HAL
open science

A Fast and Verified Software Stack for Secure Function Evaluation

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Francois Dupressoir, Benjamin Grégoire, Vincent Laporte, Vitor Pereira

To cite this version:

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Francois Dupressoir, Benjamin Grégoire, et al.. A Fast and Verified Software Stack for Secure Function Evaluation. CCS 2017 - ACM SIGSAC Conference on Computer and Communications Security, Oct 2017, Dallas, United States. pp.1-18. hal-01649104

HAL Id: hal-01649104

https://hal.science/hal-01649104

Submitted on 27 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Fast and Verified Software Stack for Secure Function Evaluation

José Bacelar Almeida
INESC TEC and
Universidade do Minho, Portugal

Manuel Barbosa
INESC TEC and FCUP
Universidade do Porto, Portugal

Gilles Barthe
IMDEA Software Institute, Spain

François Dupressoir
University of Surrey, UK

Benjamin Grégoire
Inria Sophia-Antipolis, France

Vincent Laporte
IMDEA Software Institute, Spain

Vitor Pereira
INESC TEC and FCUP
Universidade do Porto, Portugal

ABSTRACT

We present a high-assurance software stack for secure function evaluation (SFE). Our stack consists of three components: i. a verified compiler (CircGen) that translates C programs into Boolean circuits; ii. a verified implementation of Yao's SFE protocol based on garbled circuits and oblivious transfer; and iii. transparent application integration and communications via FRESCO, an open-source framework for secure multiparty computation (MPC). CircGen is a general purpose tool that builds on CompCert, a verified optimizing compiler for C. It can be used in arbitrary Boolean circuit-based cryptography deployments. The security of our SFE protocol implementation is formally verified using EasyCrypt, a tool-assisted framework for building high-confidence cryptographic proofs, and it leverages a new formalization of garbled circuits based on the framework of Bellare, Hoang, and Rogaway (CCS 2012). We conduct a practical evaluation of our approach, and conclude that it is competitive with state-of-the-art (unverified) approaches. Our work provides concrete evidence of the feasibility of building efficient, verified, implementations of higher-level cryptographic systems. All our development is publicly available.

KEYWORDS

Secure Function Evaluation, Verified Implementation, Certified Compilation

1 INTRODUCTION

Cryptographic engineering is the domain-specific area of software engineering that brings cryptography to practice. It encompasses projects that maintain widely used cryptographic libraries such as OpenSSL,[1] s2n[2] and Bouncy Castle,[3] as well as prototyping frameworks such as CHARM [1] and SCAPI [31]. More recently, a series of groundbreaking cryptographic engineering projects have emerged, that aim to bring a new generation of cryptographic protocols to real-world applications. In this new generation of protocols, which has matured in the last two decades, secure computation over encrypted data stands out as one of the technologies with the highest potential to change the landscape of secure ITC, namely by improving cloud reliability and thus opening the way for new secure cloud-based applications. Projects that aim to bring secure computation over encrypted data to practice include FRESCO[4] [27], TASTY [38] and Sharemind [21].

In contrast to other areas of software engineering for critical systems, the benefits of formal verification for cryptographic engineering have been very limited, with some recent and notable exceptions [2, 3, 8, 18, 22, 33]. The reasons for this are well known: cryptographic software is a challenge for high-assurance software development due to the tension that arises between complex specifications and the need for very high efficiency—security is supposed to be invisible, and current verification technology comes with a performance penalty. The exceptions mentioned above mark the emergence of a new area of research: high-assurance cryptography. This aims to apply formal verification to both cryptographic security proofs and the functional correctness and security of cryptographic implementations.

In this paper we demonstrate that a tight integration of high-assurance cryptography and cryptographic engineering can deliver the combined benefits of provable security and best cryptographic engineering practices at a scale that significantly exceeds previous experiments (typically carried out on core cryptographic primitives). We deliver a fast and verified software stack for secure computation over encrypted data. This choice is motivated by several factors. First, as mentioned above, this technology is among the foremost practical applications of cryptography and is a fundamental building block for making cloud computing secure. Second, it is a tremendous challenge for high-assurance cryptography, as its security proofs are markedly distinct from prior work in formalizing reductionist arguments.

Contributions. We present a high-assurance and high-speed software stack for secure multi-party computation. Figure 1 presents the overall architecture of the stack. The lowest-level component is FRESCO [27]; an existing, practical, open-source, framework for secure multi-party computation, which we use for communications and input/output. The correctness of this framework (but not its security) is part of our trusted computing base, as verifying the

A version of this report appears in the proceedings of CCS 2017.

[1] http://openssl.org
[2] http://https://github.com/awslabs/s2n
[3] https://www.bouncycastle.org/
[4] https://github.com/aicis/fresco

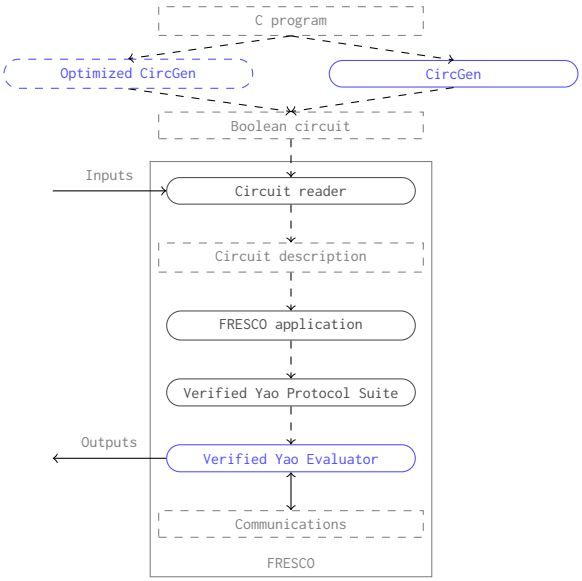

Figure 1: Verified cryptographic software stack. Blue rectangles identify the verified components of the stack, while black rectangles represent part of our trusting computing base. Dashed blue rectangles are partially verified elements and in dashed black rectangles one can find intermediate input/output items.

correctness of a Java-based communications infrastructure is out of the scope of high-assurance cryptography.

The intermediate component of our stack is a verified implementation of Yao's secure function evaluation (SFE) protocol [57] based on garbled circuits and oblivious transfer. This protocol allows two parties P_1 and P_2, 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 the cloud [41]; we allow for both scenarios where the function is public and both sides provide inputs and scenarios where one party provides the (secret albeit with leaked topology) circuit to be computed and the other party provides the input to the computation.

Our implementation is machine-checked in EasyCrypt[5] [7, 9], an interactive proof assistant with dedicated support to perform game-based cryptographic proofs in the computational model. Our proof leverages the foundational framework put forth by Bellare, Hoang and Rogaway [12] for the security of Yao's garbled circuits. Our construction of SFE relies on an n-fold extension (where n is the size of the selection string–or the circuit's input) of the oblivious transfer protocol by Bellare and Micali [13], in the hashed version presented by Naor and Pinkas [47]. The implementation is proved secure relative to standard assumptions: the Decisional Diffie-Hellman problem, and the existence of entropy-smoothing hash functions and pseudorandom functions.

[5]https://www.easycrypt.info

The higher-level component of our stack is a verified optimizing compiler from C programs to Boolean circuits that we call CircGen. Our compiler is mechanically verified using the Coq proof assistant, and builds on top of CompCert [43], a verified optimizing compiler for C programs. It reuses the front- and middle-end of CompCert (introducing an extra loop-unrolling optimization) and it provides a new verified back-end producing Boolean circuits. The back-end includes correctness proofs for several program transformations that have not previously been formally verified, including the translation of RTL programs into guarded form and a memory-agnostic static single assignment (SSA) form. Our proof of semantic preservation is conditioned on the existence of an external oracle that provides functionally correct Boolean circuits for basic operations in the C language, such as 32-bit addition and multiplication. The low-level circuits used in our current implementation for these operations have *not* been formally verified and are hence part of our trusted computing base. Verifying Boolean circuits for native C operations can be done either in Coq or using other verification techniques and it is orthogonal to the reported verification effort.

The Boolean circuits generated by CircGen compare well with alternative unverified solutions, namely CBMC-GC[6] [34], although they are slightly less efficient (as would be expected). To widen the applicability of CircGen to scenarios where speed is more important than assurance, we also implement some (yet unverified) global post-processing optimizations that make CircGen a good alternative to CBMC-GC for high-speed applications.

Our work delivers several generic building blocks (the Boolean circuit compiler, a verified implementation of oblivious transfer, ...) that can be reused by many other verified cryptographic systems. However, the main strength of our results resides in the fact that, for the first time, we are able to produce a joining of high-assurance cryptography and cryptography engineering that covers all the layers in a (passively) secure multiparty computation software framework.

Challenges. The development of the software stack raised several challenges, which we now highlight.

Machine-checked proofs of computational security. EasyCrypt [7, 9] is an interactive proof assistant with dedicated support to perform game-based cryptographic proofs. It has been used for several emblematic examples, including signatures and encryption schemes. Formalizing the proof of security for our SFE protocol in EasyCrypt involved formalizing two generic proof techniques that had not previously been considered: hybrid arguments and simulation-based security proofs.

In contrast to other standard techniques, which remain within the realm of the relational program logic that forms the core of EasyCrypt (i.e., it is used to verify transitions between successive games), hybrid arguments and simulation-based proofs lie at the interface between this relational program logic and the higher-order logic of EasyCrypt in which security statements are expressed and proved. Specifically, hybrid arguments combine induction proofs and proofs in the relational program logic. Similarly, simulation-based security proofs intrinsically require existential quantification over adversarial algorithms and the ability to instantiate security

[6]http://forsyte.at/software/cbmc-gc/

models with concrete algorithms (the simulators) that serve as witnesses as to the validity of the security claims. These two forms of reasoning excercise the expressive power of EasyCrypt's ambient logic, and are thus markedly distinct from the simple security arguments typically addressed by other similar tools like CryptoVerif [19]. Secure function evaluation is also a challenging test case in terms of its scale. Indeed, EasyCrypt had so far been used primarily for primitives and to a lesser extent for (components) of protocols. While these examples can be intricate to verify, there is a difference of scale with more complex cryptographic systems, such as SFE, which involve several layers of cryptographic constructions.

Realizing our broader goal required several improvements to the EasyCrypt tool. In particular, the complexity and scale of the proof developed here guided several aspects of EasyCrypt's development to support compositional simulation-based proofs, and the aim of producing executable code from machine-checked specifications served as initial motivation for EasyCrypt's code extraction mechanism. We contribute a generic formalization of hybrid arguments that has since been included in EasyCrypt's library of game transformations.

High-assurance and high-speed implementations. Our implementation of Yao's protocol can be thought of as a secure virtual machine for securely executing arbitrary computations. The challenge is therefore dual: in addition to a verified implementation of this virtual machine of sorts, one needs to generate correct and efficient computation descriptions in a format that can be executed in this virtual computational platform (in this case Boolean circuits). Generating such circuit representations by hand is not realistic, and appropriate tool support is critical if widespread practical adoption is the goal. The requirement of end-to-end verification further imposes that compilation into circuits must itself be verified. CircGen fills this gap from both a high-assurance cryptography perspective—verified outputs incur a small performance penalty—and a cryptographic engineering perspective—it supports unverified optimizations for speed-critical applications.

Highlights of our technical contributions at this level include: (1) the addition of a loop unrolling transformation to the CompCert middle-end that permits converting those programs that can be expressed as circuits into a loop-free form; (2) new intermediate languages in CompCert with corresponding transformations semantics preservation theorems that permit converting loop-free programs gradually into a circuit representation—this includes a new domain-specific transformation into Static Single Assignment (SSA) form; and (3) the formalization of a new target language that captures the semantics of Boolean circuits and permits stating and proving a semantics preservation theorem relating the I/O behavior of an input C program to that of a generated circuit.

Access to the development. The EasyCrypt formalisation of Yao's protocol, as well as its extracted code, can be found at https://ci.easycrypt.info/easycrypt-projects/yao. The code for CircGen can be found at https://github.com/haslab/circgen.

Structure of the paper. In Section 2 we describe the EasyCrypt formalization and the verified implementation of Yao's protocol. In Section 3 we present CircGen, our certified Boolean circuit compiler. In each of these sections, we give micro-benchmarks for the related software component. We then present an overall performance evaluation of the software stack in Section 4. In Section 5 we discuss related work, before making some concluding remarks in Section 6.

Limitations. Our approach covers a comfortable subset of C, but some features are excluded (see Table 2); some of these features will be added in future work, while others are traditionally out of reach for SFE. Moreover, some low-level optimizations have not yet been verified; however, our experiments show that the verified version of the compiler is already surprisingly close to the optimized version for most examples. Finally, our Trusted Computing Base includes the FRESCO platform, Cryptokit (used to instantiate the hash function) and justGarble (used to instantiate the PRF); the formal verification of these components is out of scope of this work.

2 VERIFIED SFE IMPLEMENTATION

We first give an overview of what we prove in EasyCrypt, relating this to established results in the field of cryptography. We do not go into the details of the (publicly available) formalization but include in Appendix A an example-driven presentation of its highlights. The formalization is available online and the various files that compose it can be easily matched to the building blocks in the high-level description we give here. At the end of the section we describe how we obtain our verified implementation from the EasyCrypt formalization.

Yao's protocol in a nutshell. Yao's protocol is based on the concept of garbled circuits. Informally, the idea of garbling a circuit computing f consists of: i. expressing the 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 the correct label assignment for x_1. Obviously, Alice cannot reveal x_1 to Bob, as this would violate the input-privacy goals of SFE. Also, 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

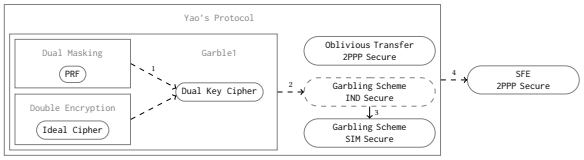

Figure 2: Yao's protocol security proof by BHR [12].

(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.[7] The protocol is completed by Alice evaluating the circuit, recovering the output, and providing the output value back to Bob. Excellent descriptions of Yao's SFE protocol with slightly different security proofs can be found in [12, 44].

A modular proof of security. Our starting point for producing a formally verified implementation of Yao's protocol is to transpose to EasyCrypt the modular security proof by Bellare, Hoang and Rogaway [12] (BHR). The central component in this proof is a new abstraction called a *garbling scheme* that captures the functionality and security properties of the circuit garbling technique that is central to Yao's SFE protocol. This new abstraction was used by BHR to make precise the different security notions that could apply to this garbling step. This permits separating the design and analysis of efficient garbling schemes from higher level protocols, which may rely on different security properties of the garbling component.[8]

Figure 2 shows the structure of the proof of security for Yao's protocol given in [12] (we focus only on the result that is relevant for this paper). We depict constructions as rectangles with grey captions and primitives (i.e., cryptographic abstractions with a well-defined syntax and security model) as rounded rectangles with black captions. Security proofs are represented by dashed arrows and implications between notions as solid arrows. A construction enclosing a primitive in the diagram indicates that the primitive is used as an abstract building block in its security proof. For example, arrow (1) indicates that the first step in the proof is the construction of a *dual key cipher* (DKC) using a standard PRF security assumption via a construction that we call *dual masking*. The same primitive is also constructed from an ideal cipher via the double encryption construction.

A DKC is a tweakable deterministic encryption scheme that can be used to lock secret keys (corresponding to gate output wire labels) and is keyed by two other independent keys (corresponding to gate input wire labels). Informally, the dual masking construction applies two masks to the encrypted key, computed as $\mathsf{PRF}_{K_i}(T)$ for $i = 1, 2$, where T is the tweak. The DKC security model is designed in an ad hoc way to be just strong enough for constructing garbling schemes from a wide range of assumptions, including interesting instantiations such as double encryption. DKC security is a real-or-random notion, where the attacker has an unbounded number of keys to choose from, both for posing as encryption keys and as encrypted keys. One of these secret keys is singled out as the challenge secret key, and it can never be encrypted nor revealed to the attacker (who may see all the other keys). The model also captures the fact that it is convenient to leak the least significant bit of such keys in order to encode the topology of a circuit.

The second step in the proof (2) is to construct a garbling scheme from a (DKC). There are two security definitions for garbling schemes: indistinguishability-based (IND) and simulation-based (SIM). The former is used as a stepping stone (hence its dashed presentation in the diagram) to proving SIM-security. Indeed, the two notions are proven to be equivalent for certain classes of garbling schemes (this is shown as step 3 in the diagram). Proving that a concrete construction called Garble1 achieves IND security is the most challenging part of the proof: it involves a hybrid argument over those wires in the circuit that are *not* visible to an attacker (the security model allows the attacker to observe the opening of the circuit for one concrete input).

The final step (4) in the proof is to show that Yao's technique of combining an oblivious transfer protocol— two-party passively (2PPP) secure—with a SIM-secure garbling scheme yields a 2PPP secure SFE protocol. This step consists of a game-based argument with two relatively simple transitions, but involving simulation-based definitions and combined universal and existential quantifications over adversarial algorithms.

Our Proof. We show in Figure 3 the structure of our EasyCrypt formalization. It is visible in the figure that the main structure of the proof, steps 1-4 are very close to the original proof of [12]. The only deviation here is that we simplify the Dual Key Cipher security game to a slightly stronger variant that is still satisfied by the dual masking instantiation, but which has an internal structure that makes the proof of security of the garbling scheme significantly easier. Intuitively, the difference is that one imposes that the tweak effectively makes encryptions of the same value indistinguishable from each other. This excludes some secure DKC instantiations that we do not consider in this paper. To further simplify our proofs, our DKC security definition is also parametrized by two integer parameters c and pos. The first parameter provides an upper-bound on the number of keys in the game, so that they can all be sampled at the beginning of the security experiment. The second parameter specifies an index in the range [1..c] that will be used in oracle queries as the index for the hidden secret key.

Figure 3 also shows three additional proof steps (5, 7 and 8, shown in blue). These correspond to instantiation (i.e., restricted forms of composition) steps that are often implicit in hand-written cryptographic proofs. For example, suppose construction $C_1^{P_2}$ is proven to be a valid instantiation for primitive P_1 under the assumption that instantiations for abstract primitive P_2 exist. Suppose also that construction $C_2^{P_3}$ is proven to be a valid instantiation of primitive P_2, assuming the existence of a valid instantiation for (lower level) primitive P_3. Then, this implies that $C_1^{C_2}$ is also a valid instantiation of P_1 under assumption P_3.

Such steps are critical in making our main Theorem (Theorem 2.1 below) apply to a concrete and efficient implementation of Yao's protocol that can readily be extracted in to OCaml code from its EasyCrypt description. To obtain such a result our formalization needs to explicitly include theorems that instantiate abstract security results into concrete security bounds for the implementation.

[7] Luckily, efficient protocols for the OT functionality exist, thereby eliminating what could otherwise be a circular dependency.

[8] Garbled circuits are used in Yao's SFE protocol, but have found many other applications in cryptography.

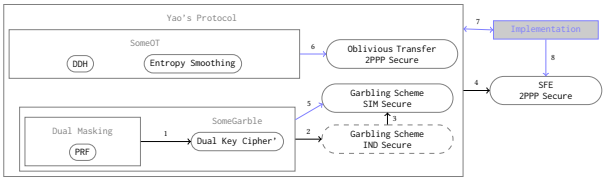

Figure 3: Structure of our verified security proof of an implementation of Yao's protocol.

More precisely, one needs to prove i. that the implementation is functionally equivalent to the composition of a concrete oblivious transfer and garbling schemes; and ii. that this implies that the security bound for the generic SFE security theorem (4 in the figure) can be instantiated into a concrete overall bound by plugging in security bounds obtained by instantiating all intermediate results all the way down to the PRF, DDH and entropy smoothing assumptions.

EasyCrypt enables formalizing both the complex abstract security proofs and the instantiation steps (with very little overhead in the case of the latter). The main theorem in our formalization states the following, for any upper bound c on the total number of wires in the circuit and any upper bound n on the number of input wires in the circuit.

Theorem 2.1. *For all* SFE *adversaries* $\mathcal{A}$ *against the EasyCrypt implementation* Impl *of Yao's protocol, there exist efficient simulator* S *and adversaries* $\mathcal{B}_{\mathsf{DDH}}$, $\mathcal{B}_{\mathsf{ES}}$ *and* $\mathcal{B}^i_{\mathsf{PRF}}$ *for* $i \in [1..c]$, *such that:*

$$\mathsf{Adv}^{\mathsf{SFE}}_{\mathsf{Impl},\mathsf{S}}(\mathcal{A}) \leq c \cdot \varepsilon_{\mathsf{PRF}} + n \cdot \mathsf{Adv}^{\mathsf{DDH}}(\mathcal{B}_{\mathsf{DDH}}) + n \cdot \mathsf{Adv}^{\mathsf{ES}}(\mathcal{B}_{\mathsf{ES}})$$

where $\varepsilon_{\mathsf{PRF}} = \max_{1 \leq i \leq c}(\mathsf{Adv}(\mathcal{B}^i_{\mathsf{PRF}}))$, *and* $\mathsf{Adv}^{\mathsf{PRF}}$, $\mathsf{Adv}^{\mathsf{DDH}}$ *and* $\mathsf{Adv}^{\mathsf{ES}}$ *represent the advantages against the PRF, the Diffie-Hellman group and entropy smoothing hash function used as primitives.*

Using Generic Lemmas. In Cryptography, it is common to repeat proof techniques in different proofs or even inside the same proof. As a side contribution of our work, we formalize a generic hybrid argument that is included 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.

As described in Figure 4, consider an adversary parameterized 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 $\mathsf{Adv}^{\mathsf{Hy}}$ is to guess, in at most n queries to O.o, if it is implementing 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.

```
type input, output, inleaks, outleaks.
module type Orcl = { proc o(_:input) : output }.
module type Orclb = {
  proc leaks(_:inleaks): outleaks
  proc oL(_:input) : output
  proc oR(_:input) : output
}.
module type Advhy (Ob:Orclb, O:Orcl) = {
  proc main () : bool
}.
module Ln (Ob:Orclb, A:Advhy) = {
  module O: Orcl = {
    (* increment C.c and call Ob.oL *)
  }
  module A' = A(Ob, O);
  proc main () : bool = {
    C.c = 0; return A'.main();
  }
}.
module Rn (Ob:Orclb, A:Adv) = {
  (* Same as Ln but use Ob.oR *)
}.
op q : int.

module B(A:Advhy, Ob:Orclb, O:Orcl) = {
  module LR = {
    var l, l0 : int
    proc orcl(m:input):output = {
      var r : output;
      if (l0 < l) r = Ob.oL(m);
      else if (l0 = l) r = O.orcl(m);
      else r = Ob.oR(m);
      l = l + 1; return r;
    }
  }
  module A' = A(Ob,LR)
  proc main():outputA = {
    var r:outputA;
    LRB.l0 ←$
[0..q−1]; LRB.l = 0;
    return A'.main();
  }
}.
```

```
lemma Hybrid: ∀ (Ob:Orclb{C,B}) (A:Advhy {C,B,Ob}),
  Pr[Ln(Ob,A): res∧ C.c ≤ n] − Pr[Rn(Ob,A): res∧ C.c ≤ n] =
    q * (Pr[Ln(Ob,B(A)): res∧ B.l ≤ n ∧ C.c ≤ 1] −
          Pr[Rn(Ob,B(A)): res∧ B.l ≤ n ∧ C.c ≤ 1]).
```

Figure 4: Hybrid argument lemma.

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 $\mathsf{Adv}^{\mathsf{Hy}}\{\mathsf{C},\mathsf{B},\mathsf{O}_b\}$ 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 $\mathsf{Adv}^{\mathsf{Hy}}$, the partially-applied module $\mathcal{B}(\mathcal{A})$ implements $\mathsf{Adv}^{\mathsf{Hy}}$ as well and can therefore be plugged in anywhere a module of type $\mathsf{Adv}^{\mathsf{Hy}}$ 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.

An application example of the generic hybrid argument is our proof of security of the oblivious transfer protocol. In Figure 5, we describe the concrete two-party OT protocol in a purely functional manner, making explicit local state shared between the various stages of each party. For example, step_1 outputs the sender's local state st_s, later used by step_3.

```
op step1 (m:(msg * msg) array) (r:int array * G) =
  let (c,hkey) = r in
  let st_s = (m,g^c,hkey) in
  let m1 = (hkey,g^c) in
  (st_s,m1).

op step2 (b:bool array) (r:G array) m1 =
  let (hkey,gc) = m1 in
  let st_c = (b,hkey,r) in
  let m2 = if b then gc / g^r else g^r in
  (st_c,m2).

op step3 st_s (r:G) m2 =
  let (m,gc,hkey) = st_s in
  let e = (H(hkey,m2^r) ⊕ m0,
           H(hkey,(gc/m2)^r) ⊕ m1) in
  let m3 = (g^r,e) in
  m3.

op finalize st_c m3 =
  let (b,hkey,x) = st_c in
  let (gr,e) = m3 in
  let res = H(hkey,gr^x) ⊕ e_b in
  res.
```

```
clone OTProt as SomeOT with
  type rand1 = G array,
  type rand2 = (G array * G) * G,
  op prot (b:input1) (r_c:rand1) (m:input2) (r_s:rand2) =
    let (st_s,m1) = step1 m (fst r_s) in
    let (st_c,m2) = step2 b r_c m1 in
    let m3 = step3 st_s (snd r_s) m2 in
    let res = finalize st_c m3 in
    let conv = (m1,m2,m3) in
    (conv,(res,())).
```

Figure 5: A 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 2.2 (OT-security of SomeOT). *For all $i \in \{1, 2\}$ and OT_i adversary $\mathcal{A}_i$ of type $\mathsf{Adv}_i^{\mathsf{OT}}$ 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*

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

In the proof of Theorem 2.2, both reductions first go to n-ary versions of the DDH and Entropy-Smoothing hypotheses before reducing these further to standard assumptions using the generic hybrid argument lemma.

Extraction and Micro Benchmarks. Our verified implementation of Yao's protocol is obtained via the extraction mechanism included in recent versions of EasyCrypt. The only exceptions to this are the low-level operations left abstract in the formalisation, namely: i. abstract core libraries for randomness generation, the cyclic group algebraic structure, a PRF relying on AES and the entropy-smoothing hash of SomeOT. These are implemented using the CryptoKit library;[9] and ii. a wrapper that handles parameter passing (circuits, messages and input/output) and calls the extracted SFE code. 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.

We conclude this section with microbenchmarking results focusing only on the extracted OCaml implementation. Our results show that, whilst being slower than (unverified) optimized implementations of SFE that use similar cryptographic techniques [11, 35, 40, 56], the performance of the extracted program is compatible with real-world deployment, providing evidence that the (unavoidable) overhead implied by our code extraction approach is not prohibitive. The overhead of our solution is not intrinsic to the verification and extraction methodology. Indeed, the more modern (unverified) implementations showing significant improvements rely on either cryptographic optimizations [35] or on new SFE protocols [56]. Moreover, although these changes have implications on the security proofs, these can be addressed using the same techniques presented here to obtain a verified implementation that benefits from these recent cryptographic advances.

Table 1: Timings (ms): P1 and P2 denote the parties, S1 and S2 the SFE protocol stage; TTime denotes total time, OT the time for OT computation, GT the garbling time and ET the evaluation time.

Circuit	NGates	TTime	P2 S1 GT	P2 S1 OT	P1 S1 OT	P2 S2 OT	P1 S2 OT	P1 S2 ET
COMP32	301	272	1	54	53	109	53	0.3
ADD32	408	275	1	55	54	112	53	0.5
ADD64	824	545	3	109	107	217	106	1
MUL32	12438	329	44	98	54	111	54	10
AES	33744	1233	118	345	216	435	215	24
SHA1	106761	2638	349	780	431	868	430	77

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 serialization and communication overheads nor do they include the time to sample the randomness, all of which we account for in Section 4.

Our measurements are conducted over circuits made publicly available by the cryptography group at the University of Bristol,[10] 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 formalization. We run our experiments on an x86-64 Intel Core i5 clocked at 2.4 GHz with 256KB L2 cache per core. The extracted code and parser are compiled with ocamlopt version 4.02.3. 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.

A subset of our results is 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 for unverified implementations of the same protocol such

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

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

as, say, that in [40] 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 our approach generates. Furthermore, additional optimization effort can lead to significant performance gains, e.g., by resorting to hardware support for low-level cryptographic implementations as in [11, 56], or implementing garbled-circuit optimizations such as those allowed by XOR gates or component based garbled-circuits [35, 42]. Indeed, we do not aim or claim to produce the fastest implementation of Yao's protocol, but simply to demonstrate that the new formal verification techniques that we introduce open the way to verifying a whole new class of provable security arguments, where modularity, abstraction, and composition (e.g., hybrid arguments) mechanisms are essential to dealing with scale and complexity.

3 CERTIFIED BOOLEAN CIRCUIT COMPILER

In this section we describe a new certified compiler called CircGen that can convert (a large subset of) C programs into Boolean circuit descriptions. This is a self-contained, standalone tool that can be used in arbitrary contexts where computation needs to be specified as Boolean circuits. By a certified compiler we mean a compiler that is coupled with a formal proof asserting that the semantics of the source program is preserved through the compilation process. In other words, whenever the source program exhibits a well-defined behavior on some input, the behavior of the target program will match it. The tool is based on the CompCert certified compiler [43], ensuring the adoption of a widely accepted formal semantics for the C language.

Relevant CompCert features. CompCert is in fact a family of compilers for implementations of the C programming language for various architectures (PowerPC, ARM, x86). It is moderately optimizing, sometimes compared to GCC at optimization level 1 or 2. It is formally verified: the semantics of the programming languages involved in the compiler (in particular C and the assembly languages) are formally specified; and correctness theorems are proved. The correctness of a compiler is stated as a behavior inclusion property: each possible behavior of the target program is a possible behavior of the source program. A behavior of a program is a (maybe infinite) sequence of events that describe the interactions of the program with its environment. For the current prototype we have adapted the 2.5 distribution of CompCert.[11]

3.1 CircGen architecture

The meaning of a C program is normally specified as a set of traces that captures the interactions with the *execution environment* triggered by the execution of the program (I/O of data, calls to the operating system, ...). In order to match it with the behavior of evaluating a Boolean circuit, we need to be somewhat more strict on the semantics of programs and, consequently, on the class of programs deemed acceptable to be translated by the tool. The overarching assumption underlying CircGen is that the input C program is coupled with a specification of two memory regions (an input region and an output region) and that we are able to identify the

[11] http://compcert.inria.fr/

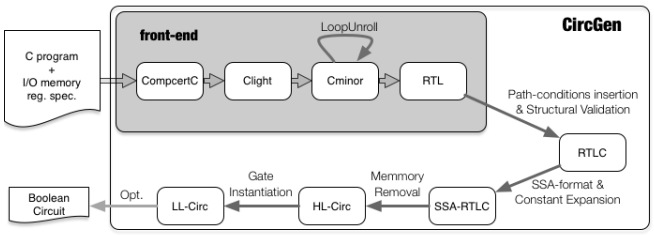

Figure 6: Architecture of the certified compiler CircGen

meaning of the C program with a Boolean function acting on those memory regions. The tool should then generate a circuit implementing that specific Boolean function, thus capturing the meaning of the source program.

The CircGen architecture is shown in Figure 6. It is split in two components: i. the front-end, whose task is to convert the source program into an intermediate language that has been tailored to already admit a Boolean circuit *interpretation*; and ii. the back-end, that formalises the intended Boolean circuit interpretation of programs, and carries out the (certified) transformations up to an explicit Boolean circuit. In other words, the front-end will reject programs for which it cannot determine that there exists a valid Boolean circuit interpretation; whereas the back-end will make explicit the Boolean circuit interpretation.

The front-end follows closely the first few compilation passes of CompCert, adapting and extending it to meet the specific requirements imposed by our domain. We develop and verify the back-end from scratch.

3.2 C features/usage restrictions

The driving goal in our design is to let the programmer use most of the C language constructs (memory, functions, control structures such as loops and conditional branches, ...) that are convenient to program complex, large circuits. However, in our presentation we will use a very simple running example. The circuit that compares its two inputs to decide which is the largest can be described by the C program shown in Figure 7 (function millionaires). In order to be correctly handled by the compiler, the program specifying the circuit must be wrapped in a main function that declares what are the inputs and the outputs of the circuit. The declaration of inputs and outputs also allows us to state the correctness of the compiler; intuitively, the trace of this program will include the incoming inputs and the outgoing outputs of the produced circuit. The dedicated header file provides convenient macros. Note that boolean circuits produced by our compiler are "party-agnostic". In the workflow, one specifies which input bits correspond to each party only when providing the circuit to underlying frameworks.

Assumptions on Input Programs. We start by enumerating natural high-level restrictions imposed on input programs: i. the program must consist of a single compilation unit; ii. input and output memory regions must be properly identified; iii. any feature that produces observable events under CompCert's semantics is disallowed (e.g. volatile memory accesses; external calls; inline assembly;

```
#include "circgen.h"

int millionaires(int x, int y) {
    if (x < y) return 1;
    else return 0;
}
int main(void) {
    static int a, b, result;
    AddInput(a);
    AddInput(b);
    BeginCirc();
    result = millionaires(a, b);
    EndCirc();
    AddOutput(result);
    return 0;
}
```

Figure 7: Example C program

Table 2: CircGen features/restrictions

Features	
Recursion	×
GOTO's	×
Dynamic memory	×
Static memory	✓
Loop unfolding	✓
Integer computations	✓
Non-integer computation	×

etc); and iv. so far, only integral types are allowed. A summary of the most relevant features and restrictions of CircGen can be found in Table 2. The fragment of C that we support is aligned with similar tools. Most of the limitations at this level are inherent to the problem of describing programs as (relatively small) Boolean circuits.

Functions. The source C program can be structured in different functions, but the tool will force all function calls to be inlined (independently of the presence of the inline keyword in function headers). As a consequence, we exclude any form of recursion in source programs (either direct or indirect). In practical terms, we adapt the function inlining pass of CompCert, which refuses to inline any kind of recursive function (each time it inlines a function f, it removes f from the context). Therefore, this restriction amounts to enforcing that, after inlining, the program entry point does not include function calls.

Control structure and termination. In order to extract a Boolean function from a C program we need to enforce termination on all possible inputs. Since recursion has already been excluded, possible non-terminating behavior can only be caused by C loop statements or unstructured use of gotos. For loops, we consider a specific compiler pass that attempts to remove them by a suitable number of unfoldings (detailed below). We choose not to support gotos in the tool; in particular, any attempt to build a loop using gotos will cause the program to be rejected.

Variables and memory. During the conversion of C programs into Boolean circuits, variables need to be converted into wires connecting gates. Specifically, each live range of a variable gives rise to a set of wires (with the number of wires matching the number of bits stored in the variable)—*writing* to a variable means that the wires corresponding to that variable originate in the output ports of some gate that produces the value to be stored; and *reading* from a variable means that the associated wires are connected to the input wires of some gate that is consuming the variable value to perform an operation. Memory accesses to fixed locations behave (in this respect) similarly to variables: a *store* and *load* to a fixed location correspond to a *write* and *read* of a specific variable, respectively.

```
main() {
        x18 = volatile load int8u(&__circgen_io)
        int8u[a] = x18
        x18 = volatile load int8u(&__circgen_io)
        int8u[a + 1] = x18
        . . .
        int8u[b + 3] = x335
        x9 = __circgen_fence()
–:      x7 = int32[a]
        x8 = int32[b]
        if (x7 <s x8) goto 14 else goto 13
14:     x332 = 1; goto 12 13: x332 = 0
12:     int32[result] = x332
        x6 = __circgen_fence()
        x329 = int8u[result]
        _ = volatile store int8u(&__circgen_io, x329)
        . . .
        x323 = int8u[result + 3]
        _ = volatile store int8u(&__circgen_io, x323)
        x2 = 0
        return x2
}
```

Figure 8: Front-end RTL output

Memory accesses can, however, be subtler when the location of the access (address) depends on additional data, as in the case of indexed memory accesses (e.g., array operations). When reading from such a composite address, one is led to a selection of specific wires from a much larger pool of wires, which amounts to a multiplexing operation in Boolean circuit jargon. Conversely, storing to an indexed address is akin to a demultiplexer gate. The problem lies in the fact that these (meta-)gates are very expensive if built from elementary logic operations, leading to exponential circuit sizes on the number of selection bits. This clearly makes unrestricted (32-bit) indices out of reach, leading to the necessity of adopting a strategy to bound them to reasonable limits. We therefore exclude any form of dynamic memory allocation (both in the heap and in the stack) and consider only programs that i. allocate memory statically; and ii. for which memory usage is determined at compile-time.

3.3 Front-end compiler passes

The front-end of CompCert, for the most part unchanged, is used to parse, unroll loops, inline functions, and perform general optimizations at the Register Transfer Language (RTL) level (constant propagation, common subexpression elimination, and redundancy elimination). The RTL intermediate representation produced by the front-end for the example input program of Figure 7 is given in Figure 8. We can observe that, because of inlining, only the main function is left. It starts with a sequence of volatile loads that take the circuit inputs from the environment into the designated global variables, one octet at a time. Then, following the code of the circuit (between the lines marked '–' and '12', in red on the Figure), comes a final sequence of volatile stores that sends the circuit outputs to the environment, one octet at a time. These three sections of the RTL program are delimited by dummy external calls (to __circgen_fence); they block any optimization across these boundaries that

could prevent the correct recognition of inputs and outputs in the next compilation pass.

Loop unroll. The loop-elimination pass is split into two elementary transformations: i. one that unrolls the loop by an arbitrary number of iterations, but leaves the loop unchanged at the end; and ii. one that establishes that the loop kept after all the unrollings is indeed redundant (i.e., that it is unreachable). By doing this, we simplify significantly the semantic preservation proof, since the first transformation follows directly from the operational semantics of loops and is always sound, independently of the number of unrolls. The second transformation can be seen as a specific instance of dead-code elimination.

We implement the first transformation as a new compiler pass in CompCert and prove its semantic preservation theorem. This pass is performed at the Cminor intermediate language since it has a unified treatment for all C loop constructors, but still retains enough structure in the loops to support a simple semantic preservation proof. The number of unfolds for each loop is received by the tool as external advice. For the second transformation we rely on the pre-existing dead code elimination pass of CompCert to remove the remaining loops that are kept after the unfolds. Note that dead code elimination is performed after loop unrolling, function inlining and constant propagation passes, which ensures that loops with simple control structure are successfully eliminated as dead-code (provided that sufficient large unroll estimates are given at the loop unroll pass; otherwise compilation will fail). To make this transformation more effective, we had to slightly improve the abstract domain used in CompCert's value analysis to improve the accuracy of the constant-propagation pass.

3.4 Back-End compiler passes

RTL Circuits. The first intermediate language of the back-end is a variant of RTL that we have called RTLC (RTL Circuits). The language is itself very similar to RTL, with the exception that the control-flow is enforced by conditional execution. Specifically, each conditional test is assigned to a propositional variable. These propositional variables are then used to build path-condition formulas that are assigned to each instruction; the execution of each instruction is conditioned on the validity of a path condition that encodes the combination of branches that can possibly lead to it. Note that RTLC retains all the memory accesses from RTL, that is, writes and reads to and from global variables and stack data.

The semantics of RTLC is sequential (each and every instruction is evaluated following the order of appearance in the program), but the execution of an instruction is guarded by the corresponding path condition. We have adopted Ordered-Reduced Binary Decision Trees [23] as canonical representatives of path-conditions, where nodes are tagged with propositional variables (branching points) and leaves are Boolean values. Figure 9 (left) shows the test program after the path-condition computation pass. Path-conditions are the guards shown at the end of each line (propositional variables are denoted by their index).

From RTL to RTLC. The translation from RTL to RTLC amounts essentially to the computation of *path-conditions* for every instruction in the program. This computation is part of the RTL structural

```
INPUTS: a[0..3] b[0..3]
OUTPUTS: result[0..3]

x6 = int32[a]            | T
x7 = int32[b]            | T
test b14 = x6 <s x7      | T
x16 = 1                  | b14
x16 = 0                  | ¬b14
int32[result] = x16      | T
```

```
INPUTS: a[0..3] b[0..3]
OUTPUTS: result[0..3]

0: int32[a]              |
1: int32[b]              |
2: w0 <s w1              | T
3: 1                     |
4: 0                     |
5: (w2?w3:w4)            |
6: int32[result] := w5   | T
```

Figure 9: Guarded instructions instead of control-flow (left) and corresponding SSA conversion (right).

validation that occurs as the final pass of the compiler front-end component. This validation ensures that a Boolean circuit interpretation can be assigned to the RTL program, making it ready to be processed by the CircGen back-end. This is accomplished by a traversal of the control-flow graph in topological order that: i. identifies boundaries of the three segments of the program (sequence of inputs, body, and sequence of outputs); ii. checks that the body only includes forward jumps; and iii. checks that it does not execute any unsupported instruction (function call, volatile memory access, etc.). Note that check ii. ensures that the control-flow graph is acyclic, which in particular validates that every loop was discharged by the redundancy elimination pass.

Path conditions for the instructions of the body are also constructed during this traversal by applying the following rules: i. initially, all instructions have the $\bot$ path condition (unreachable instruction), except for the first instruction of the body that is assigned the $\top$ path condition (unconditionally executed); ii. when a non-branching instruction is visited, its path condition is propagated to its successor (joining it with any previously computed path condition for that program point); and iii. when a branching instruction (condition test) is visited, the corresponding propositional variable (resp. its negation) is added to the path condition which gets propagated to the *then* successor (resp. the *else* successor).

Constant Expansion. The guarded execution model of RTLC is particularly well-suited to perform an optimization with significant impact on the size of the resulting circuit for certain classes of C programs: for some operations it is possible to determine that their arguments will be constant once the execution path is fixed. For those operations we expand the associated instruction into multiple instances with constant arguments, and use the associated path-conditions to differentiate between the paths. In our implementation we have instrumented this optimization exclusively for memory operations; the impact for algorithms that rely on array indexing (e.g., sorting) is dramatic, as we show in the micro-benchmarking that we present at the end of the section.

Static Single Assignment (SSA). Presenting RTLC programs in Static Single Assignment form allows for a neat correspondence between program variables (register variables in RTLC) and their intended view as wire buses in a Boolean circuit. More importantly, explicit information on the discrimination conditions for variable aggregation performed at the control-flow join points (ϕ-nodes) is easily accessible by looking at path-conditions from the incoming nodes. Indeed, during the translation into SSA, we add a rich variant of ϕ-nodes describing not only the variables that are merged in

```
1 : 64-wire INPUT
2 : 0-bit GLOBAL = []
3 : 32-bit GLOBAL = [0,0,0,0]
4 : 32-bit GLOBAL = [0,0,0,0]
5 : 32-bit GLOBAL = [0,0,0,0]
6 : selk(32,32,0)[32] ← [(1,0..31)]
7 : id(32)[32] ← [(6,0..31)]
8 : selk(32,32,0)[32] ← [(1,32..63)]
9 : id(32)[32] ← [(8,0..31)]
10: id(32)[32] ← [(9,0..31)]
11: id(32)[32] ← [(7,0..31)] 12 : slt32[1] ← [(11,0..31);(10,0..31)]
13: const32(0)[32] ← []
14: const32(1)[32] ← []
15: guard(¬12)[1] ← [(12,0)]
16: barrierN(32)[32] ← [(15,0);(13,0..31)]
17: guard(12)[1] ← [(12,0)]
18: barrierN(32)[32] ← [(17,0);(14,0..31)]
19: xorN(32)[32] ← [(16,0..31);(18,0..31)]
20: updk(32,32,0)[32] ← [TT;(19,0..31);(5,0..31)]
OUTPUT = [(20,0..31)]
```

Figure 10: High-level circuit for the example program.

the node, but also the conditions that discriminate between the different incoming paths.

At this stage, we also take the opportunity to remove most of the path-condition guards on instructions, replacing them with an implicit $\top$ path-condition, but keeping those whose presence is required by the semantic preservation result (namely, the guards on tests and memory writes). This simplification is justified by: i. the fact that SSA-form ensures enabled instructions never destroy previously computed data; and ii. the fact that ϕ-nodes already have explicit information on incoming condition guards. Figure 9 (right) illustrates the effect of the SSA pass on the running example. The SSA property is enforced by the program syntax: registers are named according to the line at which they are defined (e.g., w2 holds the value resulting from the evaluation of line 2).

High-Level Circuits. We call HLcirc a language describing Boolean circuits with complex gates. This is the next intermediate language used by the CircGen back-end. Each of these gates has a specified number of input and output wires, and behaves in accordance with a predefined Boolean function $\mathsf{eval}_G : 2^{\mathsf{in}} \rightarrow 2^{\mathsf{out}}$. Circuits are specified by a sequence (array) of *wire-buses* (sets of wires) that are fed into and collected from these complex gates. Specifically, the circuit description starts with a (nonempty) set of input wire-buses that collectively constitute the input wires of the circuit. This is followed by a topological description of the circuit, describing the gates and how they connect to each other: each line in the program specifies a *wire-bus* matching the out-arity of the gate. Inputs to the gate are specified by *connectors* that select which wires from the incoming bus are plugged to the gate's input. An obvious topological constraint is imposed: the connector for a gate can only refer to wires appearing earlier in the circuit. Finally, we have a description of the outputs of the circuit (again, described by a connector). Figure 10 presents a circuit description for the example program.

Handling of RTLC Memory Accesses. The main abstraction gap between SSA-RTLC and HLcirc is the use of memory. Recall that RTLC retains memory operations to access/update global variables and data stored on the stack. Hence, the translation into high-level circuits must keep track, at each program point, of which wires store the data for the relevant memory regions. To this end, we treat every memory region as a pool of wires, initialized in accordance with the original C program (lines 2-5 in Figure 10, which hold the initial data of stack, a, b and result respectively). These initial pools are possibly updated by input declarations (e.g. declaring a as an input redirects its wires to some of the wires in entry 1 – the input wires of the circuit). Read and store operations consist in either reading from or replacing (some of the) wires in the bus. Concretely, we consider four distinct gates to handle memory read/write operations, all parameterized by the bit-width of the elements and the memory region size:

- selk-w-n-k: takes n data wires and outputs the wires for a w-bit word corresponding to the k-th element (k is a constant).
- sel-w-n: takes n data wires and $\log(n/w)$ index wires and outputs the wires for a w-bit word corresponding to the indexed element.
- updk-w-n-k: takes a condition guard (1 bit), n data wires and w wires holding the value to store; it outputs the resulting n data wires (updated at position k).
- upd-w-n: takes a condition guard, n data wires, w value wires, and $\log(n/w)$ index wires; it outputs the updated n data wires.

Note that updates are always guarded by a guard condition. In fact, for memory writes, we retain the guarded sequential execution semantics of RTLC. By lazily keeping guards at the update gates we are able to later remove them with very small overhead (due to constant propagation) and hence obtain much better generated circuits (in terms of gate counts). Moreover, observe the distinction between arbitrary and constant indexed variants of both operations—while, in the former, the index is provided as an input to the gate, in the latter the index is a (constant) parameter. The reason for the distinction is the huge difference between the gate-complexity of those variants, since constant-index operations amount essentially to a simple rewiring, while the arbitrary indices impose heavy decoding and multiplexing operations. This is indeed the main motivation for the constant-expansion pass mentioned earlier: memory accesses constitute the best example where the impact of unfolding constant alternatives can be significant.

Reg-to-Wire Mapping and ϕ-node Placement. To finalize the translation from SSA-RTLC to HLcirc it now suffices to associate to each RTLC variable the correct number of wires and to insert explicit code to resolve ϕ-nodes. This transformation is justified via the facts that i. the SSA form ensures no cyclic dependencies in the wire definitions; and ii. that the explicit guards provided with ϕ-nodes naturally lead to a w-bit multiplexer (w being the bit-width of joined variables). This is clearly noted in lines 15–19 of Figure 10.

Circuit generation. From a high-level circuit, the CircGen back-end generates a Boolean circuit by obtaining instantiations of the high-level complex gates used in the HLcirc language from an external oracle, and expanding the entire circuit into Boolean gates. This external oracle is part of the trusted base of CircGen. If this is constructed using formally verified instantiations—for example one can have a formally verified library of Boolean circuits for all C native operations— then our semantic preservation theorem states that the generated circuit is correct with respect to the input C program. In our implementation, the high-level gate instantiation oracle produces optimized gates, tailored for multiparty computation applications similar to those used by CBMC-GC, and which we assume to be correct down to extensive testing. Formally verifying

Table 3: CBMC-GC/CircGen/Optim. CircGen: Gate Counts

	CBMC-GC		CircGen		CircGen Opt.	
Computation	Non XOR	Total	Non XOR	Total	Non XOR	Total
arith100	16'143	46'215	16'952	63'005	12'657	43'361
arith1000	160'269	465'470	166'080	616'678	126'709	431'413
arith2000	319'584	936'754	332'257	1'231'845	253'996	863'103
arith3000	479'463	1'442'320	497'014	1'842'778	381'382	1'294'251
hamming160	386	1'610	2'616	10'311	650	3'086
hamming320	784	3'260	5'261	20'801	1'355	6'316
hamming800	1'997	8'248	13'196	52'271	3'470	16'006
hamming1600	5'494	22'796	26'421	104'721	6'995	32'156
median11	10'560	17'850	3'309	14'052	2'880	12'674
median21	40'320	67'050	11'429	49'852	10'560	46'914
median31	89'280	147'600	24'424	107'627	23'040	102'754
median41	902'923	1'100'674	42'294	187'377	40'320	180'194
median51	3'520'577	3'871'968	65'039	289'102	62'400	279'234
median61	7'410'852	7'994'102	92'659	412'802	89'280	399'874
matrix3x3	32'868	85'986	28'494	86'898	27'369	79'310
matrix5x5	148'650	398'750	131'900	402'778	127'225	369'202
matrix8x8	3'641'472	7'286'912	540'224	1'650'883	522'304	1'516'930
aes128-opt	6'400	30'828	90'834	387'042	6'400	31'338
aes128-sbox	504'000	719'050	164'179	664'871	50'800	310'554
aes128-tab	865'152	1'261'780	168'069	1'033'945	50'800	490'586
sha256	28'571	114'169	42'677	201'626	25'667	116'181

the implementations of these gates can be done, e.g., by using the approach in [58]. We leave this for future work.

Unverified optimizations. During the gate expansion, we have implemented some straightforward circuit minimization techniques, such as memoization to reuse previously computed gates and the removal of entries not contributing to the output. These global optimizations are (for now) unverified, and so we report benchmarking results both when they are turned on and off. When we refer to *optimized* CircGen we mean that these optimizations are turned on, and so the semantics preservation proof does not cover the results. When we refer simply to CircGen we mean the semantics-preserving certified tool that excludes these post-processing optimizations.

3.5 Micro Benchmarks

In this section we give a detailed three-way comparison, in terms of gate count in the output circuit, of both our optimized (partly unverified) and verified CircGen and the latest version of CBMC-GC[12] (v0.9.3). The gate counts for various micro-benchmarks are given in Table 3. An important caveat should be highlighted at this point. In collecting results for CBMC-GC, we have truncated its execution time to be comparable (or at least not too much higher than that of CircGen).[13] It is possible that, by allowing the tool to run for more time, it would have produced better results. Therefore, our claim here is not that we have a better tool overall, but that the optimized version of CircGen is a competitive alternative to CBMC-GC. The exception are applications where the computation heavily relies on array accesses. As can be seen in the table, the constant expansion optimization that we introduced for static array access optimization allows us to obtain very significant reductions in gate counts, even in the verified version of CircGen, which we do not believe could be resolved via automatic optimization by CBMC-GC: this is because these optimizations heavily depend on the high-level semantics of the program.

[12] http://forsyte.at/software/cbmc-gc/

[13] All data was collected with a timeout of 600.

We give counts for both the total number of gates and the total number of non-XOR gates (AND or OR gates). The latter can be significantly more costly to evaluate in some protocols than XOR gates, for which very effective optimizations exist. The chosen benchmarks include examples provided in the CBMC-GC distribution, namely those for arithmetic computation of different complexities, Hamming distance for strings of different lengths, median computation via sorting and matrix multiplication (we believe that these examples were used to collect the results in [34]), Additionally we include an implementation of the SHA-256 compression function, taken from the NaCl library,[14] and three different implementations of AES128: **aes128-tab32** corresponds to the public-domain optimized table-based implementation put forth by Rijmen, Bosselaers and Barreto.[15] **aes128-sbox** corresponds to the tabled implementation of AES included in the *Tiny AES in C* library,[16] which, unlike the previous implementation, stores tables using 8-bit rather than 32-bit words; this greatly reduces the book-keeping required to extract values from tables. **aes128-opt** corresponds to an optimized version of aes128-sbox which we developed by modifying aes128-box to make table accesses more Boolean-circuit-friendly, taking advantage of our knowledge of the Boolean circuits used to instantiate native C operators by the back-end, as well as the global cleanup optimizations. This allows us to obtain a relatively efficient circuit implementation from both CBMC-GC and optimized CircGen.

The verified version of CircGen is surprisingly close to the optimized version in all circuits except those corresponding to the Hamming distance and the optimized AES implementation we described above (aes128-opt), for which global, circuit-wide, optimizations give the greatest benefit. A comparison of optimized CircGen with CBMC-GC shows that the two are relatively close for arithmetic operations, CMBC-GC is better in Hamming distance computations, and our tool is better in all programs that use arrays heavily, including the vanilla versions of tabled AES implementations. The global optimization passes are the subject of ongoing work. We do not envision any conceptual difficulty in verifying them, but they do imply reasonable effort to express cross-gate optimizations such as memoization and simplification. Indeed, early experiments reveal that these passes do exacerbate the memory usage of the compiler. The means that we likely will not be able to rely on the data structures made available by CompCert's infrastructure as we do in other passes (specifically, for Maps).

4 SFE SOFTWARE STACK EVALUATION

In this section we present a performance evaluation of the entire SFE software stack based on FRESCO. The FRESCO framework is able to read circuit descriptions in the format produced by CircGen. We thus use the Boolean circuits generated in the micro-benchmarks reported in the previous section to feed two protocol suites supported by this framework.[17] The results are given in Table 4.

The first protocol we test is the verified implementation of Yao's protocol described in Section 2, which has been integrated into

[14] https://nacl.cr.yp.to

[15] Google "rijndael-alg-fst.c"

[16] https://github.com/kokke/tiny-AES128-C

[17] For CBMC-GC outputs we implement a circuit translator that preserves gate counts modulo the introduction of a small number of output gates, which are required by the FRESCO input format.

FRESCO as a new protocol suite (shown in the table as Yao). The second is the Tiny Tables protocol of [29], which is provided in the vanilla distribution of FRESCO; this protocol operates in the preprocessing model, and includes XOR-specific optimizations. An interesting feature of FRESCO is the ability to run the same circuit transparently in either protocol, simply by changing the configured suite. The times shown are the longest execution time for a party participating in the protocol, using the host-local communications infrastructure that is used for testing the FRESCO framework. The linear evaluation time of our verified implementation of Yao's garbled circuit protocol verified implementation is visible in the data. The amortized execution time per gate is just under 100 μs (this ratio is not shown in the table; it is essentially a constant for all circuits). For the Tiny Tables protocol we present the online computation time (TT onl) and the amortized execution time per gate (AT pg). Here variations are caused by the optimizations that make the evaluation on non-XOR gates less costly. To make this evident, we also include in the table the ratio between the number of non-XOR gates and the total number of gates (¬XOR). Indeed, in addition to faster overall execution times due to the preprocessing trade-offs allowed by this protocol, one can see that for circuits with a lower percentage of non-XOR gates the amortized execution time per gate drops to as little as 40 μs per gate.

We stress that the goal here is *not* to compare the speed of Yao's protocol with Tiny Tables: this would be meaningless not only because these protocols offer incomparable security guarantees, but also because the two implementations have significantly different characteristics. Indeed, the fact that FRESCO operates over Java has obvious performance costs. These are somewhat mitigated for our verified implementation of Yao's protocol, which is running natively. However, this is not the case for the pre-existing Tiny Tables implementation, and so it is most likely that even faster execution times could be achieved for the same circuits in other MPC frameworks. Our true goal by presenting these results is to demonstrate integration of the software artifacts that we have developed into a pre-existing open-source framework, and to illustrate the relative benefits of the verified and optimized Boolean circuits produced by our compiler.

5 RELATED WORK

There have been significant advances towards the development of computer-aided tools for cryptography. These tools fall into two loosely related categories. The first category covers a broad spectrum of high-assurance tools, which use formal methods to deliver strong correctness or security guarantees on models or (more rarely) on implementations. The second category comprises many cryptographic engineering tools, whose goal is to facilitate the development and rapid deployment of high-speed, high-quality software. We review some of the main tools from both families. For the sake of focus, we limit our review to prior work that either delivers verified security proofs in the computational model, targets verified implementations, or is directly relevant to secure multi-party computation. We refer the reader to [20] for a more extensive account of the use of formal methods in (symbolic and computational) cryptography, and to [14, 36] for motivations on computer-aided cryptographic proofs.

5.1 High-assurance cryptography

General-purpose tools. CryptoVerif [19] was among the first tools to support cryptographic security proofs in the computational model and it has been used for verifying primitives as well as protocols. More recently, Cadé and Blanchet [24] have complemented the work on CryptoVerif with a mechanism to generate functional code from CryptoVerif models and use it to generate a verified implementation of SSH.

Swamy et al. [55] build a type-based approach for reasoning about programs written in the typed functional programming language F*. Bhargavan et al. [18] subsequently use F* to develop high-assurance implementations of TLS. Rastogi, Swamy and Hicks [54] also use F* as a host language for embedding Wysteria, a domain-specific language for multi-party computation.

Appel [5] uses VST (Verified Software Toolchain) [4] to prove the functional correctness of a machine-level implementation of SHA-256. In a companion effort, Beringer et al. [17] connect VST with FCF (Foundational Cryptographic Framework) of Petcher and Morrisett [51], in order to provide a machine-checked proof of reductionist security for a realistic implementation of HMAC.

High-assurance MPC. There have been many works that develop or apply formal methods for secure multi-party computation.

Backes et al. [6] develop computationally sound methods for protocols that use secure multi-party computation as a primitive. However, they do not consider verified implementations. Wysteria [53] 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. Dahl and Damgård [26] 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. Pettai and Laud [52] develop a static analysis for proving that Sharemind applications are secure against active adversaries.

Fournet, Keller and Laporte [32] propose a certified compiler from C to quadratic arithmetic circuits (QAP) compatible with the domain of SNARKs. However, the underlying cryptographic system does not come with a verified implementation.

Carmer and Rosulek [25] introduce LiniCrypt, a core language for writing programs that perform linear operations on a finite field and calls to random oracles. They prove that the equivalence of LiniCrypt programs can be decided efficiently, and leverage this result to build a tool for SMT-based synthesis of garbled circuits.

5.2 Engineering of MPC protocols

FRESCO [27] is a Java framework for efficient secure computation. In FRESCO, functions to be securely evaluated are described as circuits; we equip our certified compiler with a back-end that integrates seamlessly into this framework. Run-time systems in FRESCO specify how circuits are evaluated, and are thus highly dependent on the supported protocols for secure computation. In addition to our formally verified implementation of Yao's protocol and the Tiny Tables protocol we use as benchmark, run-time systems in FRESCO include support for several protocols for secure

Table 4: CBMC-GC vs CircGen vs Optimized CircGen: Timings (ms) for two FRESCO suites.

	CBMC-GC				CircGen				CircGen Opt.			
Computation	Yao	¬XOR	TT onl	AT pg	Yao	¬XOR	TT onl	AT pg	Yao	¬XOR	TT onl	AT pg
arith100	5590	35%	3260	0,071	6710	27%	3390	0,054	5196	29%	2549	0,059
hamming1600	5533	24%	1411	0,062	12038	25%	4997	0,048	6252	22%	1649	0,051
median21	6204	60%	6756	0,101	4801	23%	2367	0,047	4540	23%	2057	0,044
matrix3x3	7689	38%	5712	0,066	7700	33%	5297	0,061	7067	35%	4882	0,062
aes-opt	2836	21%	1543	0,050	32935	23%	15997	0,041	2901	20%	1182	0,038
sha256	9943	25%	5157	0,045	17309	21%	7642	0,038	9879	22%	4772	0,041

computation, including the TinyOT protocol [48] for actively secure two-party computation based on Boolean circuits; the actively secure multi-party computation protocol based on arithmetic circuits [16]; and the SPDZ protocol [28, 30] for actively and covertly secure multi-party computation based on arithmetic circuits.

Fairplay, Sharemind and TASTY are MPC frameworks alternative to FRESCO. Fairplay is a system originally developed to support two-party computation [46] and then extended to multiparty computation as FairplayMP [15]: Fairplay implements a two party computation protocol in the manner suggested by Yao; FairplayMP is based on the Beaver-Micali-Rogaway protocol [10]. Sharemind [21] is a secure service platform for data collection and analysis, employing a 3-party additive secret sharing scheme and provably secure protocols in the honest-but-curious security model with no more than one passively corrupted party. TASTY (Tool for Automating Secure Two-partY computations) is a tool suite addressing secure two-party computation in the semi-honest model [37] whose main feature is to allow the compilation and evaluation of functions using both garbled circuits and homomorphic encryption.

Holzer et al. [39] present a compiler that uses the bounded model-checker CBMC to translate ANSI C programs into Boolean circuits. The circuits can be used as inputs to the secure computation framework of Huang et al. [40]. This compiler, CBMC-GC, can also be used as a front-end to our verified implementation of Yao's protocol. However, as we show in Section 4, not only does our approach deliver higher assurance but also, if one activates all optimizations, the circuits generated by our compiler can offer, for some classes of circuits, better performace in comparison with the current version of CBMC-GC (v0.9.3).

Recently, Amy et al. [45] built a compiler that renders Revs [49] programs into space-efficient reversible circuits. The work focused on the usage of such circuits in large quantum computations and was fully developed and verified using F*.

6 CONCLUSIONS AND FUTURE WORK

We have presented a fast and efficient software stack for secure function evaluation. Possible further steps include adapting our approach to recent developments in multi-party and verifiable computation, for instance [50], and to achieve tighter integration between prototyping tools, verification tools, and verified compilers.

Acknowledgments. This work is partially supported by ONR Grants N000141210914 and N000141512750, by Cátedra PT-FLAD em Smart Cities & Smart Governance, by Fundação para a Ciência e a Tecnologia grant FCT-PD/BD/113967/2015, by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project «UID/EEA/50014/2013». The EasyCrypt definitions build on early work by Guillaume Davy. We thank Pierre-Yves Strub for his role in the development of EasyCrypt.

REFERENCES

[1] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. 2013. Charm: a framework for rapidly prototyping cryptosystems. *J. Crypt. Eng.* 3, 2 (2013).

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and Francois Dupressoir. 2013. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *ACM CCS*.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. In *23rd International Conference on Fast Software Encryption (FSE)*. 163–184.

[4] Andrew W. Appel. 2014. *Program Logics - for Certified Compilers*. Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB

[5] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 7:1–7:31.

[6] Michael Backes, Matteo Maffei, and Esfandiar Mohammadi. 2010. Computationally Sound Abstraction and Verification of Secure Multi-Party Computations. In *FSTTCS*.

[7] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII (FOSAD)*. Lecture Notes in Computer Science, Vol. 8604. Springer International Publishing, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

[8] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. 2014. Probabilistic Relational Verification for Cryptographic Implementations. In *POPL*. To appear.

[9] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *CRYPTO*.

[10] D. Beaver, S. Micali, and P. Rogaway. 1990. The Round Complexity of Secure Protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing (STOC '90)*. ACM, New York, NY, USA, 503–513. https://doi.org/10.1145/100216.100287

[11] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient Garbling from a Fixed-Key Blockcipher. In *IEEE S&P*.

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

[13] Mihir Bellare and Silvio Micali. 1989. Non-Interactive Oblivious Transfer and Applications. In *CRYPTO*.

[14] Mihir Bellare and Phillip Rogaway. 2006. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In *EUROCRYPT (Lecture Notes in Computer Science)*, Serge Vaudenay (Ed.), Vol. 4004. Springer, 409–426.

[15] Assaf Ben-David, Noam Nisan, and Benny Pinkas. 2008. FairplayMP: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security*. ACM, 257–266.

[16] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT*. 169–188.

[17] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX*

Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015., Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 207–221. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

[18] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoué. 2016. Implementing and Proving the TLS 1.3 Record Layer. Cryptology ePrint Archive, Report 2016/1178. (2016). http://eprint.iacr.org/2016/1178.

[19] Bruno Blanchet. 2008. A Computationally Sound Mechanized Prover for Security Protocols. *IEEE Trans. Dependable Sec. Comput.* 5, 4 (2008).

[20] Bruno Blanchet. 2012. Security Protocol Verification: Symbolic and Computational Models. In *POST*.

[21] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*.

[22] Sally Browning and Philip Weaver. 2010. Designing Tunable, Verifiable Cryptographic Hardware Using Cryptol. (2010).

[23] Randal E. Bryant. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.* 24, 3 (1992), 293–318. https://doi.org/10.1145/136035.136043

[24] David Cadé and Bruno Blanchet. 2013. Proved Generation of Implementations from Computationally Secure Protocol Specifications. In *POST*.

[25] Brent Carmer and Mike Rosulek. 2016. Linicrypt: A Model for Practical Cryptography. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III (Lecture Notes in Computer Science)*, Matthew Robshaw and Jonathan Katz (Eds.), Vol. 9816. Springer, 416–445. https://doi.org/10.1007/978-3-662-53015-3_15

[26] Morten Dahl and Ivan Damgård. 2014. Universally Composable Symbolic Analysis for Two-Party Protocols Based on Homomorphic Encryption. In *EUROCRYPT*.

[27] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. 2016. Confidential Benchmarking based on Multiparty Computation. In *Financial Cryptography 2016*. In print.

[28] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS*. 1–18.

[29] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. 2016. Gate-scrambling Revisited - or: The TinyTable protocol for 2-Party Secure Computation. *IACR Cryptology ePrint Archive* 2016 (2016), 695. http://eprint.iacr.org/2016/695

[30] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*. 643–662.

[31] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. 2012. SCAPI: The Secure Computation Application Programming Interface. Cryptology ePrint Archive, Report 2012/629. (2012).

[32] Cédric Fournet, Chantal Keller, and Vincent Laporte. 2016. A Certified Compiler for Verifiable Computing. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 268–280. https://doi.org/10.1109/CSF.2016.26

[33] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. 2011. Modular code-based cryptographic verification. In *ACM CCS*.

[34] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. 2014. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science)*, Albert Cohen (Ed.), Vol. 8409. Springer, 244–249. https://doi.org/10.1007/978-3-642-54807-9_15

[35] Adam Groce, Alex Ledger, Alex J. Malozemoff, and Arkady Yerukhimovich. 2016. CompGC: Efficient Offline/Online Semi-honest Two-party Computation. Cryptology ePrint Archive, Report 2016/458. (2016). http://eprint.iacr.org/2016/458.

[36] Shai Halevi. 2005. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptology ePrint Archive* 2005 (2005), 181.

[37] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. 2010. TASTY: tool for automating secure two-party computations. In *ACM Conference on Computer and Communications Security*. ACM, 451–462.

[38] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. 2010. TASTY: Tool for Automating Secure Two-party Computations. In *ACM CCS*.

[39] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. 2012. Secure Two-party Computations in ANSI C. In *ACM CCS*.

[40] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster Secure Two-party Computation Using Garbled Circuits. In *USENIX Security*.

[41] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. 2013. A Systematic Approach to Practically Efficient General Two-Party Secure Function Evaluation Protocols and their Modular Design. *Journal of Computer Security (JCS)* 21, 2 (01 2013), 283–315. https://doi.org/10.3233/JCS-130464 Preliminary version: http://eprint.iacr.org/2010/079.

[42] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP*.

[43] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 42–54. https://doi.org/10.1145/1111037.1111042

[44] Yehuda Lindell and Benny Pinkas. 2009. A Proof of Security of Yao's Protocol for Two-Party Computation. *J. Cryptol.* 22, 2 (April 2009).

[45] Rupak Majumdar and Viktor Kuncak (Eds.). 2017. *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Lecture Notes in Computer Science, Vol. 10427. Springer. https://doi.org/10.1007/978-3-319-63390-9

[46] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - Secure Two-Party Computation System. In *USENIX Security Symposium*, Matt Blaze (Ed.). USENIX, 287–302. https://www.usenix.org/publications/proceedings/?f[0]=im_group_audience%3A173

[47] Moni Naor and Benny Pinkas. 2001. Efficient Oblivious Transfer Protocols. In *SODA*.

[48] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. 2012. A New Approach to Practical Active-Secure Two-Party Computation. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. 681–700. https://doi.org/10.1007/978-3-642-32009-5_40

[49] Alex Parent, Martin Roetteler, and Krysta Marie Svore. 2015. Reversible circuit compilation with space constraints. *CoRR* abs/1510.00377 (2015). http://arxiv.org/abs/1510.00377

[50] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *IEEE S&P*.

[51] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings (Lecture Notes in Computer Science)*, Riccardo Focardi and Andrew C. Myers (Eds.), Vol. 9036. Springer, 53–72. https://doi.org/10.1007/978-3-662-46666-7_4

[52] Martin Pettai and Peeter Laud. 2014. Automatic Proofs of Privacy of Secure Multi-Party Computation Protocols Against Active Adversaries. Cryptology ePrint Archive, Report 2014/240. (2014).

[53] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *IEEE S& P*.

[54] Aseem Rastogi, Nikhil Swamy, and Michael Hicks. 2016. WYS*: A Verified Language Extension for Secure Multi-party Computations. (2016). Manuscript.

[55] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

[56] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2017. *Faster Secure Two-Party Computation in the Single-Execution Setting*. Springer International Publishing, Cham, 399–424. https://doi.org/10.1007/978-3-319-56617-7_14

[57] Andrew C. Yao. 1982. Protocols for secure computations. In *FOCS*.

[58] Cunxi Yu, Walter Brown, Duo Liu, André Rossi, and Maciej Ciesielski. 2016. Formal Verification of Arithmetic Circuits by Function Extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 12 (2016), 2131–2142. https://doi.org/10.1109/TCAD.2016.2547898

A DETAILS OF EASYCRYPT FORMALIZATION

The top-level abstraction in our formalization is a high-level view of two-party protocols, which is later independently refined to derive formalizations of both oblivious transfer and secure function evaluation. We introduce these concepts by focusing on a classic oblivious transfer protocol [13, 47] and discussing its security proof. Its small size and relative simplicity make it a good introductory example to EasyCrypt formalization. We also introduce our general framework for dealing with hybrid arguments in EasyCrypt.

Two-Party Protocols. 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 11. 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 11: Abstract Two-Party Protocol.

The top level abstraction that represents two-party protocols is given in Figure 11. 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.

Simulation-based security. 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 precisely defined leakage about the other party's input.

Formally, we express this security notion using two games (one for each party). We display one of them in Figure 12, in the form of an EasyCrypt *module*. Note that modules are used to model games and experiments, but also schemes, oracles and adversaries.[18]

```
type leak1, leak2.
op ϕ1 : input1 → leak1.
op ϕ2 : input2 → leak2.
type view1 = rand1 * conv.
type view2 = rand2 * conv.

module type Sim = {
  proc sim1(i1: input1, o1: output1, l2: leak2) : view1
  proc sim2(i2: input2, o2: output2, l1: leak1) : view2
}.

module type Simi = {
  proc simi(ii: inputi, oi: outputi, l3−i: leak3−i) : viewi
}.

module type Advi^Prot = {
  proc choose(): input1 * input2
  proc distinguish(v: viewi) : bool
}.

module Sec1(R1: Rand1, R2: Rand2, S: Sim1, 𝒜1: Adv1^Prot) = {
  proc main() : bool = {
    var real, adv, view1, o1, r1, r2, i1, i2;
    (i1,i2) = 𝒜1.choose();
    real ←$ {0,1};
    if (!validInputs i1 i2)
      adv ←$ {0,1};
    else {
      if (real) {
        r1 = R1.gen(ϕ1 i1);
        r2 = R2.gen(ϕ2 i2);
        (conv,_) = prot i1 r1 i2 r2;
        view1 = (r1, conv);
      } else {
        (o1,_) = f i1 i2;
        view1 = S.sim1(i1, o1, ϕ2 i2);
      }
      adv = 𝒜1.distinguish(view1);
    }
    return (adv = real);
  }
}.
```

Figure 12: Security of a two-party protocol protocol.

Modules are composed of a *memory* (a set of global variables, here empty) and a set of *procedures*. Note that procedures in the same module may share state; it is therefore not necessary to explicitly add state to the module signature. In addition, modules can be parameterized by other modules (in which case, we often call them *functors*) whose procedures they can query like *oracles*. Which oracles may be accessed by which procedure is specified using *module types*. A module is said to fulfill a module type if it implements all the procedures declared in that type. Any procedures implemented in addition to those appearing in the module type are not accessible as oracles. For example, even if a module that implements module type Sim is used to instantiate the S parameter of the Sec_1 module, none of the procedures in Sec_1 may call the sim_2 oracle.

Module type $\mathsf{Adv}_i^{\mathsf{Prot}}$ ($i \in \{1, 2\}$) 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

[18] Our formalisation accommodates generic protocols (e.g., oblivious transfer of an arbitrary, albeit polynomial, number of messages) which justifies the technicality of parametrising the randomness generation procedures with public information associated with the protocol inputs.

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 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. We omit the indices representing randomness generators whenever they are clear from the context.

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 $\mathsf{Adv}_i^{\mathsf{Prot}}$ ($i \in \{1, 2\}$), 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.

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 13.

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. This introduces an implicit quantification on the bound n in all results we prove.

```
clone Protocol as OT with
  type input1 = bool array,
  type output1 = msg array,
  type leak1 = int,
  type input2 = (msg * msg) array,
  type output2 = unit,
  type leak2 = int,
  op φ1 (i1: bool array) = length i1,
  op φ2 (i2: (msg * msg) array) = length i2,
  op f (i1: bool array) (i2: (msg * msg) array) = i1 i2 .
  op validInputs(i1: bool array) (i2: (msg * msg) array) =
      0 < length i1 ≤ n_max ∧ length i1 = length i2,
...
```

Figure 13: Instantiating Two-Party Protocols.

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 $\mathsf{Adv}^{\mathsf{Prot}_{i,\Phi}}$, where $\Phi = (\mathsf{length}, \mathsf{length})$, and similarly we write $\mathsf{Adv}_i^{\mathsf{OT}}$ the types for adversaries against the OT instantiation.

Garbling schemes. Garbling schemes [12] (Figure 14) 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 14: 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* [12], 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. [12], 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). We give functional specifications to the garbling algorithms in Figure 15. For clarity, we denote functional folds using stateful for loops.

```
type topo = int * 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 15: 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 $\mathsf{Adv}^{\mathsf{Gb}}$ 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 $\mathsf{Adv}^{\mathsf{Gb}}$, 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 16: Security of garbling schemes.

Following [12], 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 A.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.
The module shown in Figure 17 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 $\mathsf{Adv}^{\mathsf{IND}}$ 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

```
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 17: Indistinguishability-based Security for Garbling Schemes.

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 A.1 (IND-CPA-security implies SIM-CPA-security). *If Φ is efficiently invertible, then for every efficient* SIM-CPA *adversary $\mathcal{A}$ of type* $\mathsf{Adv}^{\mathsf{Gb}}$*, 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 IND-CPA$_{\Phi\mathsf{topo}}$-security of SomeGarble to the DKC-security of the underlying DKC primitive (see [12]). 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 A.2 (SomeGarble is IND-CPA$_{\Phi_{\mathsf{topo}}}$-secure). *For every efficient* IND-CPA *adversary $\mathcal{A}$ of type* $\mathsf{Adv}^{\mathsf{Gb-IND}}$*, we can construct a efficient* DKC *adversary $\mathcal{B}$ such that*

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

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 4) 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 A.1 and A.2, we can conclude with a security theorem for our garbling scheme.

Theorem A.3 (SomeGarble is SIM-CPA$_{\Phi_{\mathsf{topo}}}$-secure). *For every* SIM-CPA *adversary $\mathcal{A}$ that implements* $\mathsf{Adv}^{\mathsf{Gb}}$*, one can construct an efficient simulator* S *and a DKC adversary $\mathcal{B}$ such that*

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

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