Computationally Sound Abstraction and Verification of Secure Multi-Party Computations *

Michael Backes[1,2], Matteo Maffei[1], and Esfandiar Mohammadi[1]

1 **Saarland University**
{maffei,mohammadi}@cs.uni-saarland.de
2 **Max-Planck Institute for Software Systems**
backes@mpi-sws.org

Abstract

We devise an abstraction of secure multi-party computations in the applied π-calculus. Based on this abstraction, we propose a methodology to mechanically analyze the security of cryptographic protocols employing secure multi-party computations. We exemplify the applicability of our framework by analyzing the SIMAP sugar-beet double auction protocol. We finally study the computational soundness of our abstraction, proving that the analysis of protocols expressed in the applied π-calculus and based on our abstraction provides computational security guarantees.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.4.6 Security and Protection

Keywords and phrases Computational soundness, Secure multi-party computation, Process calculi, Protocol verification

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2010.352

1 Introduction

Proofs of security protocols are known to be error-prone and security vulnerabilities have accompanied academic protocols as well as carefully designed and widely deployed products (e.g., Kerberos [10]). Hence work towards the automation of security proofs started soon after the first protocols were developed. From the start, the actual cryptographic operations in such proofs were idealized into so-called Dolev-Yao models [18]. This idealization simplifies proof construction by freeing proofs from cryptographic details such as computational restrictions, probabilistic behavior, and error probabilities. The work on the computational soundness of Dolev-Yao models (e.g., [2, 7, 15, 14, 3])) has largely filled the gap between cryptographic abstractions and computational cryptography, showing that security properties carry over from the former to the latter.

While Dolev-Yao models traditionally comprise only non-interactive cryptographic operations (i.e., cryptographic operations that produce a single message and do not involve any form of communication, such as encryption and digital signatures), recent cryptographic protocols rely on more sophisticated *interactive primitives* (i.e., cryptographic operations that involve several message exchanges among parties), with unique features that go far

* This work was partially funded by the Cluster of Excellence "Multimodel Computing and Interaction" (German Science Foundation), the Emmy Noether Programme (German Science Foundation), the Miur'07 Project SOFT (*Security Oriented Formal Techniques*), the ERC starting grant "End-to-end security", and the DFG grant 3194/1-1.

IARCS Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010).
Editors: Kamal Lodaya, Meena Mahajan; pp. 352–363

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

beyond the traditional goals of cryptography to solely offer secrecy and authenticity of communication.

Secure multi-party computation (SMPC) constitutes arguably one of the most prominent and most amazing such primitive. Intuitively, in an SMPC, a number of parties $P_1, \ldots, P_n$ wish to securely compute the value $F(d_1, \ldots, d_n)$, for some well-known public function F, where each party P_i holds a private input d_i. This multi-party computation is considered secure if it does not divulge any information about the private inputs to other parties; more precisely, no party can learn more from the participation in the SMPC than she could learn purely from the result of the computation already.

SMPC provides solutions to various real-life problems such as e-voting, private bidding and auctions, secret sharing etc. The recent advent of efficient general-purpose implementations (e.g., FairplayMP [8]) paves the way for the deployment of SMPC into modern cryptographic protocols. Recently, the effectiveness of SMPC as a building block of large-scale and practical applications has been demonstrated by the sugar-beet double auction that took place in Denmark: The underlying cryptographic protocol [9], developed within the Secure Information Management and Processing (SIMAP) project, is based on SMPC.

Given the complexity of SMPC and its role as a building block for larger cryptographic protocols, it is important to develop abstraction techniques to reason about SMPC-based cryptographic protocols and to offer support for the automated verification of their security.

Our contributions. The contribution of this paper is threefold:

- We present an **abstraction** of SMPC within the applied π-calculus [1]. This abstraction consists of a process that receives the inputs from the parties involved in the protocol over private channels, computes the result, and sends it to the parties again over private channels, however augmented with certain details to enable computational soundness results, see below. This abstraction can be used to model and reason about larger cryptographic protocols that employ SMPC as a building block.
- Building upon an existing type-checker [4], we propose an automated **verification technique** for protocols based on our SMPC abstraction. We exemplify the applicability of our framework by analyzing the sugar-beet double auction protocol proposed in [9].
- We establish **computational soundness results** (in the sense of preservation of trace properties) for protocols built upon our abstraction of SMPC. This computational soundness result holds for SMPC that involve arbitrary arithmetic operations; moreover, it is compositional, since the proof is parametric over the other (non-interactive) cryptographic primitives used in the symbolic protocol and within the SMPC itself. Computational soundness holds as long as these primitives are shown to be computationally sound (e.g., in the CoSP framework [3]). We prove in particular the computational soundness of a Dolev-Yao model with public-key encryption, signatures, and the aforementioned arithmetic operations, leveraging and extending prior work in CoSP. Such a result allows for soundly modelling and verifying many applications employing SMPC as a building block, including the case studies considered in this paper.

Related work. Computational soundness was first shown by Abadi and Rogaway in [2] for passive adversaries and symmetric encryption and later extended to active adversaries and additional cryptographic primitives [2, 7, 15, 14, 3]. All these results, however, only consider non-interactive cryptographic primitives, such as encryptions, signatures, and non-interactive zero-knowledge proofs. To the best of our knowledge, our work presents the first computational soundness proof for an interactive primitive.

A salient approach for the abstraction of interactive cryptographic primitives is the Universal Composability framework [11]. The central idea is to define and prove the security of a protocol by comparison with an ideal trusted machine, called the ideal functionality. Although this framework has proven a convenient tool for the modular design and verification of cryptographic protocols, it is not suited to automation of security proofs, given the intricate operational semantics of the UC framework and that ideal functionalities operate on bitstrings (as opposed to symbolic terms). Dolev-Yao models (e.g., the applied π-calculus) offer a higher level of abstraction compared to ideal functionalities in the UC framework. Most importantly, Dolev-Yao models enable automation of security proofs. The different degree of abstraction in these models is best understood by considering digital signatures: While computational soundness proofs for Dolev-Yao abstractions of digital signatures use standard techniques [7, 15, 3] finding a sound ideal functionality for digital signatures has proven to be quite intricate [16]. Yet, securely realizable ideal functionalities constitute a useful tool for proving computational soundness of a Dolev-Yao model. Similarly to [12], we leverage a UC realizability result for showing the computational soundness of our symbolic abstraction.

A generic symbolic abstraction of ideal functionalities has been proposed in [17]. In that work it is shown that the different notions of simulatability, known in the literature, collapse in the symbolic abstraction. In contrast to our approach, that work does not address computational soundness guarantees and does not explicitly consider SMPC.

Outline. Section 2 reviews the applied π-calculus and presents our SMPC abstraction. Section 3 explains the technique used to statically analyze SMPC-based protocols and applies it to our case study. Section 4 presents the computational implementation of a process and studies the computational soundness of our abstraction and Section 5 concludes.

2 The symbolic abstraction of SMPC

In this section, we first review the syntax and the semantics of the calculus. We adopt a variant of the applied π-calculus with destructors [4]. After that, we present the symbolic abstraction of secure multi-party computation within this calculus.

2.1 Review of the applied π-calculus

We briefly review the syntax and the operational semantics of the applied π-calculus, and define the additional notation used in this paper.

Cryptographic messages are represented by *terms*. The set of terms (ranged over by K, L, M, and N) is the free algebra built from names (a, b, c, m, n, and k), variables (x, y, z, v, and w), and function symbols – also called *constructors* – applied to other terms ($f(M_1, \ldots, M_k)$). We let u range over both names and variables. We assume a *signature* Σ, which is a set of function symbols, each with an arity. For instance, the signature may contain the function symbols $enc_{/3}$ of arity 3 and $\mathsf{pk}_{/1}$ of arity 1, representing ciphertexts and public keys, respectively. The term $enc(M, \mathsf{pk}(K), L)$ represents the ciphertext obtained by encrypting M with key $\mathsf{pk}(K)$ and randomness L. We let $\underline{M}$ denote an arbitrary sequence $M_1, \ldots, M_n$ of terms. *Destructors* are partial functions that processes can apply to terms. Applying a destructor d to terms $\underline{M}$ either succeeds and yields a term N (denoted as $d(\underline{M}) = N$), or it fails (denoted as $d(\underline{M}) = \bot$). For instance, we may use the *dec* destructor with $dec(enc(M, \mathsf{pk}(K), L), K) = M$ to model decryption in a public-key encryption scheme.

Plain processes are defined as follows. The null process $\mathbf{0}$ does nothing and is usually omitted from process specifications; $\nu n.P$ generates a fresh name n and then behaves as P;

$a(x).P$ receives a message N from channel a and then behaves as $P\{N/x\}$; $\overline{a}\langle N\rangle.P$ outputs message N on channel a and then behaves as P; $P \mid Q$ executes P and Q in parallel; $!P$ behaves as an unbounded number of copies of P in parallel; $\mathsf{let}\ x = D\ \mathsf{then}\ P\ \mathsf{else}\ Q$ applies if $D = d(\underline{M})$ the destructor d to the terms $\underline{M}$; if application succeeds and produces the term N ($d(\underline{M}) = N$) or D equals a term N, then the process behaves as $P\{N/x\}$; otherwise, i.e., if $d(\underline{M}) = \bot$, the process behaves as Q.[1]

The scope of names and variables is delimited by restrictions, inputs, and lets. We write $fv(P)$ for the free variables and $fn(P)$ for the free names in a process P. A term is *ground* if it does not contain any variables. A process is *closed* if it does not have free variables. A *context* $C[\bullet]$ is a process with a hole $\bullet$. An evaluation context is a context whose hole is not under a replication, a conditional, an input, or an output.

The operational semantics of the applied-π calculus is defined in terms of *structural equivalence* ($\equiv$) and *internal reduction* ($\rightarrow$). Structural equivalence relates the processes that are considered equivalent up to syntactic re-arrangement. Internal reduction defines the semantics of process synchronizations and destructor applications. For more detail on the syntax and semantics of the calculus, we refer to the full version [5].

Safety properties. Following [19], we decorate security-related protocol points with logical predicates and express security requirements in terms of authorization policies. Formally, we introduce two processes $\mathtt{assume}\ F$ and $\mathtt{assert}\ F$, where F is a logical formula. Assumptions and assertions do not have any computational significance and are solely used to express security requirements. Intuitively, a process is safe if and only if all its assertions are entailed by the active assumptions in every protocol execution.

▶ **Definition 1** (Safety). A closed process P is *safe* if and only if for every F and Q such that $P \rightarrow^* \nu\underline{a}.(\mathtt{assert}\ F \mid Q)$, there exists an evaluation context $E[\bullet] = \nu\underline{b}.\bullet \mid Q'$ such that $Q \equiv E[\mathtt{assume}\ F_1 \mid \ldots \mid \mathtt{assume}\ F_n]$, $fn(F) \cap \underline{b} = \emptyset$, and we have that $\{F_1, \ldots, F_n\} \models F$.

A process is *robustly safe* if it is safe when run in parallel with an arbitrary *opponent*.

▶ **Definition 2** (Opponent). A closed process is an *opponent* if it does not contain any assert.

▶ **Definition 3** (Robust Safety). A closed process P is *robustly safe* if and only if $P \mid O$ is safe for every opponent O.

2.2 Abstracting SMPC in the applied π-calculus

We recall that a secure multi-party computation is a protocol among parties $P_1, \ldots, P_n$ to jointly compute the result of a function $\mathcal{F}$ applied to arguments $m_1, \ldots, m_n$, where m_i is a private input provided by party P_i. More generally, not only a function but a reactive, stateful computation is performed, which requires the participants to maintain a (synchronized) state. At the end of the computation, each party should not learn more than the result (or, more generally, a local view r_i of the result). Since the overall protocol may involve several secure multi-party computations, a session identifier *sid* is often used to link the private inputs to the intended session. Coming up with an abstraction of SMPC that is amenable to automated verification and that can be proven computationally sound is technically challenging, and it required us to refine the abstraction based on insights that were gained in the soundness proof. For the sake of exposition, we thus present simple, intuitive abstraction attempts of SMPC

[1] Using a destuctor `equal` that checks term equality, we write $\mathtt{if}\ a = b\ \mathtt{then}\ P\ \mathtt{else}\ Q$ for $\mathtt{let}\ \mathtt{equals}(a, b) = a\ \mathtt{in}\ \mathrm{P}\ \mathtt{else}\ \mathrm{Q}$.

in the applied π-calculus first, explain why these attempts do not allow for a computational soundness result, and then successively refine them until we reach the final abstraction.

First attempt. A first, naive attempt to symbolically abstract SMPC in the applied π-calculus is to let parties send to each other the public information along with the enveloped private input on a private channel. This message can be represented by a term $\mathtt{smpc}(F, i, m, sid)$, where i is the principal's identifier. The abstraction then consists of a destructor `result` whose semantics is defined by a rule like

$$\mathtt{result}(m_i, i, \mathtt{smpc}(\mathcal{F}, 1, m_1, sid), \ldots, \mathtt{smpc}(\mathcal{F}, n, m_n, sid)) = \pi(i, \mathcal{F}(m_1, \ldots, m_n))$$

where $\pi(i, \cdot)$ denotes the projection on the i-th element. This abstraction is unsound: a computational attacker (i) is capable of altering the delivery of messages, and (ii) learns the session identifiers that occur in the header of each individual message. Our abstraction attempt does not grant the adversary any such capabilities; hence a symbolic adversary is much stronger constrained than a computational adversary, thus preventing computational soundness results. These problems could be tackled by modifying the abstraction such that all messages are sent and received over a public channel. The adversary would then decide which parties receive which messages. The resulting abstraction, however, would not be tight enough anymore: Corrupted symbolic parties could send different messages to the participants, and hence cause them to compute the function $\mathcal{F}$ on different inputs. Such an attack, however, is computationally excluded by a secure multi-party computation protocol.

Second attempt. Inspired by the ideal functionality paradigm [11], we solve the aforementioned problems by introducing a trusted party to whom every participant i sends its private input and receives its own result in return over a private channel in_i. A first attempt, called **SMPC_temp**, could look as follows:

$$\begin{aligned}\textbf{SMPC\_temp} \triangleq\ & sidc(sid).in_1(x_1, = sid) \ldots in_n(x_n, = sid). \\ & \mathtt{let}\ (y_1, \ldots, y_n) = \mathtt{result}(\mathcal{F}, x_1, \ldots, x_n)\ \mathtt{in}\ \overline{in_1}\langle y_1, sid\rangle \ldots \overline{in_n}\langle y_n, sid\rangle\end{aligned}$$

There still is discrepancy between this abstraction and the computational model that invalidates computational soundness: a computational adversary learns the session identifier, which is instead concealed by **SMPC_temp**. In addition, the computation of $\mathcal{F}$ is shifted to the evaluation of the destructor `result`. Such a complicated destructor would make mechanized verification extremely difficult.

Final abstraction. To make the abstraction amenable to automated verification, in particular type-checking, we represent $\mathcal{F}$ as a context that explicitly performs the computation. The resulting abstraction of secure multi-party computation is depicted in Figure 1 as the process **SMPC**. This process is parametrized by an adversary channel adv, a session identifier channel $sidc$, n private channels in_i for each of the n participants, and a context $\mathcal{F}$. We implicitly assume that private channels are authenticated such that only the ith participant can send messages on channel in_i. The computational implementation of SMPC implements this authentication requirement. Furthermore, **SMPC** contains two restricted channels for every party i: an internal loop channel $inloop_i$ and an internal input channel lin_i.

SMPC receives a session identifier over the channel $sidc$. Then $n + 1$ subprocesses are spawned: a process $\mathbf{input}_i$ for every participant i that is responsible for collecting the ith input and for divulging public information, such as the session identifier, to the adversary, and a process that performs the actual multi-party computation. Here $\mathbf{input}_i$ waits (under a replication) on the loop channel $inloop_i$ for the trigger message `sync`() of the next round, and

$$
\begin{aligned}
\mathbf{input}_i &\triangleq !\mathit{inloop}_i(z).\mathit{in}_i(x_i, \mathit{sid}').\overline{\mathit{adv}}\langle \mathit{sid}'\rangle.\mathtt{if}\ \ \mathit{sid} = \mathit{sid}' \\
&\quad \mathtt{then}\ \overline{\mathit{lin}_i}\langle x_i\rangle\ \mathtt{else}\ \overline{\mathit{inloop}_i}\langle \mathtt{sync()}\rangle \mid \overline{\mathit{inloop}_i}\langle \mathtt{sync()}\rangle \\
\mathbf{deliver}_i &\triangleq \overline{\mathit{in}_i}\langle y_i, \mathit{sid}\rangle.\overline{\mathit{inloop}_i}\langle \mathtt{sync()}\rangle \\
\mathbf{SMPC}(\mathit{adv}, \mathit{sidc}, \underline{\mathit{in}}, \mathcal{F}) &\triangleq \mathit{sidc}(\mathit{sid}).\nu \underline{\mathit{lin}}.\ \nu \overline{\mathit{inloop}}. \\
&\quad \big(\mathbf{input}_1 \mid \cdots \mid\ \mathbf{input}_n \mid \mathcal{F}[\mathbf{deliver}_1 \mid \cdots \mid \mathbf{deliver}_n]\big)
\end{aligned}
$$

Figure 1 The process **SMPC** as the symbolic abstraction of SMPC

expects the private input x_i and a session identifier sid' over in_i. It then sends the session identifier sid' to the adversary, checks whether the session identifier sid' equals sid, and finally sends the private input x_i on the internal input channel lin_i. The actual multi-party computation is performed in the last subprocess: after the private inputs of the individual parties are collected from the internal input channels lin_i, the actual program $\mathcal{F}$ is executed.

After each computation round, the subprocesses $\mathbf{deliver}_i$ send the individual outputs over the private channels in_i to every participant i along with the session identifier sid. In order to trigger the next round, $\mathtt{sync}()$ is sent over the internal loop channels inloop_i.

The abstraction allows for a large class of functionalities $\mathcal{F}$ as described below.

▶ **Definition 4.** [SMPC-suited context] An *SMPC-suited context* is a context $\mathcal{F}$ such that:

1. $\mathit{fv}(\mathcal{F}) = \{\mathit{sid}\}$ and $\mathit{fn}(\mathcal{F}[\mathbf{0}]) = \{\mathit{lin}_1, \ldots, \mathit{lin}_n\}$.
2. Bound names and variables are distinct and different from free names and free variables.

Although one might expect additional constraints on the context $\mathcal{F}$ (e.g., it is terminating, it does not contain replications, etc.), it turns out that such constraints are not necessary as, intuitively, having more traces in the symbolic setting does not break computational soundness. Such constraints would probably simplify the proof of computational soundness, but they would make our abstraction less intuitive and less general.

Finally, we briefly describe how we model the corruption of the parties involved in the secure multi-party computation. In this paper we consider static corruption scenarios, in which the parties to be corrupted are selected before the computation starts. As usual in the applied π-calculus, we model corrupted parties by letting the adversary know the secret inputs of the corrupted parties. This is achieved by letting the channel in_i occur free in the process if party i is corrupted. As we consider static corruption, we restrict our attention to processes that do not send these channels in_i over a public channel (see Section 4).

Arithmetics in the applied π-calculus. One of the most common applications of SMPC is the evaluation of arithmetic operations on secret inputs. Modelling arithmetic operations in the applied pi-calculus is straightforward. We encode numbers in binary form via the $\mathit{string}_0(M)$, $\mathit{string}_1(M)$, and $\mathit{nil}()$ constructor applications. Arithmetic operations are modelled as destructors. For instance, the greater-equal relation is defined by the destructor $\mathtt{ge}(M_1, M_2)$, which returns M_1 if M_1 is greater equal then M_2, M_2 otherwise. With this encoding, numbers and cryptographic messages are disjoint sets of values, which is crucial for the soundness of our analysis and the computational soundness results.

The Millionaires problem. For the sake of illustration, we show how to express the Millionaires problem in our formalism (two parties wish to determine who is richer, i.e., whose

input is bigger, without divulging their inputs to each other.) The protocol is parametrized by two numbers x_1 and x_2:

$$\begin{aligned} MP &\triangleq \nu sid, sidc, c_1, c_2.\ \overline{adv}\langle sid\rangle.\ \overline{sidc}\langle sid\rangle \mid \mathbf{SMPC}(adv, sidc, \underline{c}, \mathcal{F}) \mid P_1 \mid P_2 \\ P_i &\triangleq \overline{c_i}\langle (x_i, id_i), sid\rangle.c_i(y_i, sid_i) \\ \mathcal{F}[\bullet] &\triangleq lin_1((x_1, z_1)).lin_2((x_2, z_2)).\texttt{let}\ \ z = \mathtt{ge}(x_1, x_2)\ \ \texttt{then}\ \ [y_1 := z, y_2 := z]\bullet \end{aligned}$$

where $[y_1 := z, y_2 := z]$ denotes the instantiation of variables y_1 and y_2 with variable z, which can be defined by encoding, and $\bullet$ is the hole of the context.

3 Formal verification

We propose a technique for formally verifying processes that use the symbolic abstraction **SMPC**. In principle, our abstraction is amenable to several verification techniques for the applied π-calculus such as [19, 4]. In this paper, we rely on the type-checker for security policies presented in [4], which we extend to support arithmetic operations. This type-checker enforces the robust safety property (cf. the full version [5]).

We decorate the protocol linked to our SMPC abstraction with assumptions and assertions. The assumptions are used to mark the inputs of the secure multi-party computation and to specify the correctness property for the secure multi-party computation. The assertion is used to check that the result of the SMPC fulfills the correctness property.

Specifically, for every party i an assumption `assume` $\mathrm{Input}(id_i, x_i, sid)$ is placed upon sending a private input x_i with session identifier sid, where id_i is the (publicly known) identity of party i. To check the correctness of the secure multi-party computation, `assert` $\mathrm{P}(z, sid)$ is placed immediately after the reception of the result (z, sid). We also assume a security policy, which takes in general the following form:

$$\forall \underline{id}, \underline{x}, sid, z.\big(\wedge_{i=1}^{n}\mathrm{Input}(id_i, x_i, sid) \bigwedge_{i,j\in[n]} id_i \neq id_j \wedge F_{rel}\big) \Rightarrow \mathrm{P}(z, sid)$$

The formula F_{rel} characterizes the expected relation between the inputs and the output. As an example, the policy and party annotations for the Millionaires problem would be as follows:

$$\forall id_1, id_2, x, y, sid.\Big(\bigwedge_{i\in\{1,2\}}\mathrm{Input}(id_i, x, sid) \wedge id_1 \neq id_2 \wedge x \geq y\Big) \Rightarrow \mathrm{Richer}(id_1, sid).$$

$$P_i \triangleq \texttt{assume}\ \mathrm{Input}(id_i, x_i, sid) \mid \overline{c_i}\langle (x_i, id_i), sid\rangle.c_i(y_i, sid_i).\texttt{assert}\ \mathrm{Richer}(y_i, sid).$$

Arithmetics in the analysis. We extended the type-checker to support arithmetic operations. Specifically, we modelled arithmetic operations as predicates in the logic, defined their semantics following the semantics given in the calculus, and added a few general properties, such as the transitivity of the greater-equal relation. The type theory is extended to track the arithmetic properties of terms (e.g., while typing $\texttt{let}\ \ z = \mathtt{ge}(x_1, x_2)\ \ \texttt{then}\ \ P$, the type-checker tracks that z is the greatest value between x_1 and x_2 and uses this information to type P). The type theory supports this kind of extensions as long as the set of added values is disjoint from the set of cryptographic messages and the added destructors do not operate on cryptographic terms, which holds true for our encoding of arithmetics.

Case study: sugar-beet double auction. As a case study for our symbolic abstraction, we formalized and analyzed the sugar-beet double auction that has been realized within

the SIMAP project by using an SMPC [9]. This protocol constitutes the first large scale application of an SMPC. The double auction protocol determines a market clearing price for sugar-beets. More specifically, first, a set of prices is fixed; then, for every price each producer commits itself to an amount of sugar-beets that it is willing to sell, and each buyer commits itself to the amounts of sugar-beets that it is willing to buy. The market clearing price is the maximal market clearing price for which the supply did not yet exceed the demand.

Both the producers and the buyers might want to keep their bids secret. Hence, the private input of every party has to be kept private. In the sugar-beet double auction developed by the SIMAP project, the sellers and buyers perform a joint secure multi-party computation. The producer and buyer parties send initially an input and receive at the end of the computation the result, i.e., the market clearing price. In addition, before the start of the protocol, producers and buyers receive a signature on the list of participants, the session identifier, and the set of prices from a trusted party. This signature is verified by each participant and sent as an input to the SMPC, which verifies the signatures and checks whether the data coincides This ensures that all participants agree on the common public inputs. Notice that this SMPC performs arithmetic operations as well as cryptographic operations, yet on distinct values.

Finally, we are ready to state the policy characterizing the result of the computation that is performed: the predicate $\mathrm{MCP}(z, sid)$ holds true if there are appropriate input predicates $\mathrm{Input}(id_i, x_i, sid)$ and z is the maximal price for which demand is greater than or equal to the supply, which we characterize by the predicate $\mathrm{Is\_max}(x_1, \ldots, x_n, z)$ (the semantics of this predicate is defined in terms of basic arithmetic operations supported by our type-checker).

$$\forall z, sid, x_1, \ldots, x_m, id_1, \ldots, id_n.\mathrm{Input}(id_1, x_1, sid) \wedge \cdots \wedge \mathrm{Input}(id_n, x_n, sid) \bigwedge_{i,j \in [n]} id_i \neq id_j \wedge \mathrm{Is\_max}(x_1, \ldots, x_m, z) \Rightarrow \mathrm{MCP}(z, sid)$$

The verification of this policy is challenging in that our abstraction comprises about 1400 lines of code and it relies on complex functions. The type-checker succeeds in 5 minutes and 30 seconds for the SMPC process and the ceritifcation issuer and additional 40 seconds for every participant. [2]

4 Computational soundness of symbolic SMPC

In this section, we establish the computational soundness of our abstraction. We first introduce the computational implementation of a process. Thereafter, we define the notion of robust computational safety. Finally, we state the computational soundness results.

Our computational soundness result is parameterized over the symbolic model $(\mathcal{D}, \mathcal{P})$; this symbolic model consists of a set of constructors and destructors $\mathcal{D}$ (modelling non-interactive primitives such as encryption and decryption) and a class $\mathcal{P}$ of processes.

Computational execution of a process. The semantics of the applied π-calculus is purely symbolic (i.e., it does not involve probabilities, cryptographic messages are represented as symbolic terms instead of bitstrings, the adversary is not computational, etc.). Along the lines of [3], we introduce a probabilistic polynomial-time interactive Turing machine (ITM), called the *computational π-execution*, that interacts with a ppt ITM, called the adversary.

[2] The source code can be found at `http://www.lbs.cs.uni-saarland.de/publications/smpc_simap.spi`. We type checked the protocol with 2 prices, 3 computation parties, and 2 input parties.

The computational π-execution expects as input a process P and a security parameter k. We summarize the main properties of this ITM. It enforces the reduction rules of the operational semantics, draws a random bitstring for each fresh nonce (i.e., restricted name), and executes each constructor and destructor application $d(M)$ (for $d \in \mathcal{D}$) as the computation of the polynomial-time algorithm A_d on input M; A_d is called the implementation of d and $\underline{A}$ is the family of all implementations A_d ($d \in \mathcal{D}$). In the operational semantics of the applied π-calculus, the reduction rules are non-deterministic. This non-determinism is resolved in the implementation by letting the adversary select the reduction steps: The computational execution sends the current process to the adversary and receives back a message indicating the next instruction to be executed. Whenever the execution encounters an `assert` F, it stores a tuple of the form $(F_1, \ldots, F_n, F, \eta, \mu, P)$, called an *assertion tuple*, where F_i are the active assumptions of the current process and η and μ are mappings assigning a bitstring to each variable and name, respectively. We denote the interaction between the computational π-execution using the implementations $\underline{A}$ and an adversary $\mathcal{A}dv$ as $\textsc{Exec}^{\pi}_{P,\underline{A},\mathcal{A}dv}(1^k)$. Given a polynomial p, the distribution of sequences of assertion tuples raised in an interaction of $\textsc{Exec}^{\pi}_{P,\underline{A},\mathcal{A}dv}(1^k)$ within the first $p(k)$ computation steps is denoted as $\mathit{Assertions}^{\pi}_{P,\underline{A},p,\mathcal{A}dv}(k)$.

Our computational π-execution is close in spirit to previously proposed computational executions for the applied π-calculus (e.g., [3]). Such an implementation, however, treats each $\mathbf{SMPC}(\mathit{adv}, \mathit{sidc}, \underline{\mathit{in}}, \mathcal{F})$ abstraction as a trusted host performing a computation and sharing a private channel with each party, whereas the final implementation runs a distributed SMPC protocol. We thus refine the computational π-execution by letting the actual distributed SMPC protocol be executed in place of the (implementation of the) $\mathbf{SMPC}(\mathit{adv}, \mathit{sidc}, \underline{\mathit{in}}, \mathcal{F})$ abstraction. The resulting computational execution $\textsc{Exec}^{\text{SMPC}}_{P,\underline{A},\tau,\mathcal{A}dv}(1^k)$, called *computational SMPC-execution*, is parameterized by the family τ of SMPC protocols implementing each $\mathbf{SMPC}(\mathit{adv}, \mathit{sidc}, \underline{\mathit{in}}, \mathcal{F})$ abstraction. $\mathit{Assertions}^{\text{SMPC}}_{P,\underline{A},\tau,p,\mathcal{A}dv}(k)$ is defined analogous to $\mathit{Assertions}^{\pi}_{P,\underline{A},p,\mathcal{A}dv}(k)$ (see the full version [5] for the formal definitions).

Robust computational safety. The computational notion of robust safety depends on the computational notion of logical entailment. A major difference between the standard symbolic entailment relation and the computational entailment relation is that while the former models destructor application tests $d(\underline{M}) = N$ via logical predicates of the form $\mathrm{Red}(d^{\#}(\underline{M}), N)$ (whose semantics follows from the symbolic rules of the applied π-calculus), the latter computationally checks $A_d(\tilde{\underline{M}}) = \tilde{N}$, $\tilde{\underline{M}}$ and $\tilde{N}$ being the bitstrings computed for $\underline{M}$ and N using η, μ, and $\underline{A}$. Thus, we denote the computational entailment relation as $\models_{\eta,\mu,\underline{A}}$. We now introduce two definitions of robust computational safety, with respect to $\textsc{Exec}^{\pi}$ and $\textsc{Exec}^{\text{SMPC}}$, respectively.

▶ **Definition 5** (Robust computational safety)**.** Let P be a process, $\underline{A}$ an implementation of the destructors in P, and τ a family of secure multi-party computations. We say that P is π-(resp. SMPC-)robustly computationally safe using $\underline{A}$ (resp. $\underline{A}, \tau$) iff for all polynomial-time interactive machines $\mathcal{A}dv$ and all polynomials p, $\Pr[\text{for all } ((F_1, \ldots, F_n), F, \eta, \mu, Q) \in a, \{F_1, \ldots, F_n\} \models_{\eta,\mu,\underline{A}} F : a \leftarrow \mathit{Assertions}^{\pi}_{P,\underline{A},p,\mathcal{A}dv}(k)]$ (resp. $\Pr[\text{for all } ((F_1, \ldots, F_n), F, \eta, \mu, Q) \in a, \{F_1, \ldots, F_n\} \models_{\eta,\mu,\underline{A}} F : a \leftarrow \mathit{Assertions}^{\text{SMPC}}_{P,\underline{A},\tau,p,\mathcal{A}dv}(k)]$) is overwhelming in k.

A symbolic model is computationally sound if robust safety carries over to the computational setting. This definition is used in our first theorem, which is parameterized over the non-interactive primitives used in the protocol.

▶ **Definition 6** (Computationally sound model)**.** Let $\underline{A}$ be a set of constructor and destructor implementations. We say that a symbolic model $(\mathcal{D}, \mathcal{P})$ is computationally sound using $\underline{A}$ iff for all $P \in \mathcal{P}$ such that P is robustly safe, P is π-robustly computationally safe using $\underline{A}$.

There are properties for which computational soundness cannot be shown. For a function H whose range is smaller than its domain (such as a collision-resistant hash function), consider the injectivity property $\forall x, y.x \neq y \Rightarrow H(x) \neq H(y)$. Symbolically, this property holds true if H is a constructor. Computationally, however, this formula naturally does not hold as x and y are universally quantified and collisions cannot be avoided. To exclude such cases, we only consider quantification over protocol messages. This is formalized by requiring that all quantified subformulas $\forall x.F$ and $\exists x.F$ are of the form $\forall x.p(\ldots, x, \ldots) \Rightarrow F'$ and $\exists x.p(\ldots, x, \ldots). \wedge F'$ (where p is a predicate) and we call the resulting class of first-order formulas *well-formed formulas* (see the technical report [5] for a formal definition). Hence $\forall x, y.x \neq y \Rightarrow H(x) \neq H(y)$ is not well-formed. Instead, $\forall x, y.p(x) \wedge p(y) \wedge x \neq y \Rightarrow H(x) \neq H(y)$[3] (where p is a predicate, meant to be assumed upon reception of x and y) is well-formed. As we exclude assumptions of the form $\forall x.p(x)$, $p(m)$ has to be assumed explicitly and m must be a protocol message. For a collision-resistant hash function H, the above formula holds computationally. We stress that in contrast to other computational soundness results (e.g., [3]), where formulas are assumed to be term-free (i.e., contain nullary predicates only), our result holds for formulas containing terms of the calculus and destructor application tests.

In addition we require that (i) restricted channels in_i are never output; (ii) session identifiers are sent at most once on session identifier channels; and (iii) for all subprocesses $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ of P, the context $\mathcal{F}$ is SMPC-suited. Processes fulfilling this constraints are called *well-formed processes*. A *well-formed class of processes* only contains well-formed processes and enjoys some closeness properties, such as closeness under subprocesses or top-level name restrictions (we refer to the full version [5] for more details).

Computational soundness of symbolic SMPC. We now state the main computational soundness result of this work: the robust safety of a process using non-interactive primitives and our SMPC abstraction carries over to the computational setting, as long as the non-interactive primitives are computationally sound. This result ensures that the verification technique from Section 3 provides computational safety guarantees. We stress that the non-interactive primitives can be used both within the SMPC abstractions and within the surrounding protocol. The proof uses a result from [13] on generic UC-realizable MPC protocols that only holds under standard cryptographic assumptions: the setup assumption of the *CRS-model*[4] and the existence of *enhanced trapdoor permutations*. Moreover, as also required in [3], all implementations $\underline{A}$ have to be *length-regular*.

▶ **Theorem 7** (Computational soundness of symbolic SMPC). *Let $(\mathcal{D}, \mathcal{P})$ be computationally sound, well-formed model using $\underline{A}$, where $\underline{A}$ is length-regular. If enhanced trapdoor permutations exist, then there is a family ρ of SMPC implementations in the CRS-model such that for each well-formed $P \in \mathcal{P}$, the robust safety of P implies the SMPC-robust computational safety of P using $\underline{A}, \rho$.*

Computational soundness of arithmetic operations. We show that our computational soundness result applies to a large class of protocols by proving the computational soundness of a symbolic model with public-key encryption, signatures, and arithmetics[5]. We recall that

[3] This formula is logically equivalent to $\forall x.p(x) \Rightarrow \forall y.p(y) \Rightarrow x \neq y \Rightarrow H(x) \neq H(y)$. For the sake of a clear presentation, the policies in Section 3 are not well-formed, but adding for every quantified variable x a predicate $p(x)$ to the premise makes these policies well-formed as well.

[4] More precisely, in each SMPC session all parties have access to a shared bitstring, called the CRS.

[5] This result extends prior work [3] in the CoSP framework to arithmetics and first-order logic formulas.

numbers are modelled as symbolic bitstrings and arithmetic operations as destructors on these bitstrings. In this way, we enforce a strict separation between numbers and cryptographic terms, such as signatures and keys. Thus, it is not possible to apply an arithmetic operation on nonces, signatures, or encrypted messages. Due to this separation, the impossibility result for computational soundness of XOR [6] does not apply. As in [3], we impose some standard conditions on protocols to ensure that all encryptions and signatures are produced using fresh randomness and that secret keys are not sent around. A protocol satisfying these conditions is called *key-safe*. We denote the resulting symbolic model as $(\mathcal{D}_{\mathrm{ESA}}, \mathcal{P}_{\mathrm{ESA}})$.

The computational soundness proof for $(\mathcal{D}_{\mathrm{ESA}}, \mathcal{P}_{\mathrm{ESA}})$ follows almost exactly the lines of the symbolic model for *key-safe* protocols in the work of Backes, Hofheinz, and Unruh [3]. The proof can be found in the full version [5].

▶ **Theorem 8** (Computational soundness of $(\mathcal{D}_{\mathsf{ESA}}, \mathcal{P}_{\mathsf{ESA}})$)**.** *If enhanced trapdoor permutations exist, there is an length-regular implementation $\underline{A}$ such that $(\mathcal{D}_{ESA}, \mathcal{P}_{ESA})$ is a computationally sound, well-formed model.*

As a corollary of Section 7 and Theorem 8, we get the computational soundness of protocols using SMPC, public-key encryption, signatures, and arithmetics, with such non-interactive cryptographic primitives possibly used within both SMPC and the surrounding protocol.

▶ **Corollary 9** (SMPC computational soundness with $(\mathcal{D}_{\mathsf{ESA}}, \mathcal{P}_{\mathsf{ESA}})$)**.** *There is an implementation $\underline{A}$ and a family of UC-protocols τ such that for each well-formed $P \in \mathcal{P}_{ESA}$, the robust safety of P implies the SMPC-robust computational safety of P using $\underline{A}$ and τ.*

5 Conclusion

We have presented an abstraction of SMPC in the applied π-calculus. We have shown that the security of protocols based on this abstraction can be automatically verified using a type system, and we have established computational soundness results for this abstraction including SMPC that involve arbirtrary arithmetic operations. This is the first work to tackle the abstraction, verification, and computational soundness of protocols based on an interactive cryptographic primitive.

Our framework allows for the verification of protocols incorporating SMPC as a building block. In particular, the type-checker ensures that the inputs provided by the participants to the SMPC are well-formed and, after verifying the correctness of SMPC, the type-checker obtains a characterization of the outputs (e.g., in the Millionaires problem, the output is the identity of the participant providing the greatest input) that can be used to establish global properties of the overall protocol. Our framework is general and covers SMPC based on arithmetic operations as well as on cryptographic primitives.

This work focuses on trace properties, which include authenticity, integrity, authorization policies, and weak secrecy (i.e., the attacker cannot compute a certain value)[6]. As a future work, we plan to extend our framework to observational equivalence relations, possibly building on recent results for a fragment of the applied π-calculus [14]. It would be interesting to formulate and verify an indistinguishability-based secrecy property for SMPC. This task is particularly challenging since the result of an SMPC typically reveals some information about the secret inputs and standard secrecy definitions based on observational equivalence are

[6] To symbolically model weak secrecy, one can add the process c(x).if x=n then assert false (where c is a public channel) to check the secrecy of n: The protocol is robustly safe only if the attacker does not learn n.

thus too restrictive. An additional problem is the computational soundness of observational equivalence for processes with private channels, which are excluded in [14] but are needed in our abstraction and, arguably, in any reasonable SMPC abstraction.

References

1 M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115. ACM Press, 2001.

2 Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

3 Michael Backes, Dennis Hofheinz, and Dominique Unruh. CoSP: A general framework for computational soundness proofs. In *CCS*, pages 66–78. ACM Press, 2009.

4 Michael Backes, Catalin Hritcu, and Matteo Maffei. Type-checking zero-knowledge. In *CCS*, pages 357–370, 2008.

5 Michael Backes, Matteo Maffei, and Esfandiar Mohammadi. Computationally sound abstraction and verification of secure multi-party computations. Available at: `http://www.infesc.cs.uni-saarland.de/~mohammadi/publications/smpc.pdf`.

6 Michael Backes and Birgit Pfitzmann. Limits of the cryptographic realization of Dolev-Yao-style XOR. In *ESORICS*, volume 3679 of *LNCS*, pages 178–196. Springer, 2005.

7 Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations (extended abstract). In *CCS*, pages 220–230, 2003.

8 Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS*, pages 257–266. ACM, 2008.

9 Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography*, volume 5628 of *LNCS*, pages 325–343. Springer, 2009.

10 F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal analysis of Kerberos 5. *Theoretical Computer Science*, 367(1):57–87, 2006.

11 Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

12 Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key exchange protocols. In *TCC*, volume 3876 of *LNCS*, pages 380–403. Springer, 2006.

13 Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503. ACM, 2002.

14 Hubert Comon-Lundh and Véronique Cortier. Computational soundness of observational equivalence. In *CCS*, pages 109–118, 2008.

15 Véronique Cortier and Bogdan Warinschi. Computationally sound, automated proofs for security protocols. In *ESOP*, pages 157–171, 2005.

16 Anupam Datta, Ante Derek, John Mitchell, Ajith Ramanathan, and Andre Scedrov. Games and the impossibility of realizable ideal functionality. In *TCC*. Springer, 2006.

17 Stéphanie Delaune, Steve Kremer, and Olivier Pereira. Simulation based security in the applied pi calculus. In *FSTTCS*, pages 169–180. Schloss Dagstuhl, 2009.

18 Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

19 C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *CSF*, pages 31–45. IEEE, 2007.