete definitions and proofs. In Figure 10, we formalize a standard SFE construction detailed informally in Appendix A.1.

```
clone Garble as Gb.
clone OT as OT.

clone Protocol as SFE with
  type rand1 = OT.rand1,
  type input1 = bool array,
  type output1 = Gb.output,
  type leak1 = int,
  type rand2 = OT.rand2 * Gb.rand,
  type input2 = Gb.func * bool array,
  type output2 = unit,
  type leak2 = Gb.func * int,
  op f i1 i2 = let (c,i2) = i2 in Gb.eval c (i1 || i2),(),
  type conv = (Gb.funcG * token array * Gb.outputK) * OT.conv,
  op validInputs (i1:input1) (i2:input2) =
    0 < length i1 ∧ Gb.validInputs (fst i2) (i1 || snd i2),
  op prot (i1:input1) (r1:rand1) (i2:input2) (r2:rand2) =
    let (c,i2) = i2 in
    let fG = Gb.funG c (snd r2) in
    let oK = Gb.outputK c (snd r2) in
    let iK = Gb.inputK c (snd r2) in
    let iK1 = (take (length i1) iK) in
    let (ot_conv, (t1,_)) = OT.prot i1 r1 iK1 (fst r2) in
    let GI2 = Gb.encode (drop (length i1) iK) i2 in
    (((fG,GI2,oK),ot_conv), (Gb.decode oK (Gb.evalG fG (t1 || GI2)),())).
```

Figure 10: Abstract SFE Construction.

Theorem 3 (Abstract SFE security)**.** *For any oblivious transfer protocol OT and any garbling scheme Gb, let SFE_a be the SFE protocol built using Yao's construction from OT and Gb. For all randomness generators R^G, R^O_1 and R^O_2, we can construct SFE randomness generators R^{SFE}_1 and R^{SFE}_2 such that, for all SFE adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ implementing type $\mathcal{Adv}^{\mathsf{SFE}} = (\mathcal{Adv}^{\mathsf{SFE}}_1, \mathcal{Adv}^{\mathsf{SFE}}_2)$, OT simulator S^O and garbling simulator S^G, we can construct efficient adversaries $\mathcal{A}^O = (\mathcal{A}^O_1, \mathcal{A}^O_2)$ of type*

$\mathcal{A}dv^{\mathsf{OT}} = (\mathcal{A}dv_1^{\mathsf{OT}}, \mathcal{A}dv_2^{\mathsf{SFE}})$ and $\mathcal{A}^G$ of type $\mathcal{A}dv^{\mathsf{Gb}}$ and an efficient simulator S, such that the following inequalities hold.

$$\mathsf{Adv}^{\mathsf{SFE}^1_\Phi}_{\mathsf{SFE}_a,\mathsf{S}}(\mathcal{A}_1) \le \mathsf{Adv}^{\mathsf{OT}^1_\Phi}_{OT,S^O}(\mathcal{A}_1^O) + \mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_\Phi}_{Gb,S^G}(\mathcal{A}^G)$$

$$\mathsf{Adv}^{\mathsf{SFE}^2_\Phi}_{\mathsf{SFE}_a,\mathsf{S}}(\mathcal{A}_2) \le \mathsf{Adv}^{\mathsf{OT}^2_\Phi}_{OT,S^O}(\mathcal{A}_2^O)$$

Proof (Sketch). Since no additional randomness is used by the SFE protocol, the SFE randomness generators are trivially constructed from the randomness generators for OT and garbling. The proof then follows the argument sketched in [10]. From the adversary $\mathcal{A}$, we can easily construct $\mathcal{A}^O$ and $\mathcal{A}^G$ by moving some of the code from the SFE protocol into the adversary. The security assumptions on the oblivious transfer protocol and garbling scheme can then be used to build two simulators S^O and S^G. In turn, these can be combined into a valid simulator S for the full SFE protocol. □

A CONCRETE SFE PROTOCOL. Finally, we instantiate OT with the concrete oblivious transfer protocol we proved secure in Section 2 and Gb with the concrete garbling scheme we proved secure in Section 3 in the construction. The security proof for this concrete construction immediately follows from Theorems 3, 1 and 2. However, we take this opportunity to implement some instantiation-specific optimizations across abstraction boundaries and translate high-level programming constructs like maps and higher-order functions into more efficient data structures such as arrays. We also instantiate the constants n and c as described further in Section 6 and discharge any axioms affecting them. A separate proof that our efficient implementation is perfectly equivalent to the one on which we performed the security proof yields the final security theorem.

Theorem 4 (Security of the concrete SFE protocol)**.** *For all* SFE *adversary $\mathcal{A}$ against the Concrete SFE protocol, we construct an efficient simulator S and efficient adversaries $\mathcal{B}_{\mathsf{DKC}}$, $\mathcal{B}_{\mathsf{DDH}}$ and $\mathcal{B}_{\mathsf{ES}}$, such that the following inequalities hold:*

$$\mathsf{Adv}^{\mathsf{SFE}^1_{\Phi_{\mathsf{topo}}}}_{Concrete,\mathsf{S}}(\mathcal{A}) \le (\mathsf{c}+1)\cdot \mathsf{Adv}^{\mathsf{DKC}}(\mathcal{B}_{\mathsf{DKC}}) + \varepsilon,$$

$$\mathsf{Adv}^{\mathsf{SFE}^2_{\Phi_{\mathsf{topo}}}}_{Concrete,\mathsf{S}}(\mathcal{A}) \le \varepsilon,$$

where $\varepsilon = \mathsf{n}\cdot \mathsf{Adv}^{\mathsf{DDH}}(\mathcal{B}_{\mathsf{DDH}}) + \mathsf{n}\cdot \mathsf{Adv}^{\mathsf{ES}}(\mathcal{B}_{\mathsf{ES}})$.

Proof (Sketch). By functional equivalence of the Concrete protocol and SFE(SomeOT,SomeGarble) (called SFE_c below), we prove $\mathsf{Adv}^{\mathsf{SFE}^1_{\Phi_{\mathsf{topo}}}}_{\mathsf{Concrete},\mathsf{S}}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{SFE}^1_{\Phi_{\mathsf{topo}}}}_{\mathsf{SFE}_c,\mathsf{S}}(\mathcal{A})$ and it suffices to bound the latter. This can be done by applying Theorem 3 and using the adversaries constructed in Theorems 1 and 2 along with the corresponding bounds to conclude. □

5. Verifiable Computation

We now move to our second example of a verifiable computation protocol, in order to further demonstrate the wide applicability of our techniques. In Appendix A, the interested reader will find a description of the functionality and security properties required of a verifiable computation protocol. Here we give only the highlights of how we formalize the seminal construction proposed by Gennaro, Gentry and Parno (GGP) [18].

For the GGP protocol we require a garbling scheme that provides authenticity guarantees, and so we need a slightly different construction from that presented in the previous sections. Intuitively, authenticity imposes that, given one opening of the garbled circuit, it is infeasible to find a valid encoding of an incorrect output. For Yao's construction to display this property, it is necessary that the decoding key includes the labels for the output wires, so that garbled outputs can be checked for consistency. In our formalisation, this consistency check is handled by a new algorithm called valid_outG. Formalising this new garbling scheme requires little effort, as one can simply (re-)define the operators shown in Figure 11 and inherit all the original definitions from the formalisation presented before. The resulting scheme corresponds to Garble2 in [10].

```
op outputK (fn:fun_t) (r:rand_t) =
    let (n, m, q, aa, bb) = fst fn in
    init m (λ i, (proj r[(n+q−m+i,false)], proj r[(n+q−m+i,true)])).

op valid_outG (oK:outputK_t) (oG:outputG_t) =
  alli (λ i x, x = fst oK[i] ∨ x = snd oK[i]) oG.

op decode (oK:outputK_t) (oG:outputG_t) =
  mapi (λ i x, x=snd oK[i]) oG.
```

Figure 11: Modified garbling scheme

The GGP verifiable computation protocol can be explained as follows. Take the garbling of the circuit to be delegated to be a public key that is given to the worker. The authenticity property immediately yields a weak form of verifiability, whereby the delegating party can securely outsource the computation of a single input. Verifiability is lost when more than one computation is delegated because, intuitively, if the worker sees more than one opening of the circuit, then it may be able to *mix* the two garbled outputs it obtained to generate a new forged one. To enable the verifiable delegation of multiple inputs using the same garbled circuit, the client in the GGP protocol encrypts each newly encoded input using a fresh FHE key pair. This ensures that, whilst the worker is still able to evaluate the garbled circuit (homomorphically) it will never be able to combine two garbled outputs into a new one: intuitively, the FHE guarantees that the different evaluations become independent, in the sense that one cannot (even homomorphically) use data from previous evaluations to produce a forgery on the next one.

Our formalisation of an abstract FHE scheme in EasyCrypt (Figure 12) is extremely simple. It is a standard public key encryption scheme, for which there exists an additional associated algorithm which, given a transformation function g, maps fresh ciphertexts encrypting any message x to homomorphically transformed ones encrypting $g(x)$. As shown below, for our purposes it suffices to formalize and axiomatize the homomorphism w.r.t. the concrete transformation g that performs the evaluation of a garbled circuit. Additionally, the FHE scheme is assumed to be IND−CPA-secure.

```
op gen : randg_t → pkey_t * skey_t.
op enc : pkey_t → plaini_t → rande_t → cipheri_t.
op hom_eval : (inputG_t → outputG_t) → cipheri_t → ciphero_t.
op dec : skey_t → ciphero_t → plaino_t.

axiom homomorphic : ∀ f x r rg re,
      Scheme.validRand f r ⇒ validInputs f x ⇒
      let fG = funG f r in
      let iK = inputK f r in
      let xG = Input.encode iK x in
      let oK = outputK f r in
            let (pk,sk) = gen rg in
            let ci = enc pk xG re in
            let co = hom_eval (Scheme.evalG fG) ci in
                  dec sk co = evalG fG xG.
```

Figure 12: Abstract Fully-Homomorphic Encryption scheme

We present our proof of security for the GGP protocol as an additional contribution, which may be of independent interest. The

structure of our argument is markedly simpler than that presented in [18], as it directly relies on the formalisation of garbling schemes subsequently put forth by Bellare et al. [10, 11]. On the other hand, our formalisation complements the presentation in [11], where a detailed proof is presented only for one-time secure schemes.

We show that the security of the GGP protocol can be proven solely based on the security of the underlying FHE scheme and the adaptive authenticity of the underlying garbling scheme denoted aut1 in [11]. In this form of authenticity, the attacker gets to see the garbled circuit before she chooses the input for which she wishes to produce a forged output. We do not require obliviousness in our proof. As shown in [11], this is needed for a one-time-secure variant of the GGP construction that does not rely on FHE and uses only Yao's garbled circuits. However, it is no longer required when we can rely on FHE for privacy.

We split our presentation in two parts: first we describe the necessary steps to establish the correctness and authenticity properties of the garbling scheme; then we sketch the proof of security for the GGP protocol.

5.1 Correctness and authenticity of the garbling scheme

Proving the correctness of the modified garbling scheme involved little overhead with respect to the proof constructed for the original one presented in the previous sections. Indeed, the bulk of the inductive argument that establishes a correct opening of all the gates throughout the circuit is identical for both constructions, and only the final decoding operation needs to be handled differently.

Our proof of adaptive authenticity for Garble2 reflects the current state of the art: to the best of our knowledge, there is currently no proof that this construction meets the notion of aut1 security proposed in [11] down to standard assumptions. Indeed, Bellare, Hoang and Rogaway point out a gap in the original proof of security of the GGP protocol that corresponds precisely to this issue. One plausible solution to this problem, suggested in [11], is to simply assume that Garble2 provides this level of security. We go a bit further, and reduce the adaptive authenticity of Garble2 to a simpler property, which we call aPriv. The question of whether Garble2 satisfies aPriv can be seen both as an interesting open problem and a computational assumption that underlies the security of our instantiation of the GGP protocol. The details follow.

Theorem 5 (Adaptive Authenticity)**.** *For all efficient authenticity adversaries $\mathcal{A}$ against Garble2, we can construct efficient adversary $\mathcal{B}$ against the* aPriv *property of Garble2 such that:*

$$\mathsf{Adv}^{\mathsf{Auth}}_{\mathit{Garble2}}(\mathcal{A}) \leq 2 \cdot \mathsf{m} \cdot \left(\mathsf{Adv}^{\mathsf{aPriv}}_{\mathit{Garble2}}(\mathcal{B}) + 2^{-\lambda}\right).$$

where m *is an upper bound on the number of output wires, and λ is the length of the random tokens used in the dual-key cipher.*

Proof (Sketch). Our proof combines the proof strategies of Bellare et al. [10] for (non-adaptive) authenticity and the proof sketch presented in the GGP paper [18] for the verifiability of the GGP protocol. We first guess (one of) the output bit(s) i that will be flipped by the adversary's forgery, and we also bet on which will be the correct value of this bit β: this costs us a factor of $2 \cdot \mathsf{m}$ in the reduction. We then rely on the aPriv property to jump to a final game where we replace the function f chosen by the adversary with another function f'. Function f' is identical to f in all gates except output gate i, where all entries in the truth table now encode bit β. The aPriv property has been tailored to be as weak as possible and still allow this transition. It imposes that, for adversarially chosen (f, f', i, β) such that f and f' are related as described above, the garblings of f and f' are indistinguishable. The attacker gets to adaptively choose an input x, for which it will receive an encoding (under the usual restriction that f and f' collide on input x); and it is also given the two decoding tokens for output bit i (not the entire decoding key).

Observing that, by construction, the garbling of f' is independent from the token that the adversary needs to guess in order to successfully forge (this corresponds to $\bar{\beta}$), the proof is completed by showing that a forgery in this final game is equivalent to guessing a uniformly distributed token. □

5.2 GGP protocol security proof sketch

The proof of correctness for the GGP protocol is straightforward once the correctness of the underlying FHE is assumed and that of the underlying garbling scheme is proven. Similarly, as presented in [18], the privacy result follows via a hybrid argument from the IND−CPA security of the underlying FHE. We sketch here, in more detail, our security proof for the verifiability property, which is where we depart from [18] and establish an equivalent result using a simpler game hopping argument.

Theorem 6 (Verifiability)**.** *For all efficient verifiability adversaries $\mathcal{A}$ against the GGP protocol, we can construct efficient adversaries $\mathcal{B}^{\mathsf{FHE}}$ and $\mathcal{B}^{\mathsf{Garble2}}$ such that:*

$$\mathsf{Adv}^{\mathsf{Verif}}_{\mathit{GGP}}(\mathcal{A}) \leq \mathsf{q}^2 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathit{FHE}}(\mathcal{B}^{\mathsf{FHE}}) + \mathsf{q} \cdot \mathsf{Adv}^{\mathsf{Auth}}_{\mathit{Garble2}}(\mathcal{B}^{\mathsf{Garble2}}).$$

where q *is an upper bound on the number of problem generation queries placed by the adversary.*

Proof (Sketch). The verifiability adversary can place up to q queries to a problem generation oracle. Our first step in the proof is to guess which of these queries will be selected by the adversary to produce a forgery. We call this query l. This introduces a factor of q in the reduction. We then use the security of the underlying FHE to modify the answers to all the problem generation queries except query l, so that the encoding provided to the adversary in encrypted form is totally unrelated to the garbled circuit that has been delegated. In this final game, the proof can be concluded by reducing directly to the (adaptive) authenticity property for the underlying garbling scheme that we have presented above. □

6. Experimental Results

In this section we present a performance evaluation of an SFE implementation generated from the EasyCrypt formalisation described in Sections 2 to 4. The major part of this implementation has been obtained via the extraction mechanism included in the more recent version of EasyCrypt, with the exception of the low-level operations left abstract in the formalisation, namely:

- abstract core libraries such as cyclic algebraic structures, DKC encryption, or the entropy-smoothing hash of SomeOT. These are implemented using the CryptoKit library.[4] As DKC scheme we use the AES-based algorithm presented by Bellare et al [10].
- a front-end that parses circuits and runs the extracted SFE code (instrumented to perform time measurements).

We fix the bound c on circuit sizes to be the largest OCaml integer ($2^{k-1}-1$ on a k-bit machine), allowing us to represent circuits without having to use arbitrary precision arithmetic whilst remaining large enough to encode all practical circuits. We use this same value to instantiate n. Inputs are generated randomly using OCaml's Rand module, and the cryptographic randomness is generated using CryptoKit's RNG.

Our preliminary results show that, whilst being slower than optimized implementations of SFE [19, 20], the performance of the *extracted* program is compatible with real-world deployment, providing some evidence that the (unavoidable) overhead implied

[4] See http://forge.ocamlcore.org/projects/cryptokit/

by our formal verification and code extraction approach is not prohibitive. We now present our experimental results in details.

METHODOLOGY. In addition to the overall execution time of the SFE protocol and the splitting of the processing load between the two involved parties, we also measure various speed parameters that permit determining the weight of the underlying components: the time spent in executing the OT protocol, and the garbling and evaluation speeds for the garbling scheme. Our measured execution times do not include serialisation and communication overheads (which are out of the scope of this work), nor do they include the time to sample the randomness (which can be pre-generated). We run our experiments on an x86-64 Intel Core 2 Duo clocked at a modest 1.86 GHz with a 6MB L2 cache. The extracted code and parser are compiled with ocamlopt version 4.00.1. The tests are run in isolation, using the OCamlSys.time operator for time readings. We run tests in batches of 100 runs each, noting the median of the times recorded in the runs.

TEST CIRCUITS AND RESULTS. Our measurements are conducted over circuits made publicly available by the cryptography group at the University of Bristol, [5] precisely for the purpose of enabling the testing and benchmarking of multiparty-computation and homomorphic encryption implementations. A simple conversion of the circuit format is carried out to ensure that the representation matches the conventions adopted in the formalisation.

A subset of our results are presented in Table 1, for circuits COMP32 (32-bit signed number less-than comparison), ADD32 (32-bit number addition), ADD64 (64-bit number addition), MUL32 (32-bit number multiplication), AES (AES block cipher), SHA1 (SHA-1 hash algorithm). The semantics of the evaluation of the arithmetic circuits is that each party holds one of the operands. In the AES evaluation we have that P1 holds the 128-bit input block, whereas P2 holds the 128-bit secret key. Finally, in the SHA1 example we model the (perhaps artificial) scenario where each party holds half of a 512-bit input string.

We present the number of gates for each circuit as well as the execution times in milliseconds. A rough comparison with results presented in, for example [19], where an execution of the AES circuit takes roughly 1.6 seconds (albeit including communications overhead and randomness generation time) allows us to conclude that real-world applications are within the reach of the implementations generated using the approach described in this paper. Furthermore, additional optimisation effort can lead to significant performance gains, e.g., by resorting to hardware support for low-level cryptographic implementations as in [20], or implementing garbled-circuit optimisations such as those allowed by XOR gates [21].

7. Related work

We concentrate on closely related work on the verification of multiparty computation protocols and cryptographic software implementations. We refer to Appendix B for other related work in cryptography, and to [22] for a more extensive account of the use of formal methods in (symbolic and computational) cryptography.

Dahl and Damgård et al. [23] consider the symbolic analysis of specifications extracted from two-party SFE protocol descriptions, and show that the symbolic proofs of security are computationally sound in the sense that they imply security in the standard UC model for the original protocols. This complements earlier work by Backes et al. [24], who develop computationally sound methods for protocols that use secure multi-party computation as a primitive. However, these works do not consider verified implementations.

WYSTERIA [25] is a new programming language for mixed-mode multiparty computations. Its design is supported by a rigorous pen-and-paper proof that typable programs do not leak information in unintended ways. However, their guarantees are cast in the setting of language-based security, rather than in the usual style of provable security. Independently, Pettai and Laud [26] have developed a static analysis for proving that SHAREMIND applications are secure against active adversaries. They show that programs accepted by their analysis satisfy a simulation-based notion called black-box security.

There have also been efforts to use formal methods for generating circuits. For instance, Holzer *et al* [27] present a compiler that uses CBMC to translate ANSI C programs into circuits that can be used as inputs to the secure computation framework of Huang et al. [19]. This compiler can also be used as a front-end to our verified implementation of Yao's protocol.

There have been many attempts to develop tools that support verified implementations. Many of these tools support proofs in the symbolic model of cryptography. There are some notable exceptions, however. For instance, Cadé and Blanchet [28] present a mechanism to generate functional code from CryptoVerif models, and use it to generate a verified implementation of SSH. Similarly, Almeida *et al* [29] use a verified compiler to generate a verified x86 implementation of PKCS#1 v2.1 encryption from an EasyCrypt formalization. Finally, Kuesters *et al* [30] develop a framework to verify cryptographic applications written in Java, and use it to verify a non-trivial cloud storage system. Another practical approach to achieve high-confidence cryptographic implementations is to use prototyping systems for cryptography [6–9]. In a series of works starting from [31], Akinyele *et al* advocate the convergence between such systems and verification tools; in particular, their latest work [32] combines AutoBatch and EasyCrypt to generate verified implementations of batch verifiers.

8. Conclusions and future work

We have demonstrated by example that it is already possible to build nearly practical verified implementations of cryptographic systems using EasyCrypt. However, further steps are required to encourage the systematic development of verified implementations; the next logical step is to achieve a tight integration between prototyping tools like [6–9] and verification tools like EasyCrypt.

In addition, we believe that it will be essential to support principled approaches for verifying implementations of cryptographic systems, taking fully into account the environment in which they are deployed in the real world. This involves developing and formalizing non-destructive compositionality results, which has recently been identified as a major challenge for cryptography itself [33]. The natural starting point for this task are the existing theoretical frameworks that deal with this issue, such as the Universal Composability [34] framework. More immediately, we also intend to formalize recent developments in multi-party and verifiable computation, notably [15].

References

[1] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Trans. Dependable Sec. Comput.*, vol. 5, no. 4, 2008.

[2] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin, "Computer-aided security proofs for the working cryptographer," in *CRYPTO*, 2011.

[3] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *POPL*, 2010.

[4] J. B. Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Z. Béguelin, "Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols," in *ACM CCS*, 2012.

[5] http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/

Table 1: Execution times (milliseconds): total (TTime), P2 stage 1 garbling (P2 S1 GT), P2 stage 1 OT (P2 S1 OT), P1 stage 1 OT (P1 S1 OT), P2 stage 2 OT (P2 S2 OT), P1 stage 2 OT (P2 S1 OT) and P1 stage 2 evaluation (P1 S2 ET).

Circuit	NGates	TTime	P2 S1 GT	P2 S1 OT	P1 S1 OT	P2 S2 OT	P1 S2 OT	P1 S2 ET
COMP32	301	253	2	48	50	102	50	1
ADD32	408	251	3	48	49	101	49	1
ADD64	824	494	6	95	97	197	97	3
MUL32	12438	356	95	47	47	98	48	22
AES	33744	1301	248	192	199	403	200	60
SHA1	106761	2825	803	366	367	744	366	180

[5] G. Barthe, G. Danezis, B. Grégoire, C. Kunz, and S. Z. Béguelin, "Verified computational differential privacy with applications to smart metering," in *CSF*, 2013.

[6] J. A. Akinyele, C. Garman, I. Miers, M. W. Pagano, M. Rushanan, M. Green, and A. D. Rubin, "Charm: a framework for rapidly prototyping cryptosystems," *J. Crypt. Eng.*, vol. 3, no. 2, 2013.

[7] Y. Ejgenberg, M. Farbstein, M. Levy, and Y. Lindell, "SCAPI: The secure computation application programming interface," Cryptology ePrint Archive, Report 2012/629, 2012.

[8] M. Barbosa, D. Castro, and P. F. Silva, "Compiling CAO: From cryptographic specifications to C implementations," in *POST*, 2014.

[9] S. Browning and P. Weaver, "Designing tunable, verifiable cryptographic hardware using Cryptol," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, 2010.

[10] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits," in *ACM CCS*, 2012.

[11] M. Bellare, V. Hoang, and P. Rogaway, "Adaptively secure garbling with applications to one-time programs and secure outsourcing," in *Advances in Cryptology ASIACRYPT 2012*, ser. Lecture Notes in Computer Science, X. Wang and K. Sako, Eds., vol. 7658. Springer Berlin Heidelberg, 2012, pp. 134–153. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34961-4_10

[12] M. Bellare and S. Micali, "Non-interactive oblivious transfer and applications," in *CRYPTO*, 1989.

[13] M. Naor and B. Pinkas, "Efficient oblivious transfer protocols," in *SODA*, 2001.

[14] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct NIZKs without PCPs," in *EUROCRYPT*, 2013.

[15] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE S&P*, 2013.

[16] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular code-based cryptographic verification," in *ACM CCS*, 2011.

[17] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *IEEE S&P*, 2013.

[18] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *CRYPTO*, 2010.

[19] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in *USENIX Security*, 2011.

[20] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *IEEE S&P*, 2013.

[21] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *ICALP*, 2008.

[22] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *POST*, 2012.

[23] M. Dahl and I. Damgård, "Universally composable symbolic analysis for two-party protocols based on homomorphic encryption," in *EUROCRYPT*, 2014.

[24] M. Backes, M. Maffei, and E. Mohammadi, "Computationally sound abstraction and verification of secure multi-party computations," in *FSTTCS*, 2010.

[25] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *IEEE S& P*, May 2014.

[26] M. Pettai and P. Laud, "Automatic proofs of privacy of secure multi-party computation protocols against active adversaries," Cryptology ePrint Archive, Report 2014/240, 2014.

[27] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *ACM CCS*, 2012.

[28] D. Cadé and B. Blanchet, "Proved generation of implementations from computationally secure protocol specifications," in *POST*, 2013.

[29] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations," in *ACM CCS*, 2013.

[30] R. Küsters, E. Scapin, T. Truderung, and J. Graf, "Extending and Applying a Framework for the Cryptographic Verification of Java Programs," Cryptology ePrint Archive, Report 2014/038, 2014.

[31] J. A. Akinyele, M. Green, S. Hohenberger, and M. W. Pagano, "Machine-generated algorithms, proofs and software for the batch verification of digital signature schemes," in *ACM CCS*, 2012.

[32] J. A. Akinyele, G. Barthe, B. Grégoire, B. Schmidt, and P.-Y. Strub, "Certified synthesis of efficient batch verifiers," in *CSF*, 2014, to appear.

[33] C. E. Landwehr, D. Boneh, J. C. Mitchell, S. M. Bellovin, S. Landau, and M. E. Lesk, "Privacy and cybersecurity: The next 100 years," *Proceedings of the IEEE*, vol. 100, 2012.

[34] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS*, 2001.

[35] Y. Lindell and B. Pinkas, "A proof of security of Yao's protocol for two-party computation," *J. Cryptol.*, vol. 22, no. 2, Apr. 2009.

[36] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *ASIACRYPT*, 2009.

[37] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "Reusable garbled circuits and succinct functional encryption," in *STOC*, 2013.

[38] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay a secure two-party computation system," in *USENIX Security*, 2004.

[39] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Tasty: Tool for automating secure two-party computations," in *ACM CCS*, 2010.

[40] Y. Lindell, B. Pinkas, and N. P. Smart, "Implementing two-party computation efficiently with security against malicious adversaries," in *SCN*, 2008.

[41] L. Babai, "Trading group theory for randomness," in *STOC*, 1985.

[42] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM J. Comput.*, vol. 18, no. 1, 1989.

[43] S. Arora and S. Safra, "Probabilistic checking of proofs; a new characterization of NP," in *FOCS*, 1992.

[44] L. Babai, L. Fortnow, and C. Lund, "Non-deterministic exponential time has two-prover interactive protocols," in *FOCS*, 1990.

[45] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy, "Checking computations in polylogarithmic time," in *STOC*, 1991.

[46] J. Kilian, "A note on efficient zero-knowledge proofs and arguments (extended abstract)," in *STOC*, 1992.

[47] ——, "Improved efficient arguments (preliminary version)," in *CRYPTO*, 1995.

[48] S. Micali, "CS proofs (extended abstracts)," in *FOCS*, 1994.

[49] B. Applebaum, Y. Ishai, and E. Kushilevitz, "From secrecy to soundness: Efficient verification via secure computation," in *ICALP*, 2010.

[50] K.-M. Chung, Y. T. Kalai, and S. P. Vadhan, "Improved delegation of computation using fully homomorphic encryption," in *CRYPTO*, 2010.

[51] R. Gennaro and V. Pastro, "Verifiable computation over encrypted data in the presence of verification queries," Cryptology ePrint Archive, Report 2014/202, 2014.

[52] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *ASIACRYPT*, 2010.

[53] H. Lipmaa, "Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments," in *TCC*, 2012.

A. Background

A.1 Yao's Garbled Circuits and Yao's SFE Protocol

Excellent descriptions of Yao's contributions can be found in [10, 35]. Yao's idea of garbling the circuit computing f consists informally of: i. expressing such a circuit as a set of truth tables (one for each gate) and meta information describing the wiring between gates; ii. replacing the actual Boolean values in the truth tables with random cryptographic keys, called *labels*; and iii. translating the wiring relations using a system of *locks*: truth tables are encrypted one label at a time so that, for each possible combination of the input wires, the corresponding labels are used as encryption keys that lock the label for the correct Boolean value at the output of that gate. Then, given a garbled circuit for f and a set of labels representing (unknown) values for the input wires encoding x_1 and x_2, one can obliviously evaluate the circuit by sequentially computing one gate after another: given the labels of the input wires to a gate, only one entry in the corresponding truth table will be decryptable, revealing the label of the output wire. The output of the circuit will comprise the labels at the output wires of the output gates.

To build a SFE protocol between two honest-but-curious parties, one can use Yao's garbled circuits as follows. Bob (holding x_2) garbles the circuit and provides this to Alice (holding x_1) along with: i. the label assignment for the input wires corresponding to x_2, and ii. all the information required to decode the Boolean values of the output wires. In order for Alice to be able to evaluate the circuit, she should be able to obtain an the correct label assignment for x_1. Obviously, Alice cannot reveal x_1 to Bob, as this would totally destroy the goal of SFE. Furthermore, Bob cannot reveal information that would allow Alice to encode anything other than x_1, since this would reveal more than $f(x_1, x_2)$. To solve this problem, Yao proposed the use of an *oblivious transfer* (OT) protocol. This is a (lower-level) SFE protocol for a very simple functionality that allows Alice to obtain the labels that encode x_1 from Bob, without revealing anything about x_1 and learning nothing more than the labels she requires.[6] The protocol is completed by Alice evaluating the circuit, recovering the output, and providing the output value back to Bob.[7] The combined security of the garbled circuit technique and the OT protocol guarantee that f can be securely evaluated in this manner.

A.2 Verifiable Computation

We recall the notion of a Verifiable Computation (VC) scheme introduced by Gennaro, Gentry and Parno [18] and informally describe the associated security notions. Subsequent works have taken slightly different approaches to the formalisation of security, namely by strengthening the verifiability requirement to allow for fully adaptive queries and, sometimes, dropping the privacy requirement. However, the essence of the primitive remains the same.

A VC scheme is a protocol between two polynomial-time parties, a client and a worker, that enables them to collaborate on the computation of a function $f : \{0,1\}^n \longrightarrow \{0,1\}^m$. It consists of four steps:

Preprocessing A one-time stage in which the client computes some auxiliary (public and private) information associated with f. This phase can take time comparable to computing the function from scratch, but it is performed only once, and its cost is amortized over all the future executions.

Input Preparation When the client wants the worker to compute $f(x)$, it prepares some auxiliary (public and private) information about x. The public information is sent to the worker.

Output Computation Once the worker has the public information associated with f and x, it computes a string π_x which encodes the value $f(x)$ and returns it to the client.

Verification From the value π_x, the client can compute the value $f(x)$ and verify its correctness.

The crucial efficiency requirement is that Input Preparation and Verification must take less time than computing f from scratch (ideally linear time, $O(n + m)$). Also, the Output Computation stage should take roughly the same amount of computation as f, so as not to overburden the worker.

Two security properties are required from a VC protocol, verifiability and privacy. Privacy imposes that, given the output of the Preprocessing stage, a malicious worker is unable to distinguish between two input-prepared values of its choosing, given access to an oracle from which it can obtain arbitrary input-prepared values.

Verifiability is defined by means of a game in which the malicious worker gets access to the output of the Preprocessing stage, and can request up to q input-prepared values of its choosing. The adversary is then required to output a proof which causes the verification procedure to accept an incorrect value for the delegated computation on one of the q inputs it requested from its oracle.

In the definition put forth in the GGP [18] paper, which we adopt, the worker cannot be allowed to know whether verification procedures were successful or not. This is due to a limitation of the GGP construction, which has been addressed in subsequent work. This discussion is not relevant in the context of this paper, where we simply use the GGP protocol to demonstrate the capabilities of our formal verification infrastructure.

B. Additional related work

B.1 Yao's techniques in practice

The original SFE protocol proposed by Yao has been the subject of renewed interest. On one hand, it is now widely accepted that this generic protocol offers the best performance tradeoff to evaluate important classes of functions [36], which means that it plays an important role in the current state-of-the-art of practice-oriented SMPC research. On the other hand, Yao's garbled circuits have found new uses in the construction of novel and powerful cryptographic primitives such as Functional Encryption [37] and Verifiable Computation [18] that are intrinsically *higher order* in their operation, i.e., they take as input parameters the description of functions that must be applied to secret data.

Fairplay [38] was the first of a series of works that demonstrated the practical applicability of Yao's technique. Tasty [39] was later introduced as a general framework to generate efficient implementations of secure computation protocols, following a series of papers where various optimizations and extensions of Yao's protocol were proposed [21, 36, 40]. The secure computation framework of Huang et al. [19] introduces an API-based framework in which to implement secure computation applications based on Yao's protocol, exploring various implementation optimizations and communications pipelining that improve the performance of the resulting

[6] Luckily, efficient OT protocols exist that can be used for this specific purpose, thereby eliminating what could otherwise be a circular dependency.

[7] This is a simplified view of Yao's protocol. It suffices because we are dealing with honest-but-curious adversaries assumed to follow the protocol.

implementations. This system was proposed as a kind of *virtual machine* in which circuits can be securely evaluated.

The recently proposed JustGarble [20] system gives a highly efficient implementation of garbled circuits that can be used for various applications, namely secure two-party computation. The implementation relies of highly efficient constructions of dual-key ciphers and resorts to hardware implementations of various operations, namely AES.

B.2 Verifiable Computation

The problem of efficiently verifying complex computations was first addressed in the context of Interactive Proofs [41, 42], where a powerful (possibly super-polynomial) prover can (probabilistically) convince a weak verifier of the validity of statements that the verifier could not compute on its own. Probabilistically checkable proofs (PCPs) permit achieving the same goal, whilst reducing the workload on the verifier by limiting the parts of the proof that need to be checked [43–45]; other contributions that aimed to reduce the verifier's effort can be found in the work of Kilian [46, 47] and Micali [48].

More recently, the advent of cloud computing has led researchers to consider application scenarios where also the work of the prover, including the actual computation to be verified, is polynomial-time, so as to capture computation delegation. An example of this approach is the GGP verifiable computation protocol that we covered in this paper, as well as other FHE-based solutions [18, 49–51]. A very promising line of work in a different direction has considered alternative ways of representing computations using arithmetic constructions [14, 52, 53], which have very recently given rise to quasi-practical implementations [15].