TASTY: Tool for Automating Secure Two-partY computations

(Full Version)

Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi,
Thomas Schneider, Immo Wehrenberg
System Security Lab
Ruhr-University Bochum
Germany
{ahmad.sadeghi,thomas.schneider}@trust.rub.de,
{wilko.henecka,stefan.koegl,immo.wehrenberg}@rub.de

ABSTRACT

Secure two-party computation allows two untrusting parties to jointly compute an arbitrary function on their respective private inputs while revealing no information beyond the outcome. Existing cryptographic compilers can automatically generate secure computation protocols from high-level specifications, but are often limited in their use and efficiency of generated protocols as they are based on either garbled circuits or (additively) homomorphic encryption only.

In this paper we present TASTY, a novel tool for automating, i.e., describing, generating, executing, benchmarking, and comparing, efficient secure two-party computation protocols. TASTY is a new compiler that can generate protocols based on homomorphic encryption and efficient garbled circuits as well as combinations of both, which often yields the most efficient protocols available today. The user provides a high-level description of the computations to be performed on encrypted data in a domain-specific language. This is automatically transformed into a protocol. TASTY provides most recent techniques and optimizations for practical secure two-party computation with low online latency. Moreover, it allows to efficiently evaluate circuits generated by the well-known Fairplay compiler.

We use TASTY to compare protocols for secure multiplication based on homomorphic encryption with those based on garbled circuits and highly efficient Karatsuba multiplication. Further, we show how TASTY improves the online latency for securely evaluating the AES functionality by an order of magnitude compared to previous software implementations. TASTY allows to automatically generate efficient secure protocols for many privacy-preserving applications where we consider the use cases for private set intersection and face recognition protocols.

This is the full version of the ACM CCS'10 paper [23].

Categories and Subject Descriptors

D.0 [**Software**]: General

General Terms

Design, Security, Languages, Performance, Measurement

Keywords

Cryptography, secure function evaluation, compiler, garbled circuits, homomorphic encryption

1. INTRODUCTION

The design of efficient secure two-party computation protocols is vital for a variety of security-critical applications with sophisticated privacy and security requirements such as electronic auctions [40], data mining [34], remote diagnostics [7], medical data classification [1], or face recognition [15, 50, 44] to name some.

Modern cryptography provides various tools for secure computation. The concept of two-party Secure Function Evaluation (SFE) was introduced in 1982 by Yao [59]. The idea is to let two mutually mistrusting parties compute an arbitrary function (known by both)[1] on their private inputs without revealing any information about their inputs beyond the function's output. However, the real-world deployment of SFE was believed to be very limited and expensive for a relatively long time. Fortunately, the cost of SFE has been dramatically reduced in the recent years thanks to many algorithmic improvements and automatic tools, as well as faster computing platforms and communication networks.

For several years, two different approaches for secure two-party computation have co-existed. One of them is based on *homomorphic encryption* (HE). Here one party sends its encrypted inputs to the other party, who then computes the desired function under encryption using the homomorphic properties of the cryptosystem, and sends back the encrypted result. Popular examples are the additively homomorphic cryptosystems of Paillier [45] and Damgård-Jurik [12], and the recent fully homomorphic schemes [17, 13, 55]. The other approach is based on *garbled circuits* (GC), introduced by Yao [60], that works as follows: one party (con-

[1]Universal circuits [58, 32, 49] allow to hide the function from one party.

structor) "encrypts" the circuit (using symmetric keys), the other party (evaluator) obliviously obtains the keys corresponding to both parties' inputs and the garbled circuit, and is able to decrypt the corresponding output value. Both approaches have their respective advantages and disadvantages, i.e., GC requires to transfer the garbled circuit (communication complexity is at least linear in the size of the circuit) but allows to pre-compute almost all expensive operations resulting in a low latency of the online phase, whereas most HE schemes require relatively expensive public-key operations in the online phase but can result in a smaller overall communication complexity.

In the recent years several cryptographic compilers and specification languages have been proposed that, after a programmer has manually mapped an existing algorithm to integer arithmetics, automatically compile this into SFE protocols. We will give an overview on such previous works in §1.2. However, such tools are currently restricted to generating protocols based on only one SFE paradigm, i.e., use either garbled circuits (GC) or homomorphic encryption (HE), which often results in protocols with suboptimal efficiency. For instance HE allows efficient addition and multiplication of large values (as confirmed by our implementation results in §5.1.2), whereas GCs are better for non-linear functionalities such as comparison [31]. By combining both approaches, relatively efficient protocols can be obtained when designing privacy-preserving applications, e.g., remote diagnostics [7], classification [1], or face recognition [50].

The main goal of this work is the design and implementation of the first compiler, we call TASTY, that can automatically generate efficient protocols based on homomorphic encryption (HE) and garbled circuits (GC) as well as combinations of both from a high-level description of the protocol.

Finally, we would like to stress that although *fully* homomorphic encryption schemes have emerged recently [17, 13, 55], they are still not efficient enough to be used in practical applications. Nevertheless, they could be integrated into our compiler framework once they are efficient enough.

1.1 Our Contribution and Outline

In this paper, we present the following contributions in the respective sections.

SFE Compiler: We present TASTY, a tool that allows to automatically *generate, benchmark* and *compare the performance* of efficient two-party SFE protocols in the semi-honest model (§4). We show how TASTY is related to, improves over, and can be combined with existing tools for automatic generation of (two-party) SFE protocols (§1.2).

Specification Language: TASTYL, the TASTY input Language, allows to *describe* SFE protocols as *sequence of operations on encrypted data* based on *combinations of Garbled Circuits (GC) and Homomorphic Encryption (HE)*. We review the underlying theoretical framework for such modularly composed SFE protocols [31] in §2. TASTYL is based on the Python programming language and *hides technical cryptographic details* from the programmer (§4.1).

Efficient Building Blocks: TASTY implements *efficient building blocks* for HE and GC which allow to shift most of the complexity into the less time critical setup phase resulting in SFE protocols with a *low-latency online phase* (§4.3). While the implemented techniques have been known before, their combination and implementation in a single package is unique and useful. We show how the combination of these techniques speeds up the online phase for *secure evaluation of AES* (a large circuit with more than $30,000$ gates) compared to the currently fastest software implementation of GCs [48] from 5 s to only 0.5 s, while the total costs for setup plus online phase stay almost the same (§5.2).

Circuit Optimizations: Additionally, TASTY has built-in tools for *on-the-fly generation and minimization of boolean circuits* (§4.3). As new circuit building block we implement *fast multiplication circuits* based on Karatsuba method [28] that are more efficient than textbook multiplication (used in previous SFE tools), already for 20 bit numbers; for multiplication of 128 bit values, it is more efficient by 45% (§5.1.1).

Benchmarking: Using TASTY, we obtain measurements for a detailed *performance comparison of multiplication protocols based on GCs with those based on HE*. Our experiments show that GC-based multiplication has large communication and time complexity in the setup phase, but results in a more efficient online time than HE-based multiplication for small values (§5.1.2). In particular, multiplication of two garbled values with bitlength $\ell \leq 16$ bits requires less online communication and time than the multiplication of two homomorphically encrypted values for short-term security.

Applications: We show that TASTY is a usable and useful tool for describing and automatically generating efficient protocols for several privacy-preserving applications. We implemented set intersection and face recognition (§3).

The paper is concluded with an overview on future work which could be based on the TASTY framework (§6).

1.2 Related Work

While the theoretical foundations of two-party Secure Function Evaluation (SFE) have been laid already in the eighties [59, 60], recent optimizations and prototype implementations show that SFE is ready to be used in practical applications (e.g., [35, 48]). To allow the deployment of SFE in a wide range of privacy-preserving applications it is not only important to maximize the *efficiency* of SFE protocols, but also to make SFE *usable* by automatically generating protocols from high-level descriptions. For this, several frameworks for SFE consisting of languages and corresponding tools have been developed in the last years. We review these proposals briefly in the following.

Existing SFE frameworks can be divided into three classes on different abstraction levels as summarized in Table 1.

Table 1: Abstraction Levels for Automatic Generation of SFE Protocols

Abstraction Level	Primitives
Function Description	I/O, computation
Protocol Description	I/O, enc/dec, computation under encryption
Protocol Implementation	I/O, protocols, messages, cryptographic primitives

Function Description languages allow to specify *what* function should be computed securely. The function is described in a domain-specific high-level programming language which allows programmers to write programs using SFE without any expert knowledge about SFE. Functions described in such languages can then be (formally) analyzed to ensure security of the function (e.g., no information leak to the other party) and are compiled (potentially through lower-level SFE languages) into SFE protocols. Examples are Fairplay's Secure Function Definition Language (SFDL) [37, 3] which can be compiled to boolean circuits (see below), or the Secure Multiparty Computation Language (SMCL) [42] and its Python-based successor PySMCL [41] which allow compilation into arithmetic circuit-based secure multiparty computation (SMPC) protocols such as the Virtual Ideal Functionality Framework (VIFF) [11].

Protocol Description languages allow to specify *how* the SFE protocol is composed as sequence of basic operations on encrypted (or secret-shared data). Examples (described in more detail below) are VIFF [11], the Secure Multiparty Computation language (SMC) [43, 54], Sharemind [5], and the compiler of MacKenzie et al. [36]. These languages allow to specify SFE protocols while abstracting away the details of the underlying cryptographic protocols. The language and compiler we present in this paper also fall into this class. However, in contrast to previous works which were restricted to using homomorphic encryption only, our compiler TASTY allows arbitrary combinations of computations under encryption based on garbled circuits and/or homomorphic encryption for *highly efficient* SFE protocols.

Protocol Implementation languages allow to describe *how exactly* the target SFE protocol is composed as sequence of basic cryptographic protocol building blocks. They reside at the lowest level of the abstraction hierarchy and require a substantial amount of expert knowledge in cryptographic protocol design. For example the L1 language [52] allows to describe secure computation protocols as sequence of basic primitives such as oblivious transfer (OT), homomorphic encryption/decryption, creation and evaluation of garbled circuits, and messages to be exchanged. Qilin [38] is a Java library for rapid prototyping of cryptographic protocols which currently provides common cryptographic protocols (e.g., OT [39] and coin flipping) using cryptographic primitives (e.g., Pedersen Commitment [47] and ElGamal [14]) implemented with elliptic curves.

Next we describe SFE frameworks which are closely related to ours. In contrast to TASTY, the existing SFE frameworks are based on *either* garbled circuits (GC) *or* homomorphic encryption (HE), but not combinations of both.

Garbled Circuits (GC). The most prominent example for automatic generation of SFE protocols is Fairplay [37] which is based on GCs. Fairplay provides a high-level function description language, SFDL, which allows to specify the function to be computed securely, i.e., the inputs and outputs of the involved parties, and how the outputs are to be computed from the inputs. The language resembles a simplified version of a hardware description language, such as Verilog or VHDL[2], and supports types, variables, functions, boolean operators $(\wedge, \vee, \oplus, \dots)$, arithmetic operators $(+, -)$, comparison $(<, \geq, =, \dots)$ and control structures like if-then-else or for-loops with constant range. The Fairplay compiler compiles and optimizes an SFDL program into a boolean circuit which is stored in a file. The circuit can then be evaluated using the Fairplay runtime environment, two Java programs which securely evaluate the circuit using Yao's garbled circuit protocol, communicating over a TCP socket. Fairplay is supplemented by FairplayMP [3], a multi-party version of Fairplay suited for three or more parties with the more powerful SFDL 2 input language (with support for $*, /$ and generic functions) and a corresponding circuit compiler. TASTY can serve as efficient runtime environment for the Fairplay compiler suite, i.e., it allows to read in circuits generated by the FairplayMP compiler from SFDL 2 programs[3] and optimizes these for efficient secure evaluation with state-of-the-art GC evaluation techniques.

[2]Very high speed integrated circuit Hardware Description Language

Homomorphic Encryption (HE). VIFF [11], the Virtual Ideal Functionality Framework, is an open source framework written in Python for specifying secure multi-party computation (SMPC) protocols as a sequence of operations performed on secret-shared (i.e., encrypted) data. While VIFF was mainly designed for secret-sharing based SMPC protocols with three or more parties, it also offers a two-player runtime based on the additively homomorphic Paillier cryptosystem [45]. Using operator overloading, VIFF allows the programmer to express a desired secure computation directly as standard arithmetic without knowing about the used protocol. Indeed, TASTYL, the input language of our compiler, is inspired by the VIFF language, but additionally allows to combine HE with GC-based computations.
In contrast to general-purpose compilers such as Fairplay, VIFF, and TASTY, the compilers described below are built for specific application scenarios, e.g., use specific number representations [36, 5] or require $n \geq 3$ parties [43, 54, 5]:
The compiler of MacKenzie et al. [36] implements secure two-party computations over values which are secret-shared between both parties using $\binom{2}{2}$ secret-sharing over a prime field. The computations are composed as sequence of basic operations on the shared data (e.g., addition or multiplication). The compiler can be used for specific functions such as cryptographic primitives defined over prime fields, e.g., signatures or encryption schemes, where the secret key is shared between both parties.
SMC [43, 54], the Secure Multiparty Computation language, provides a declarative language for describing SMPC based on constraint programming. A program is distributed among the parties in the computation along with an interpreter, each party gives its secret inputs and the interpreter calculates the result. Computations are specified as arithmetic circuits and at least 3 parties are required as the underlying multiplication protocol is based on the BGW protocol [4].
Sharemind [5] allows secure computation over the ring of 32-bit integers for three parties and provides an assembly-like programming language. As this setting is fixed and very specific it allows highly efficient protocols.

2. THEORETICAL BACKGROUND

In this section we summarize the framework for modular design of efficient two-party Secure Function Evaluation (SFE) protocols of [31] on which TASTY is built.

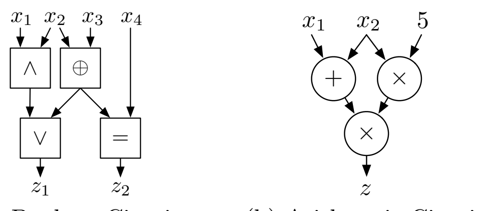
Model. We concentrate on the semi-honest model, where both parties follow the protocol but try to infer additional information from the transcript of messages seen in the protocol. Far from trivial, this model covers many typical practical settings such as protection against insider attacks. Further, designing and evaluating the performance of protocols in the semi-honest model is a first stepping stone to-

[3]FairplayMP's compiler can generate circuits for two parties.

wards protocols with stronger security guarantees. Indeed, most protocols and implementations of protocols for practical privacy-preserving applications focus on the semi-honest model [40, 7, 1, 15, 50, 44]. For a detailed discussion on the semi-honest model and its extensions we refer to [34, 31].

Notation. We call the two semi-honest SFE participants *client* $\mathcal{C}$ and *server* $\mathcal{S}$. This naming choice is influenced by the asymmetry in the SFE protocols, which fits into the client-server model. We stress that, while in most real-life two-party SFE scenarios this client-server relationship in fact exists, we do not limit ourself to this setting.

Function Representations. Given a function f that should be computed securely, the first task during the design of the corresponding SFE protocol is to find a suitable representation for f. Well-established representations which allow efficient SFE protocols are boolean circuits and arithmetic circuits as shown in Fig. 1.[4] The representation determines the size of the function, e.g., multiplication can be expressed as arithmetic circuit with a single multiplication gate while its representation as boolean circuit is substantially larger (cf. §5.1). As described in §2.2, the online phase for SFE of boolean circuits is substantially more efficient than SFE of arithmetic circuits, so especially non-linear functions such as comparisons benefit from boolean circuits [30]. The framework of [31] allows to modularly compose functions from building blocks which are compactly represented as boolean or arithmetic circuits and then convert back and forth between the representations under encryption.



(a) Boolean Circuit (b) Arithmetic Circuit

Figure 1: Function Representations

In the following, we summarize efficient methods for SFE of arithmetic and boolean circuits, and conversions between them which are implemented in TASTY. For a comprehensive description we refer to [31] and list the specific primitives implemented in TASTY in §4.3.

2.1 Homomorphic Encryption: SFE of Arithmetic Circuits

Additively homomorphic encryption schemes (e.g., [45, 12]) are semantically secure encryption schemes with plaintext space P and ciphertext space C that allow addition under encryption: The operation $+$ can be computed on plaintexts by defining a corresponding operation $\boxplus$ on ciphertexts which satisfies $\forall x, y \in P : [\![x]\!] \boxplus [\![y]\!] = [\![x+y]\!]$. This naturally allows for multiplication with a plaintext constant a using repeated doubling and adding: $\forall a \in \mathbb{N}, x \in P : a[\![x]\!] = [\![ax]\!]$. We write $[\![x]\!]$ for homomorphic encryption of plaintext x.

SFE of arithmetic circuits can be naturally based on additively homomorphic encryption as follows: Client $\mathcal{C}$ generates a key-pair for the homomorphic cryptosystem and sends the public key together with his inputs encrypted under the public key to server $\mathcal{S}$. $\mathcal{S}$ uses the homomorphic property to evaluate the arithmetic circuit on the encrypted data. If the cryptosystem is only additively homomorphic, multiplication under encryption requires the help of $\mathcal{C}$ in a single round of interaction (details in [31]). Finally, $\mathcal{S}$ sends the encrypted outcome of the computation back to $\mathcal{C}$ who can decrypt. As often the maximum size of elements in the plaintext space (e.g., $P = \mathbb{Z}_n$ with RSA modulus n for the Paillier cryptosystem [45]) is substantially larger than the size of encrypted values, $\mathcal{S}$ can *pack* multiple values under encryption using Horner's method before sending them to $\mathcal{C}$ to reduce communication and number of decryptions by $\mathcal{C}$.

As described in [31], the interactive approach for multiplication currently results in faster SFE protocols than using schemes which also provide one (e.g., [6, 18]) or arbitrary many (e.g., [17, 13, 55]) multiplications under encryption, called *fully homomorphic encryption*. Such schemes could be integrated in TASTY in future work as described in §6.

2.2 Garbled Circuits: SFE of Boolean Circuits

Garbled circuits (GC) are an efficient method for SFE of boolean circuits. The general idea of GCs, going back to Yao [60], is to encrypt (*garble*) each wire with a symmetric encryption scheme. In contrast to homomorphic encryption (cf. §2.1), the encryptions/garblings here cannot be operated on directly, but require helper information which is generated and sent to $\mathcal{C}$ in the setup phase in form of a *garbled table* for each gate. On the other hand, the online phase of GCs is highly efficient as it requires only symmetric cryptographic operations, e.g., the GC method of [48] implemented in TASTY needs one invocation of SHA-256 per non-XOR gate (cf. §4.3).

On a high-level, Yao's GC protocol works as follows: In the setup phase, the *constructor* (server $\mathcal{S}$) generates an encrypted version of the function f (represented as boolean circuit), called *garbled circuit* $\widetilde{f}$. For this, he assigns to each wire W_i of f two randomly chosen garbled values $\widetilde{w}_i^0, \widetilde{w}_i^1$ (symmetric keys) that correspond to the respective values 0 and 1. Note that $\widetilde{w}_i^j$ does not reveal any information about its plain value j as both keys look random. Then, for each gate of f, the constructor creates helper information in form of a *garbled table* $\widetilde{T}_i$ that allows to decrypt only the output key from the gate's input keys. The garbled circuit $\widetilde{f}$ consists of the garbled tables of all gates and is sent to $\mathcal{C}$ in the setup phase. Later, in the online phase the *evaluator* (client $\mathcal{C}$) obliviously obtains the garbled values $\widetilde{x}$ and $\widetilde{y}$ corresponding to the plain inputs x and y of $\mathcal{C}$ and $\mathcal{S}$, respectively (see below). Afterwards, $\mathcal{C}$ evaluates the garbled circuit $\widetilde{f}$ on $\widetilde{x}, \widetilde{y}$ by evaluating the garbled gates one-by-one using their garbled tables. Finally, $\mathcal{C}$ obtains the corresponding garbled output values $\widetilde{y}$ which allow $\mathcal{S}$ to decrypt them into the corresponding plain output $z = f(x, y)$.

For converting a plain input bit y_i of $\mathcal{S}$ into its garbled equivalent, $\mathcal{S}$ simply sends the key $\widetilde{y}_i^{y_i}$ to $\mathcal{C}$. Similarly, $\mathcal{C}$ must obtain the garbled bit $\widetilde{x}_i$ corresponding to his input bit x_i, but without $\mathcal{S}$ learning x_i. This can be achieved by running (in parallel for each bit x_i of x) a 1-out-of-2 *Oblivious Transfer (OT)* protocol. OT is a cryptographic protocol into which $\mathcal{C}$ inputs his choice bit $b = x_i$ and $\mathcal{S}$ inputs two strings $s^0 = \widetilde{x}_i^0$ and $s^1 = \widetilde{x}_i^1$. The protocol guarantees that $\mathcal{C}$ obtains only the chosen string $s^b = \widetilde{x}_i^{x_i} =$

[4]Ordered Binary Decision Diagrams (OBDDs), an alternative function representation which also fits into the framework of [31] is not implemented in TASTY yet.

$\widetilde{x}_i$ while $\mathcal{S}$ learns no information on $b = x_i$. We summarize efficient instantiations for parallel OT later in §4.3.

We emphasize that GCs cannot be evaluated twice, and refer to [33] for a proof of security for Yao's protocol in the semi-honest model and to [31] for a summary of different methods for constructing garbled tables and converting garbled outputs into plain values.

2.3 Hybrid SFE of Mixed Representations

The SFE framework proposed in [31] allows to modularly compose SFE protocols as sequence of operations on encrypted data as shown in Fig. 2: Both parties have *Plain Values* as their inputs into the protocol. These plain values, denoted as x, are first encrypted by converting them into their corresponding encrypted value. A *Garbled Value*, denoted as $\widetilde{x}$, held by client $\mathcal{C}$ or a *Homomorphic Value*, denoted as $[\![x]\!]$ held by server $\mathcal{S}$, depending on which operations should be applied. After encryption, the function is securely evaluated on the encrypted values, which may involve conversion of the encryptions into the respective other type of encryption (see below). Finally, the encrypted output values are revealed and can be decrypted by converting them into their corresponding plain output values. In the following we describe how to efficiently convert between the two types of encryptions.

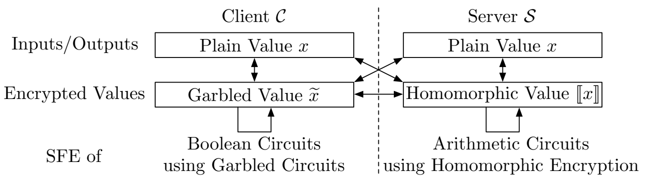

Client $\mathcal{C}$
Server $\mathcal{S}$
Inputs/Outputs
Plain Value x
Plain Value x
Encrypted Values
Garbled Value $\widetilde{x}$
Homomorphic Value $[\![x]\!]$
SFE of
Boolean Circuits using Garbled Circuits
Arithmetic Circuits using Homomorphic Encryption

Figure 2: Hybrid SFE Protocols

Conversion between Garbled and Homomorphic Values. To convert an Homomorphic Value $[\![x]\!]$ into a Garbled Value $\widetilde{x}$, $\mathcal{S}$ adds a random mask r under homomorphic encryption, sends the blinded value $[\![\bar{x}]\!] = [\![x]\!] \boxplus [\![r]\!]$ to $\mathcal{C}$ who decrypts and both parties evaluate a garbled subtraction circuit which takes off the random mask under "garbled encryption". A similar method can be used for converting a Garbled Value $\widetilde{x}$ into an Homomorphic Value $[\![x]\!]$. For details we refer to [31].

3. SELECTED APPLICATIONS

In this section we show how the TASTY framework can be used to intuitively describe, and automatically generate and measure the performance of two privacy-preserving applications. We consider privacy-preserving set intersection (§3.1) and privacy-preserving face recognition (§3.2). A detailed description of TASTY and its input language TASTYL is given later in §4; further performance results and the specs of the machines used in our performance measurements are given in §5.

3.1 Privacy-Preserving Set Intersection

Privacy-preserving set intersection is a fundamental building block for many privacy-preserving applications such as privacy-preserving checking of no-flight list. We briefly summarize the HE-based set-intersection protocol of [16, 34]:

Two parties, client $\mathcal{C}$ and server $\mathcal{S}$, have as inputs a set $X = \{x_1, \ldots, x_m\}$ respectively $Y = \{y_1, \ldots, y_n\}$. The protocol should compute the intersection $X \cap Y$ without revealing any other elements to the other party. The main idea behind this protocol is to encode X as a polynomial $p(x)$ whose roots are the m values x_i, i.e., $p(x) = (x - x_1)(x - x_2) \ldots (x - x_m) = \sum_m a_i x^i$. $\mathcal{C}$ computes the coefficients a_i of $p(x)$, encrypts them separately using homomorphic encryption and then sends these ciphertexts to $\mathcal{S}$. Then, $\mathcal{S}$ evaluates the polynomial p under homomorphic encryption: $[\![p(y_i)]\!] = [\![a_k]\!] y_i^k \boxplus [\![a_{k-1}]\!] y_i^{k-1} \boxplus \ldots \boxplus [\![a_0]\!]$. This is done efficiently with Horner's method. Now, for each $y_i \in Y$, $\mathcal{S}$ picks a random value r_i, computes $[\![\bar{y}_i]\!] = [\![r_i * p(y_i) + y_i]\!]$, and sends it to $\mathcal{C}$. If y_i is equal to an element in X, then this is an encryption of y_i (as $p(y_i)$ evaluates to 0), and of a random element otherwise. $\mathcal{C}$ finally decrypts $[\![\bar{y}_i]\!]$ into $\bar{y}_i$, and if $\bar{y}_i \in X$, $\mathcal{C}$ puts $\bar{y}_i$ into the intersection set.

This protocol can be implemented in TASTYL as listed in Appendix §A. The performance for random 32-bit inputs, measured automatically with TASTY, is shown in Table 2.

Table 2: Set Intersection of [16, 34] with TASTY.

Elements ($m = n$)	10	100	1,000
$\mathcal{C}$ setup	153 ms	969 ms	9.3 s
$\mathcal{S}$ setup	194 ms	1.6 s	15.8 s
$\mathcal{C}$ online	357 ms	7.2 s	489 s
$\mathcal{S}$ online	216 ms	6.2 s	478 s
Total send	19.6 kB	186 kB	1.86 MB

3.2 Privacy-Preserving Face Recognition

For privacy-preserving face recognition, client $\mathcal{C}$ has a query face which should be searched in a database (DB) of faces held by server $\mathcal{S}$ without disclosing any additional information on the queried face to $\mathcal{S}$ nor any information on the DB to $\mathcal{C}$ (besides the size and the outcome of the computation). At the end, $\mathcal{C}$ obtains either the index of the queried face in the DB, or $\perp$ if no match was found.

We summarize the face recognition protocol of [50] which evaluates the well-known Eigenface algorithm [57] under encryption and can be divided into the following three phases:
Projection. First, the query face Γ is projected into a low-dimensional eigenspace. This is done under homomorphic encryption as follows: $\mathcal{C}$ encrypts Γ pixelwise and sends $[\![\Gamma]\!]$ to $\mathcal{S}$ who performs the projection under encryption and obtains the encrypted projected query face $[\![\bar{\Omega}]\!]$.
Distance. Then, the squared Euclidean distance $[\![D_i]\!] = [\![(\Omega_i - \bar{\Omega})^2]\!]$ between the projected face and all faces Ω_i in $\mathcal{S}$'s DB is computed under homomorphic encryption.
Minimum. Finally, the minimum value of $\{[\![D_i]\!]\}$ is computed and, if smaller than a threshold τ provided by $\mathcal{S}$, the corresponding index in the DB is revealed to $\mathcal{C}$. Otherwise, no match was found and $\perp$ is returned. The protocol of [50] improves over [15] by computing this phase with garbled circuits instead of homomorphic encryption.

The TASTYL code of this protocol is given in Appendix §B.
Performance. In the following we compare the performance of this protocol implemented in TASTY with its hand-optimized implementation of [50] and the original protocol of [15] based on HE only. As previous works we perform our measurements with ultra-short term security parame-

ters (cf. Table 4). The results are summarized in Table 3 and visualized in Fig. 3 (time) and Fig. 4 (communication).

Setup Phase. While [50] focused on the online phase only, we also provide performance measurements for the setup phase. As expected, both time and communication of the setup phase grow linearly in the DB size (corresponding to linear circuit size and number of OTs). A constant overhead is needed for pre-computing random masks for HE ($\approx 20s$). The setup phase is less efficient than that of the HE-only protocol of [15] as GCs need to be generated and transferred.

Online Phase. When comparing the online time of the protocol generated by TASTY with the hand-optimized implementation of [50] we observe that they mostly differ by a constant overhead. In fact, the online phase is dominated by homomorphically encrypting the vector Γ in the projection phase which is not yet optimized in TASTY. For the online communication complexity we observe that data serialization in TASTY, which is also not optimized yet, requires approximately twice as much data as the theoretical lower bound of the protocol of [50]. Still, even without further serialization optimizations, the online communication complexity outperforms that of [15] already for databases with slightly more than 200 faces. We note that the communication complexity of [15] closely matches the theoretical lower bound, i.e., their serialization cannot be optimized further.

Table 3: Hybrid Face Recognition with TASTY

		Time in s		Communication in MBytes	
	|DB|	Setup	Online	Setup	Online
[15]	320	22	18	7.3	
[50]	320	-	8.4	-	2.8
TASTY	320	38.1	41.5	3.3	5.9
[50]	1,000	-	13	-	3.5
TASTY	1,000	83.4	56.2	10.2	6.8

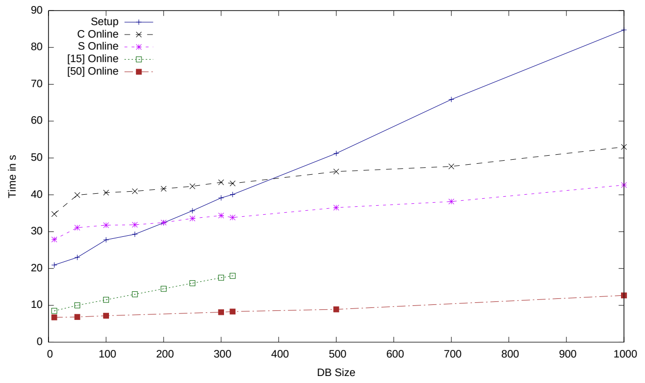


Figure 3: Hybrid Face Recognition: Times

SCiFI - a system for secure face identification. We note that the recent face recognition system of [44], consisting of a novel recognition algorithm which was co-designed together with a highly efficient SFE protocol, is more accurate and efficient than Eigenface-based protocols.

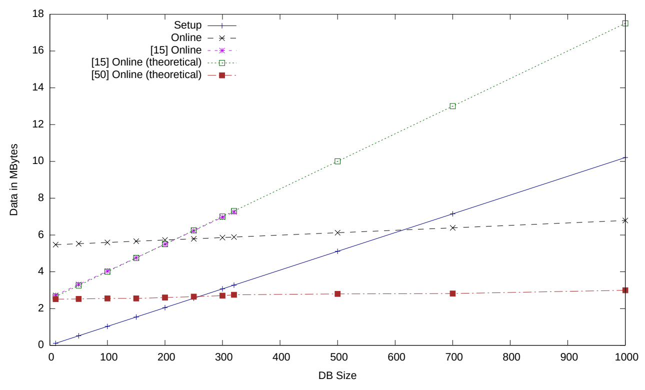


Figure 4: Hybrid Face Recognition: Communication

4. TASTY

In this section we present TASTY, our tool for describing and automatically generating, benchmarking, and evaluating hybrid secure two-party computation protocols.

Design Goals. TASTY was designed and developed to meet the following goals:

1. SFE protocols are *programmed* in TASTYL, an intuitive high-level language for describing the protocol as sequence of operations on encrypted data (cf. §4.1).
2. TASTY allows to *test, benchmark and compare* the performance of the generated SFE protocols (cf. §4.2).
3. The generated SFE protocols aim at *minimizing the latency of the online phase*, i.e., the time from providing the inputs until obtaining the outputs. This is achieved by using a combination of highly efficient primitives and pre-computations (cf. §4.3).

Architecture and Workflow (cf. Fig. 5). The workflow for using TASTY is as follows:

1. Both users, client $\mathcal{C}$ and server $\mathcal{S}$, agree on a *Protocol Description* of the SFE protocol in the TASTY input Language (TASTYL) as described in detail in §4.1.
2. Both users invoke TASTY's *Runtime Environment* (details later in §4.2), a program that can automatically analyze, run, test, and benchmark the SFE protocol:
   (a) In the *Analyzation Phase*, the runtime environment checks the syntactical correctness of the protocol description, exchanges a hash of it to ensure that both parties run the same protocol, and analyzes the protocol to automatically determine which parts of the protocol can be pre-computed.
   (b) In the *Setup Phase*, the parties pre-compute those parts of the protocol which are independent of their inputs, e.g., create/send garbled circuits and oblivious transfers (OT), see §4.3 for details.
   (c) Finally, in the *Online Phase*, both parties provide their inputs to the computation, and the online part of the SFE protocol is executed (e.g., homomorphic encryptions and decryptions, online OTs, and evaluation of GCs) to jointly compute the respective outputs for both parties.
3. TASTY provides a tool to compare the performance costs of multiple SFE protocols as described in §4.2.

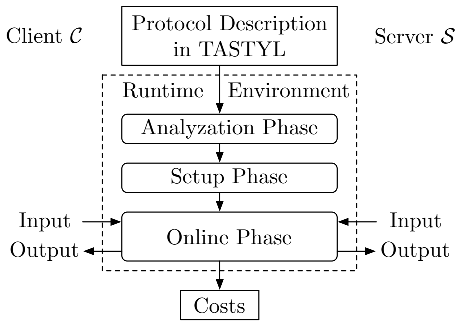


Figure 5: Architecture and Workflow of TASTY

Implementation. We selected Python as implementation language for TASTY as it combines elements from both, object oriented and functional programming paradigms. In particular the built-in support for generators, a function which yields a value and can be resumed afterwards, was useful for intuitive programming of streamlined large data structures, e.g., for dynamic generation of circuits which allows TASTY to evaluate very large circuits. We successfully created and evaluated garbled circuits with 2^{21} non-XOR gates in less than 14 minutes on the PCs used for the experiments in §5.

4.1 TASTY input Language (TASTYL)

TASTYL, the input language for TASTY, allows to formulate secure computations as sequence of operations on encrypted data, allowing to abstract away all details of the underlying cryptographic protocols. We start with an overview of the types and operators provided by TASTYL in §4.1.1 and explain the concrete syntax afterwards in §4.1.2.

4.1.1 Types and Operators

The type system of TASTYL and the operators supported by each type are shown in Fig. 6. Each variable in TASTYL is either a scalar *Value* (cf. top half of Fig. 6) or a *Vector* (cf. bottom half of Fig. 6) which consists of *N* Values. They can be either unencrypted *Plain Values/Vectors* or encrypted *Garbled or Homomorphic Values/Vectors*.

All Values and Vectors provide the basic operators for (component-wise) addition, subtraction, and multiplication; Vectors also provide dot multiplication: $\mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{N} v_i w_i$.

Number Representation. Each Value has a bitlength ℓ that represents the number of bits needed for its representation. *Unsigned* are unsigned integer values in the range $[0, 2^\ell[$, *Signed* are signed integers in the range $]-2^{\ell-1}, 2^{\ell-1}[$[5], and *Modular* are elements in the plaintext space of the homomorphic cryptosystem, i.e., $\mathbb{Z}_n$ for Paillier.

In addition to the operations of Value/Vector, the plain/encrypted types support further operations and conversions:

Plain Value/Vector. Inputs and outputs of the two parties are *Plain Values/Vectors*. They can be chosen uniformly at random and provide additional operations (integer) division[6] and comparison.

[5]Note, we exclude the value $-2^{\ell-1}$ for signed integers to also allow sign-magnitude representation.

[6]Division raises an exception for division by zero or (the unlikely event of) a non-invertible Modular value.

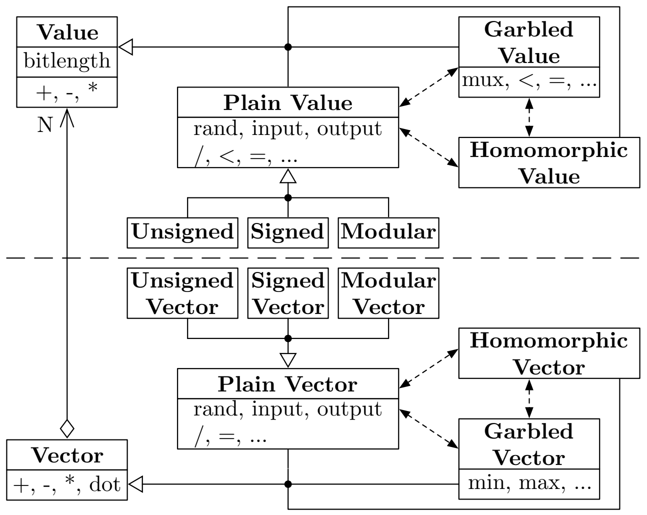


Figure 6: TASTYL Types and Operators

Homomorphic Value/Vector. Unsigned, Signed and Modular Values/Vectors can be converted into and from homomorphically encrypted *Homomorphic Values/Vectors* of server $\mathcal{S}$. While Unsigned and Modular values are mapped directly, for Signed values, the positive values are mapped to the elements $0, 1, \ldots$ of the plaintext space of the underlying homomorphic cryptosystem, and the negative values to $n-1, n-2, \ldots$ as described in [31]. Addition of two Homomorphic, and (dot) multiplication of a Homomorphic with a Plain Value/Vector provided by $\mathcal{S}$ is done non-interactively. (Dot) multiplication of two Homomorphic Values/Vectors requires one round of interaction.

Garbled Value/Vector. Unsigned/Signed Plain and Homomorphic Values/Vectors can be converted into and from *Garbled Values/Vectors* of client $\mathcal{C}$. A Garbled Value can be compared with another one resulting in a Garbled Value of length one bit. This can be used to multiplex (`mux`) between two Garbled Values. Similarly, the minimum or maximum value and/or index of the components of a Garbled Vector can be determined as Garbled Value(s), e.g., `min_value` computes the minimum value. For each operation on Garbled Values/Vectors, TASTY automatically infers the underlying garbled circuit.

4.1.2 Syntax and Example

TASTYL is a subset of the Python language; we use the following example to explain its syntax and semantics.

Example (cf. Fig. 7). Client $\mathcal{C}$ and server $\mathcal{S}$ have vectors $\mathbf{v}$ and $\mathbf{w}$ of $N = 4$ unsigned 32-bit values as inputs. As output, $\mathcal{C}$ obtains $\min_{i=1,..,N}(v_i \cdot w_i)$. The products $v_i \cdot w_i$ are computed with homomorphic encryption (HE) and the minimum with garbled circuits (GC).

This protocol can be directly formulated in TASTYL as shown in Fig. 7 and described in the following: The protocol gets two parties `client` and `server` as inputs to whom the variables used throughout the protocol are bound (details below). At the beginning, two constants $N = 4$ and $L = 32$ are defined. Then, the input of $\mathcal{C}$, `client.v`, is defined as an unsigned vector of bitlength L and dimension N, and read from standard input. Similarly, the input of $\mathcal{S}$, `server.w`, is

```
# -*- coding: utf-8 -*-
def protocol(client, server):
    N = 4
    L = 32

    # input of client
    client.v = UnsignedVec(bitlen=L, dim=N)
    client.v.input(desc="enter values for v")

    # input of server
    server.w = UnsignedVec(bitlen=L, dim=N)
    server.w.input(desc="enter values for w")

    # convert unsigned to homomorphic vector
    client.hv = HomomorphicVec(val=client.v)
    server.hv <<= client.hv

    # multiply vectors (component-wise)
    server.hx = server.hv * server.w

    # convert homomorphic to garbled vector
    client.gx <<= GarbledVec(val=server.hx)

    # compute minimum value
    client.gmin = client.gx.min_value()

    # convert garbled to unsigned value and output
    client.min = Unsigned(val=client.gmin)
    client.min.output(desc="minimum value")
```

Figure 7: Example TASTYL Program

defined and read. Then, $\mathcal{C}$'s input vector `client.v` is converted into a homomorphic vector `server.hv` for $\mathcal{S}$ who multiplies this component-wise with his input vector `server.w` resulting in the homomorphic vector `server.hx`. This homomorphic vector is converted into a garbled vector `client.gx` and its minimum value `client.gmin` is computed. Finally, $\mathcal{C}$ obtains the intended output by decrypting (converting) `client.gmin` into the unsigned value `client.min`.

Type Conversions. Types can be naturally converted into each other by providing them as input to the constructor of the target type, e.g., in Fig. 7, the unsigned vector `client.v` is converted into the homomorphic vector `client.hv` via `client.hv=HomomorphicVec(val=client.v)`. The underlying conversion protocols are described in §2.

Garbled Bit Manipulations. To allow manipulation of single bits, a Garbled Value `gv` can be converted back and forth into a list of *Garbled Bits* (= Garbled 1-bit Values): `gv[i]` yields the i-th garbled bit of `gv` ($i = 0$ is the least significant bit). Vice versa, a (unsigned) garbled m-bit value `gv` can be constructed from a list of m garbled bits, e.g., `gv = Garbled(val=[gb0,gb1])`.

Send Operator. The send operator `<<=` transfers variables between the parties, e.g., in Fig. 7, `hv` is sent from $\mathcal{C}$ to $\mathcal{S}$ with `server.hv <<= client.hv`. When combined with a type conversion, the send operator invokes the corresponding conversion protocol, e.g., in Fig. 7, homomorphic vector `hx` held by $\mathcal{S}$ is converted into garbled vector `gx` held by $\mathcal{C}$ with `client.gx <<= GarbledVec(val=server.hx)`.

Binding of Variables. While constants can be declared globally (e.g., `N` and `L` in Fig. 7), each variable has to be assigned to one of the parties as an attribute.

Inferring Type and Length Automatically. For each operator, TASTY automatically infers the bitlength and type of the output variables from those of the input variables s.t. no overflow occurs. Homomorphic variables raise an exception if the result does not fit into the plaintext space of the homomorphic cryptosystem. For example, in Fig. 7 the component-wise product of two vectors with `N` components of unsigned `L`-bit values results in the homomorphic vector `server.hx` with `N` components of unsigned `2L`-bit values.

Multiple Outputs. Garbled circuits can also have multiple garbled output values written as comma separated list on the left side of the assignment operator, e.g., the garbled minimum value `gv` and its index `gi` can be computed as `(client.gv, client.gi)=client.gx.min_value_index()`.

Circuits from File. TASTY allows secure evaluation of boolean circuits read from an external file, e.g., circuits generated by the FairplayMP compiler [3]. For this, the labels of the input- and output wires of the circuit are mapped to Garbled Values of corresponding bitlength. An example TASTYL file with the concrete syntax for evaluating a garbled file circuit is available at [56].

4.2 Tools

The TASTY framework provides the following tools to initialize, execute, and post-process TASTYL programs:

`tasty_init <path>` creates a new directory which contains a file `protocol.py` with a template for the TASTYL program (the example program shown in Fig. 7) and a file `protocol.ini` which contains default configuration parameters such as the intended security level (cf. Table 4), or the IP address and port of the server.

`tasty <options> <path>` is the runtime environment of TASTY as explained in §4 (cf. Fig. 5): it analyzes the TASTYL program in `path`, establishes a TCP/IP socket between server $\mathcal{S}$ and client $\mathcal{C}$, and runs the setup phase and online phase of the SFE protocol. The option flags allow to overwrite the default parameters and to specify if run as server (`-s`) or as client (`-c`).

Testing and Benchmarking. When invoked with the `-d` option, `tasty` runs in *driver mode*. Here, the TASTYL program is instrumented by a driver, an additional class written in `protocol.py`. The driver can invoke the protocol multiple times with varying static parameters (e.g., different bitlengths) and inputs to the TASTYL program; the outputs of the TASTYL program are sent back to the driver which allows to write functional test cases. The costs of each protocol run, i.e., detailed information on the transferred data and timings of the sub-tasks of the protocol phases, are written into a file which can be post-processed as described next.

`tasty_post <analyze_script> <cost_files>` can post-process the costs measured in one or more driver runs with an analyze script, e.g., average, print, or plot graphs [22]. All graphs in this paper were plotted with `tasty_post`.

A concrete example for how to use TASTY's benchmarking capability is given in Appendix §C.

4.3 Primitives and Optimizations

In TASTY we implemented the following efficient primitives and automatic optimizations that allow to move expensive operations as pre-computations into the setup phase (cf. Fig. 5) in order to achieve an online phase with low latency. The modular architecture of TASTY allows easy extension with other primitives as well. Due to the lack of space we mention the key-features of the used primitives and refer to the description in [31] and the original papers for details.

Pre-Defined Security Levels. TASTY has pre-defined security levels following standard recommendations of NIST and ECRYPT II [19] as shown in Table 4. By using matching basic primitives both security and efficiency are optimized simultaneously. We use elliptic curves from the SECG standard [53] and SHA-256 as cryptographic hash function.

Table 4: Pre-Defined Security Levels in TASTY.

Security Level	Symmetric/ Statistical	Asymmetric	Curve [53]
ultra-short	80 bit	1,248 bit	secp160r1
short	96 bit	1,776 bit	secp192r1
medium	112 bit	2,432 bit	secp224r1
long	128 bit	3,248 bit	secp256r1

Homomorphic Encryption (HE). We use the additively homomorphic cryptosystem of Paillier [45]. As key generation for Paillier (an RSA modulus n) is computationally expensive and can be used over multiple protocol runs, the public key is generated and exchanged in the analyzation phase. For efficient encryption we use the extensions of [12, Sect. 6] for pre-computing expensive modular exponentiations of the form $r^n \mod n^2$ in the setup phase and only two modular multiplications per encryption in the online phase. As $\mathcal{C}$ knows the factorization p, q of n, he uses Chinese remaindering modulo p and q for pre-computing $r^n \mod n^2$ and efficient decryption. Paillier ciphertexts have twice the length of the asymmetric security parameter as the ciphertext space is $\mathbb{Z}^*_{n^2}$. For modular arithmetics we use gmpy [21], a Python wrapper for the GMP library [20].

Garbled Circuits (GC). We use the GC construction with free XORs and garbled row reduction of [48] secure in the random-oracle model. This GC construction provides free XOR gates (no garbled table and negligible computation). For non-XOR d-input gates, the garbled table consists of $2^d - 1$ entries (of size $t + 1$ bit each with symmetric security parameter t), creation requires 2^d and evaluation 1 invocation of SHA-256 modeled as random oracle.

Circuits. For computations on Garbled Values/Vectors, TASTY dynamically generates circuits using the efficient circuit constructions of [30] which are optimized for a low number of non-XOR gates (cf. §5.1.1 for multiplication circuits). Alternatively, circuits can be generated externally, e.g., using the Fairplay compiler [37], and read from a file (cf. §4.1.2). TASTY optimizes the circuits to a low number of non-XOR gates using the optimization of [48] which replaces 3-input gates with a low number of 2-input non-XOR gates. XNOR gates are replaced by an XOR gate and an inversion gate which is propagated into successor gates [46]. Generating, reading, and optimizing circuits is mostly pipelined to allow processing of large circuits with low memory footprint.

Oblivious Transfer (OT). All OTs are pre-computed in the setup phase (cf. Fig. 5) using the construction of [2]; the resulting online phase for OT is highly efficient (transfer and XOR of bitstrings) and depends mostly on the network latency for two messages. To minimize the computation complexity of the setup phase, we use the efficient OT extension of [24] to reduce the usually large number of OTs needed in the protocol down to at most t real OTs and some invocations of SHA-256 modeled as random oracle, where t is the symmetric (computational) security parameter. The remaining real OTs (at most t) are implemented with the OT protocol of [39, Sect. 3.1] using elliptic curves and SHA-256 as random oracle. The elliptic curve implementation provides (optional) point compression to reduce communication at the cost of a negligibly larger computation overhead.

Compiler Optimizations. TASTY parses the TASTYL program and performs several optimizations on the resulting abstract syntax tree (AST): constant propagation, dead code elimination, partial code evaluation, and loop unrolling.

5. PERFORMANCE MEASUREMENTS

We measure the performance of primitives implemented in TASTY and compare different protocols against each other and with existing SFE implementations: multiplication circuits and protocols based on GC or HE (§5.1), SFE of an AES circuit generated by the Fairplay compiler (§5.2), and SFE of large GCs (§5.3).

System Setup. All performance measurements are performed on two desktop PCs with Intel Core 2 Duo CPU (E6850) running at 3.00GHz and 4GB RAM connected via Gigabit Ethernet. The system runs on 64 bit Gentoo Linux with Python version 2.6.5, gmpy version 1.11 and GMP version 4.3.2. Unless stated otherwise, all measurements were performed for short-term security (cf. Table 4) and using point compression for elliptic curves (cf. §4.3).

5.1 Multiplication Circuits and Protocols

As arithmetic circuits can express arbitrary computations as sequence of additions and multiplications, multiplication is a fundamental basic operation. Indeed, the main difference between SFE protocols based on arithmetic and boolean circuits is the cost for multiplications. We present efficient multiplication circuits in §5.1.1 and compare the performance of secure multiplication protocols in §5.1.2.

5.1.1 Multiplication Circuits

Textbook Multiplication. The usual way of multiplying two unsigned ℓ-bit integers x and y, called "Textbook Method", multiplies x with each bit of y and adds up all the properly shifted results according to the formula $x \cdot y = \sum_{i=0}^{\ell-1} xy_i 2^i$. This results in a circuit with $2\ell^2 - \ell$ non-XOR 2-input gates [30].

Karatsuba Multiplication. As observed by Karatsuba [28], multiplication can be performed more efficiently using the following recursive method (details in Algorithm 1): x and y are split into two halves as $x = x_h 2^{\lceil \ell/2 \rceil} + x_l$ and $y = y_h 2^{\lceil \ell/2 \rceil} + y_l$. Then, the product can be computed as $xy = (x_h 2^{\lceil \ell/2 \rceil} + x_l)(y_h 2^{\lceil \ell/2 \rceil} + y_l) = z_h 2^{2\lceil \ell/2 \rceil} + z_d 2^{\lceil \ell/2 \rceil} + z_l$. After computing $z_h = x_h y_h$ and $z_l = x_l y_l$, z_d can be computed with only one multiplication as $z_d = (x_h + x_l)(y_h + y_l) - z_h - z_l$. This process is continued recursively until the numbers are sufficiently small ($\ell = 19$ in our case as described below) and multiplied with the classical school method. Overall, multiplying two ℓ bit numbers with Karatsuba's method requires three multiplications of $\ell/2$ bit numbers and some additions and subtractions with linear bit complexity resulting in costs

$$T_{Kara}(\ell) = 3T_{Kara}(\ell/2) + c\ell + d$$

for constants c and d. The master theorem [8, §4.3f] yields asymptotic complexity $T_{Kara}(\ell) \in \mathcal{O}(\ell^{\log_2 3}) \approx \mathcal{O}(\ell^{1.585})$.

Circuit Complexity. In TASTY we have implemented both methods for multiplication based on efficient addition and subtraction circuits of [30]. As shown in Fig. 8 and Table 5, Karatsuba multiplication is more efficient, i.e., results

Algorithm 1 Karatsuba multiplication

```
1:  function KARATSUBA($x, y$)                ▷ $x, y$ are $\ell$-bit integers
2:      if $\ell \leq 19$ then
3:          return TEXTBOOK($\mathbf{x}, \mathbf{y}$)
4:      end if
5:      $x_h || x_l \leftarrow x$                      ▷ $x = x_h 2^{\lceil \ell/2 \rceil} + x_l$
6:      $y_h || y_l \leftarrow y$                      ▷ $y = y_h 2^{\lceil \ell/2 \rceil} + y_l$
7:      $P_h \leftarrow$ KARATSUBA($x_h, y_h$)
8:      $P_l \leftarrow$ KARATSUBA($y_l, y_l$)
9:      $x_s \leftarrow x_h + x_l$
10:     $y_s \leftarrow y_h + y_l$
11:     $P_s \leftarrow$ KARATSUBA($x_s, y_s$)
12:     $P_d \leftarrow P_s - P_h - P_l$
13:     return $(P_h 2^{2^{\lceil \ell/2 \rceil}}) + P_d 2^{\lceil \ell/2 \rceil} + P_l$
14: end function
```

in circuits with less non-XOR gates, than Textbook multiplication already for multiplication of 20 bit operands. By interpolating through the points for bitlength $\ell \in \{32, 64, 128\}$ and solving the resulting system of linear equations we obtain as approximation for the number of non-XOR gates

$$T_{Kara}(\ell) \approx 9.0165\ell^{1.585} - 13.375\ell - 34.$$

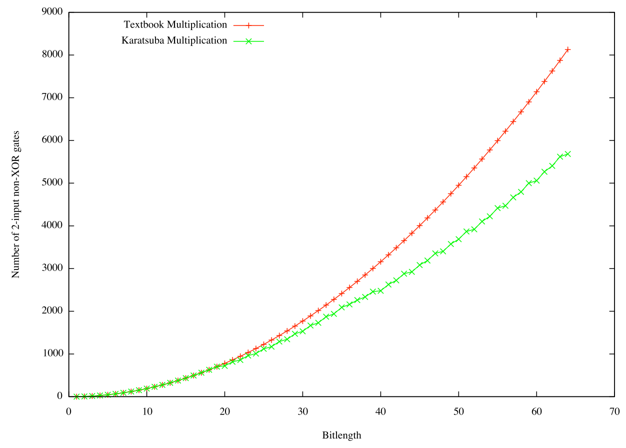


Figure 8: Size of Multiplication Circuits

Table 5: Size of Multiplication Circuits (in number of 2-input non-XOR gates)

Bitlength ℓ	19	20	32	64	128
Textbook	703	780	2,016	8,128	32,640
Karatsuba	703	721	1,729	5,683	17,973
Improvement	0.0 %	7.6 %	14.2 %	30.1 %	44.9 %

5.1.2 *Multiplication Protocols*

Using TASTY we compare the performance of different secure multiplication protocols based on homomorphic encryption (HE) and garbled circuits (GC). For this we constructed four basic test cases. For each SFE paradigm, we consider the case where both inputs are provided by one party ($\mathcal{S}$ for GC1 and $\mathcal{C}$ for HE1), or one by each of the parties (GC2 and HE2). The inputs are Unsigned ℓ-bit values and the output, a 2ℓ-bit Unsigned value is converted into a Plain output for $\mathcal{C}$. In the following, we compare the communication- and the computation complexity of the setup- and online phase of the protocols.

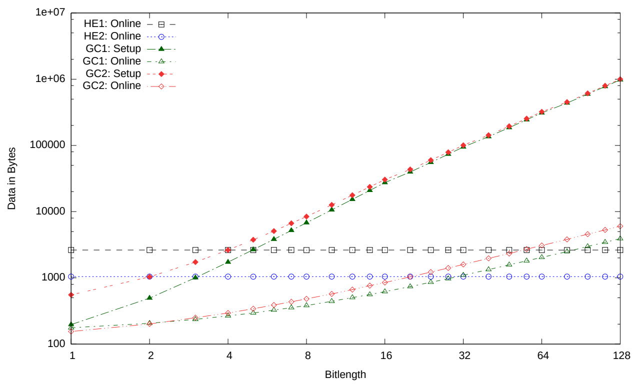


Figure 9: Multiplication Protocols: Communication

Communication (cf. Fig. 9). Our experiments show that GC-based multiplication requires a substantial amount of setup communication (for transfer of GCs) whereas the online communication of GC is better than HE for multiplication of small values. The online communication for multiplying with HE is independent of the bitlength ℓ as a constant number of ciphertexts (2 for HE1 and 5 for HE2) is exchanged. For multiplying with GC, the setup communication grows rapidly due to the large size of the GCs, whereas the online communication complexity grows much slower.

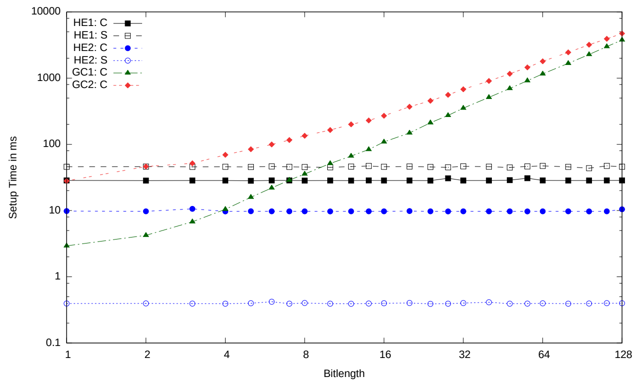
Setup Time (cf. Fig. 10(a)). The time of the setup phase for GC-based multiplication protocols depends on the bitlength ℓ as GCs need to be computed; for better visualization we do not plot GC setup times for $\mathcal{S}$ in Fig. 10(a) as they are similar to those of $\mathcal{C}$. For HE-based multiplication, the setup time is independent of ℓ as a constant number of encryptions is pre-computed.

Online Time (cf. Fig. 10(b)). For GC-based multiplication, the time needed by $\mathcal{C}$ depends on the size of the evaluated GC which grows with the bitlength ℓ; GC's online time for $\mathcal{S}$ is negligible. For HE-based multiplication, the time in the online phase is almost independent of ℓ for small bitlengths.

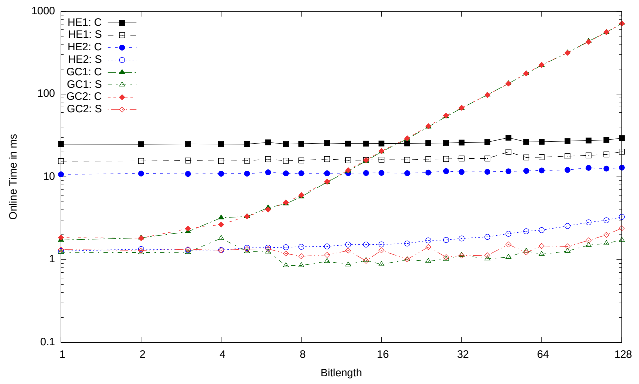
Conclusion. The setup phase for GC-based multiplication is substantially more expensive than that of HE-based multiplication. However, for small values, GC-based multiplication can result in a faster online time than HE-based multiplication. Furthermore, GC-based multiplication, in contrast to HE-based multiplication, needs no (when composed with other GC-based computations) or negligible online interaction and workload for $\mathcal{S}$.

Parallel Multiplications. When N multiplications are done in parallel, e.g., component-wise multiplication of two vectors of N components, time and data complexity of GC-based multiplication grows linearly in N. HE-based parallel multiplication increases slower as multiple homomorphic values can be packed before sending from $\mathcal{S}$ to $\mathcal{C}$ (cf. §2.1).

Security Level. We note that when the security level is



(a) Setup Time



(b) Online Time

Figure 10: Multiplication Protocols: Times

increased to medium- or even long-term security, the performance of HE-based multiplication decreases rapidly while the performance of GC-based multiplication is affected only moderately, as the asymmetric security parameter grows substantially faster than the symmetric one (cf. Table 4).

5.2 Evaluation of Fairplay Circuits and AES

As described in §4.1.2, TASTY can evaluate externally generated file circuits. Using this feature, we compare the performance of TASTY for evaluation of the AES functionality with the state of the art software implementation of GCs reported in [48, Table 2] which is implemented in C++ and measured on two machines also with Intel Core 2 Duo's running at 3.0 GHz and 4GB of RAM connected by gigabit ethernet. We use the AES circuit of [48] which has 128 bit input bits provided by each party, 128 output bits for $\mathcal{C}$ and is optimized for a low number of non-XOR gates ($22,594$ XOR gates and $11,286$ non-XOR 2-input gates).

The performance of different GC implementations for evaluating the AES functionality is compared in Table 6:

For ultra-short-term security, when evaluating AES with Fairplay's Java runtime [37], we see that Fairplay requires substantially more communication than TASTY, as Fairplay provides no free XOR gates (2/3 of the gates are XOR gates). Also TASTY's time complexity is slightly better than that of Fairplay due to free XOR and more efficient OT.

Also for long-term security, TASTY's online phase is faster than that of [48] by an order of magnitude. Recall, a short online phase, i.e., latency from providing the inputs until obtaining the outputs, is important for many real-world applications. To minimize this, TASTY shifts most computations into the less time-critical setup phase (cf. §4). Also TASTY has a slightly shorter total time than [48], whereas the data complexity is slightly larger due to less optimal data serialization in Python. More detailed, the setup time of [48] is 1s for GC creation and 1s for data transfer, and the online time is 3s for OT[7] and 2s for GC evaluation. In TASTY the setup time is dominated by 1.1s for OT and 1.8s for GC creation, and the online time is dominated by 0.4s for GC evaluation.

Table 6: GC Evaluation of AES. Times in seconds.

		Time			KByte
	Security	Setup	Online	Total	Total
[37]	ultra-short	-	-	4	3760
TASTY	ultra-short	2.9	0.4	3.3	567
[48]	long	2	5	7	503
TASTY	long	4.0	0.5	4.5	860

5.3 Evaluation of Large GCs

As AES, compared in §5.2, is the only non-trivial circuit for which performance results are reported in [48] and the code is not available yet, we compare the performance of TASTY for evaluation of large GCs with the original Fairplay system [37] implemented in Java.

Our test circuits have 10 unsigned 8-bit input values[8] provided by $\mathcal{C}$, $|C|$ non-XOR 2-input gates and one output bit for $\mathcal{S}$. For our measurements we use ultra-short term security parameters which provide the same security as Fairplay. Our measurement results are shown in Table 7.

Comparison. *Memory.* With 4GB memory, Fairplay was not able to evaluate circuits with 2^{22} gates and raised an OutOfMemoryError, while TASTY ran out of memory for circuits with 2^{23} gates. In both cases, the huge blowup in memory is due to allocation of intermediate objects for each gate.

Communication. As expected, TASTY needs less communication than Fairplay as the chosen GC technique needs 3 instead of 4 entries per garbled 2-input non-XOR gate. Further, as GC is transferred in the setup phase, TASTÝs online communication is independent of the circuit size.

Time. TASTY's setup time is slower than Fairplay's total time by approximately a factor of 2 which we assume is due to less efficient internal data structures and nested function calls in the completely interpreted Python language compared to bytecode-compiled Java. On the other hand, TASTY's online time is faster than the total time of Fairplay by factor 2 as we shifted complexity into the setup phase.

Conclusion. From the performance measurements with previous GC implementations in §5.2 and §5.3 we conclude that choosing more efficient primitives has large impact on the communication complexity whereas the memory and time complexity are dominated by non-cryptographic factors

[7]As OT seemed not to be the performance bottleneck in [48], they implemented a less efficient, UC secure OT protocol.
[8]This yields the maximum number of 80 OTs before the OT extension kicks in which is not provided by Fairplay.

Table 7: **Evaluation of Large GCs**

$\|C\|$	Time [s]			Communication [MB]		
	[37]	TASTY Setup	TASTY Online	[37]	TASTY Setup	TASTY Online
2^{10}	2	0.8	0.3	0.155	0.07	0.003
2^{15}	4	6	1	3.67	1.7	0.003
2^{18}	17	44	8	28.8	13.5	0.003
2^{20}	80	177	32	113	54.2	0.003
2^{21}	175	368	65	226	109	0.003
2^{22}	-	787	132	-	217	0.003
2^{23}	-	-	-	-	-	-

such as optimizing data structures and flow for the selected programming language.

6. FUTURE WORK

To facilitate future work, TASTY is available for download at [56]. It is ready for being used as a tool for describing, implementing, benchmarking, and comparing protocols for many privacy-preserving applications. It could also be extended into a platform for comparing cryptographic primitives, e.g., rapidly emerging (fully) homomorphic encryption schemes.

Further Primitives. By adding 1-out-of-n OT as further primitive, TASTY could be used for Hamming distance based computations [25] (based on HE and 1-out-of-n OT) with application to secure face identification [44]. Also other additively homomorphic encryption schemes for large [12] or small [9, 10] ciphertext space, or the schemes of [6, 18] which allow arbitrary many additions and one multiplication might be useful for some protocols. If applications require multiplication of very large numbers within a circuit one might consider implementation of multiplication circuits which are asymptotically faster than Karatsuba multiplication, e.g., Toom-3 [29, Sect. 4.3.3.A] splits each factor into 3 parts and performs 5 instead of 9 sub-multiplications resulting in complexity $\Theta(n^{\log_3 5}) \approx \Theta(n^{1.47})$.

Compilation to TASTYL. As a long-term goal it would be beneficial to automatically generate TASTYL programs from a high-level description of the algorithm to be computed securely in a function description language such as Fairplay's SFDL language (cf. §1.2). Using TASTY's capabilities for measuring the costs of the generated protocols the compilation process could automatically choose between circuits or homomorphic encryption (with one out of multiple homomorphic encryption schemes) for specific sub-tasks to generate highly efficient protocols.

Streaming vs. Pre-Computation. As discussed in §4, a crucial design goal of TASTY is to shift as many operations as possible into the setup phase resulting in a highly efficient online phase. This paradigm is justified when the two parties have a long-term relation where the function is known in advance and pre-computations can be performed ahead of time; this leads to a quick response time once the parties provide their inputs in the online phase. In some application scenarios where parties make ad-hoc decision when and what to compute securely, such pre-computations are not possible. HE-based SFE protocols naturally allow to change the evaluated functionality on-the-fly and stream data as soon as it is ready to keep CPUs and network busy simultaneously (e.g., as implemented in VIFF [11]). Also GC-based SFE protocols can be adapted to this streaming scenario as described in [26, 27, 51]: In contrast to the compilation paradigm used in Fairplay [37, 3], TASTY already generates the circuits on-the-fly gate-by-gate. Fore each gate, the garbled table can be generated on-the-fly [26], sent over the network and evaluated directly [27]. Also OT can be extended on-the-fly as mentioned in [24].

7. ACKNOWLEDGEMENTS

We thank anonymous reviewers of ACM CCS'10 for their helpful comments and Benny Pinkas for his suggestions on an early version of this paper. This work was funded by EU FP7 project CACE and ECRYPT II.

8. REFERENCES

[1] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. Secure evaluation of private linear branching programs with medical applications. In *European Symposium on Research in Computer Security (ESORICS'09)*, volume 5789 of *LNCS*, pages 424–439. Springer, 2009.

[2] D. Beaver. Precomputing oblivious transfer. In *Advances in Cryptology – CRYPTO'95*, volume 963 of *LNCS*, pages 97–109. Springer, 1995.

[3] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (ACM CCS'08)*, pages 257–266. ACM, 2008. http://fairplayproject.net/fairplayMP.html.

[4] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Symp. on Theory of Comp. (STOC'88)*, pages 1–10. ACM, 1988.

[5] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security (ESORICS'08)*, volume 5283 of *LNCS*, pages 192–206. Springer, 2008.

[6] D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Theory of Cryptography (TCC'05)*, volume 3378 of *LNCS*, pages 325–341. Springer, 2005.

[7] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. Privacy-preserving remote diagnostics. In *ACM Conference on Computer and Communications Security (ACM CCS'07)*, pages 498–507. ACM, 2007.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press and McGraw-Hill Book Company, 2001.

[9] I. Damgård, M. Geisler, and M. Krøigaard. Efficient and secure comparison for on-line auctions. In *Australasian Conference on Information Security and Privacy (ACISP'07)*, volume 4586 of *LNCS*, pages 416–430. Springer, 2007.

[10] I. Damgård, M. Geisler, and M. Krøigaard. A correction to "Efficient and Secure Comparison for On-Line Auctions". Cryptology ePrint Archive, Report 2008/321, 2008. http://eprint.iacr.org.

[11] I. Damgård, M. Geisler, M. Krøigård, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key*

Cryptography (PKC'09), volume 5443 of *LNCS*, pages 160–179. Springer, 2009. `http://viff.dk`.

[12] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public-Key Cryptography (PKC'01)*, volume 1992 of *LNCS*, pages 119–136. Springer, 2001.

[13] M. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology – EUROCRYPT'10*, LNCS, pages 24–43. Springer, 2010.

[14] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology – CRYPTO'84*, volume 196 of *LNCS*, pages 10–18. Springer, 1985.

[15] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *Privacy Enhancing Technologies (PET'09)*, volume 5672 of *LNCS*, pages 235–253. Springer, 2009.

[16] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology – EUROCRYPT'04*, volume 3027 of *LNCS*, pages 1–19. Springer, 2004.

[17] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC'09)*, pages 169–178. ACM, 2009.

[18] C. Gentry, S. Halevi, and V. Vaikuntanathan. A simple BGN-type cryptosystem from LWE. In *Advances in Cryptology – EUROCRYPT'10*, volume 6110 of *LNCS*, pages 506–522. Springer, 2010.

[19] D. Giry and J.-J. Quisquater. Cryptographic key length recommendation, 2010. `http://keylength.com`.

[20] GMP – GNU multi precision arithmetic library. `http://gmplib.org`.

[21] gmpy – multiprecision arithmetic for Python. `http://code.google.com/p/gmpy`.

[22] Gnuplot.py. `http://gnuplot-py.sourceforge.net`.

[23] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM Conference on Computer and Communications Security (ACM CCS'10)*, pages 451–462. ACM, 2010.

[24] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[25] A. Jarrous and B. Pinkas. Secure hamming distance based computation and its applications. In *Applied Cryptography and Network Security (ACNS'09)*, volume 5536 of *LNCS*, pages 107–124. Springer, 2009.

[26] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Embedded SFE: Offloading server and network using hardware tokens. In *Financial Cryptography and Data Security (FC'10)*, volume 6052 of *LNCS*, pages 207–221. Springer, 2010.

[27] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In *Cryptographic Hardware and Embedded Systems (CHES'10)*, volume 6225 of *LNCS*, pages 383–397. Springer, 2010.

[28] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *SSSR Academy of Sciences*, 145:293–294, 1962.

[29] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1997.

[30] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security (CANS'09)*, volume 5888 of *LNCS*, pages 1–20. Springer, 2009.

[31] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security (JCS)*, 21(2):283–315, 01 2013. Preliminary version available at `http://eprint.iacr.org/2010/079`.

[32] V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *Financial Cryptography and Data Security (FC'08)*, volume 5143 of *LNCS*, pages 83–97. Springer, 2008.

[33] Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009. Cryptology ePrint Archive: Report 2004/175.

[34] Y. Lindell and B. Pinkas. Secure multiparty computation for privacy-preserving data mining. *J. of Privacy and Confidentiality*, 1(1):59–98, 2009.

[35] Y. Lindell, B. Pinkas, and N. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *Security and Cryptography for Networks (SCN'08)*, volume 5229 of *LNCS*, pages 2–20. Springer, 2008.

[36] P. MacKenzie, A. Oprea, and M. K. Reiter. Automatic generation of two-party computations. In *ACM Conference on Computer and Communications Security (ACM CCS'03)*, pages 210–219. ACM, 2003.

[37] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security Symposium*, 2004. `http://fairplayproject.net/fairplay.html`.

[38] T. Moran. The qilin crypto SDK – an open-source java SDK for rapid prototyping of cryptographic protocols. `http://qilin.seas.harvard.edu`.

[39] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *ACM-SIAM Symposium On Discrete Algorithms (SODA'01)*, pages 448–457. Society for Industrial and Applied Mathematics, 2001.

[40] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *ACM Conf. on Electronic Commerce*, pages 129–139, 1999.

[41] J. D. Nielsen. *Languages for Secure Multiparty Computation and Towards Strongly Typed Macros*. PhD thesis, University of Aarhus, Denmark, 2009.

[42] J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *Workshop on Programming Languages and Analysis for Security (PLAS'07)*, pages 21–30. ACM, 2007.

[43] J. Nzouonta, M. C. Silaghi, and M. Yokoo. Secure computation for combinatorial auctions and market exchanges. In *Autonomous Agents and Multiagent Systems (AAMAS'04)*, pages 1398–1399. IEEE, 2004.

[44] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI - a system for secure face identification. In *IEEE Symposium on Security & Privacy (S&P'10)*, pages 239–254. IEEE, 2010.

[45] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, 1999.

[46] A. Paus, A.-R. Sadeghi, and T. Schneider. Practical secure evaluation of semi-private functions. In *Applied Cryptography and Network Security (ACNS'09)*, volume 5536 of *LNCS*, pages 89–106. Springer, 2009.

[47] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology – CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, 1992.

[48] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT'09*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

[49] A.-R. Sadeghi and T. Schneider. Generalized universal circuits for secure evaluation of private functions with application to data classification. In *International Conference on Information Security and Cryptology (ICISC'08)*, volume 5461 of *LNCS*, pages 336–353. Springer, 2008.

[50] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *12th International Conference on Information Security and Cryptology (ICISC '09)*, LNCS. Springer, 2009.

[51] A.-R. Sadeghi, T. Schneider, and M. Winandy. Token-based cloud computing – secure outsourcing of data and arbitrary computations with lower latency. In *Trust and Trustworthy Computing (TRUST'10) - Workshop on Trust in the Cloud*, volume 6101 of *LNCS*, pages 417–429. Springer, 2010.

[52] A. Schröpfer, F. Kerschbaum, D. Biswas, S. Geißinger, and C. Schütz. L1 – faster development and benchmarking of cryptographic protocols. In *ECRYPT Workshop on Software Performance Enhancements for Encryption and Decryption and Cryptographic Compilers (SPEED-CC '09)*, October 12-13, 2009.

[53] Standards for efficient cryptography, SEC 2: Recommended elliptic curve domain parameters. Technical report, Certicom Research, 2000. Available from `http://www.secg.org`.

[54] M. C. Silaghi. SMC: Secure multiparty computation language. `http://cs.fit.edu/~msilaghi/SMC/`, 2004.

[55] N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography (PKC'10)*, volume 6056 of *LNCS*, pages 420–443. Springer, 2010.

[56] TASTY – Tool for Automating Secure Two-partY computations. `http://tastyproject.net`.

[57] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[58] L. G. Valiant. Universal circuits (preliminary report). In *ACM Symposium on Theory of Computing (STOC'76)*, pages 196–203. ACM, 1976.

[59] A. C. Yao. Protocols for secure computations. In *IEEE Symposium on Foundations of Computer Science (FOCS'82)*, pages 160–164. IEEE, 1982.

[60] A. C. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science (FOCS'86)*, pages 162–167. IEEE, 1986.

APPENDIX

A. SET INTERSECTION (§3.1) IN TASTYL

```
# -*- coding: utf-8 -*-
from tasty.crypt.math import \
    getPolyCoefficients

def protocol(c, s):
    M = 100  # size of client's set
    N = 100  # size of server's set

    c.X = ModularVec(dim=M).input(desc="X")
    s.Y = ModularVec(dim=N).input(desc="Y")

    # interpolate coeffs of poly with roots c.X
    c.a = getPolyCoefficients()(c.X)

    # encrypt and send coefficients to server
    s.ha <<= HomomorphicVec(val=c.a)

    # evaluate and rerandomize p(y_i) under enc
    s.hbarY = HomomorphicVec(dim=N)
    for i in xrange(N):   # 0, ..., N-1
        # eval poly using Horner scheme
        s.p = s.ha[M]
        for j in xrange(M-1,-1,-1):
            s.p = (s.p * s.Y[i]) + s.ha[j]
        s.hbarY[i] = s.p * Modular().rand() + \
            Homomorphic(val=s.Y[i])

    # send hbarY to client and decrypt
    c.hbarY <<= s.hbarY
    c.barY     = ModularVec(val=c.hbarY)

    # compute intersection of c.X and c.barY
    for e in c.X:
        if e in c.barY:
            c.output(e, desc="in output set")
```

B. FACE RECOGNITION (§3.2) IN TASTYL

```
# -*- coding: utf-8 -*-
def protocol(c, s):
  K = 12       # dimension of eigenspace
  N = 10304    # number of pixels
  M = 42       # size of database

  # Declarations
  s.homegabar  = HomomorphicVec(dim=K)
  s.hgamma     = HomomorphicVec(dim=N)
  s.hD         = HomomorphicVec(dim=M)
  c.bot = Unsigned(val=M, bitlen=bitlength(M+1))
  c.gbot = Garbled(val=c.bot)

  # Client inputs
  c.gamma=UnsignedVec(bitlen=8, dim=N).input()

  # Server inputs
  s.omega = UnsignedVec(bitlen=32, dim=(M,
      K)).input()
  s.psi = UnsignedVec(bitlen=8, dim=N).input()
  s.u = SignedVec(bitlen=8, dim=(K, N)).input()
  s.tau = Unsigned(bitlen=50).input()

  # Projection
  s.hgamma <<= HomomorphicVec(val=c.gamma)
  for i in xrange(K):
    s.homegabar[i] = Homomorphic(val=-(s.u[i].\
      dot(s.psi)))+ (s.hgamma.dot(s.u[i]))

  # Distance
  s.hs3 = s.homegabar.dot(s.homegabar)
  for i in xrange(M):
    s.hD[i] = s.hs3 +
        s.omega[i].dot(s.omega[i])
    s.hD[i] += s.homegabar.dot(s.omega[i]*(-2))

  # Minimum
  c.gD <<= GarbledVec(val=s.hD,
      force_bitlen=50, force_signed=False)
  c.gDmin_val,c.gDmin_ix=c.gD.max_value_index()
  c.gtau <<= Garbled(val=s.tau)
  c.gcmp = c.gDmin_val < c.gtau
  c.gout = c.gcmp.mux(c.gbot, c.gDmin_ix)
  c.out = Unsigned(val=c.gout)
  if c.out == c.bot:
    c.output("no match found")
  else:
    c.out.output(desc="matched index in DB")
```

C. EXAMPLE: AUTOMATIC BENCHMARKING WITH TASTY

In this section we give a basic example how TASTY can be used to easily and automatically benchmark a TASTYL program for different parameter lengths. A high-level overview of TASTY's benchmarking capabilities is given in §4.2.

First, we create a new TASTYL project folder by invoking

```
> tasty_init -d our_example
```

Afterwards, we adapt the benchmarking driver and the TASTYL program to our needs as shown in Fig. 11. The benchmarking driver's `next_params` method yields the bitlengths for which the protocol should be run: $l = 1, \ldots, 80$. The TASTYL program whose performance we would like to benchmark is a protocol which chooses two random l-bit inputs and multiplies them using homomorphic encryption.

We copy the TASTY project folder (`our_example/`) to client and server and invoke TASTY on both machines (client connects to server on TCP port 9000):

```
server> tasty -sd our_example
client> tasty -cd -H server:9000 our_example
```

TASTY automatically iterates over all bitlengths returned by the benchmarking driver, invokes the TASTYL program for this bitlength and stores the measured performance benchmarks into a file (`results/costs.bin`) on the client.[9]

Finally, we adapt the post-processing script (`analyze.py`) as shown in Fig. 12 to select the costs to be drawn into the graph and run it with the measured costs:

```
client:our_example/> tasty_post analyze.py \
                                 results/costs.bin
```

The resulting graph is shown in Fig. 13. As expected, the costs are similar to HE2 in Fig. 10(b).

[9]To see a list of all costs which were measured use tasty_post −i our_example/results/costs.bin.

```
from tasty.types.driver import Driver
from tasty import utils

class BenchmarkingDriver(Driver):
  """ basic driver for benchmarking """
  def next_data_in(self):
    """ yield bitlength for each protocol run:
        l = 1, 2, ..., 80 """
    for bitlen in xrange(1, 81):
      self.params = {"l" : bitlen}
      self.client_inputs = {"x":
          utils.rand.randint(0, 2**bitlen - 1)}
      self.server_inputs = {"y":
          utils.rand.randint(0, 2**bitlen - 1)}
      yield

driver = BenchmarkingDriver() # select driver

def protocol(client, server, params):
  """ TASTYL program to multiply two unsigned
      values held by C and S using HE """

  # use parameter yielded by next_params
  LENGTH = params["l"]

  client.x =
      Unsigned(bitlen=LENGTH).input(src=driver,
      desc="x")
  server.y =
      Unsigned(bitlen=LENGTH).input(src=driver,
      desc="y")
  client.hx = Homomorphic(val=client.x)
  server.hx <<= client.hx
  server.hr = server.hx * server.y
  client.hr <<= server.hr
  client.r = Unsigned(val=client.hr)
  client.r.output(dest=driver, desc="r")
```

Figure 11: Driver and TASTYL Program (protocol.py)

```
from tasty.cost_results import extract_costs
from tasty.utils import tasty_plot
from tasty.postprocessing import *

def process_costs(cost_objs):
  """ Process measured cost objects """
  x_values = [i["params"]["l"] for i in
      cost_objs[0][0]]

  # select costs to be drawn
  y_values = extract_costs(cost_objs[0],
    # C's online time
    ("C Online","client>real>online>duration"),
    # S's online time
    ("S Online","server>real>online>duration"))

  x_label = "bitlength"  # label of x axis
  y_label = "Time in ms" # label of y axis
  graph_name = "Times for HE Multiplication"

  # draw graph into PDF file
  tasty_plot(graph_name, x_label, y_label,
      x_values, y_values, legend="inside",
      outfile="multiplication_time.pdf")
```

Figure 12: Postprocessing Script To Draw Graph (analyze.py)

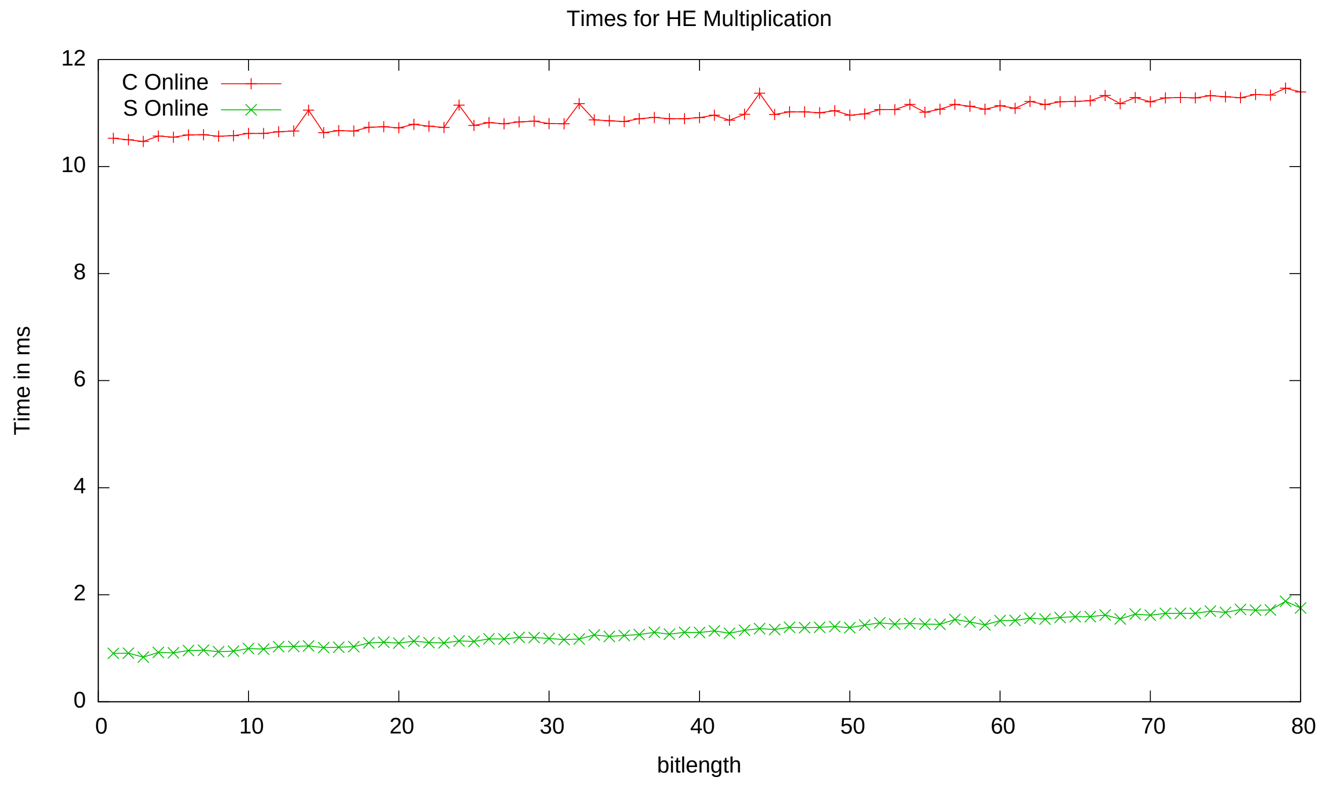


Figure 13: Graph generated by TASTY(multiplication_time.pdf)