© ACM, 2003. This is the authors' version of the work. It is posted here by permission of ACM for your personal use.
Not for redistribution. The definitive version is available at http://doi.acm.org/10.1145/948109.948139.

Automatic Generation of Two-Party Computations

(Extended Abstract)

Philip MacKenzie* Alina Oprea† Michael K. Reiter‡

ABSTRACT

We present the design and implementation of a compiler that automatically generates protocols that perform two-party computations. The input to our protocol is the specification of a computation with secret inputs (e.g., a signature algorithm) expressed using operations in the field $\mathbb{Z}_q$ of integers modulo a prime q and in the multiplicative subgroup of order q in $\mathbb{Z}_p^*$ for $q|p-1$ with generator g. The output of our compiler is an implementation of each party in a two-party protocol to perform the same computation securely, i.e., so that both parties can together compute the function but neither can alone. The protocols generated by our compiler are provably secure, in that their strength can be reduced to that of the original cryptographic computation via simulation arguments. Our compiler can be applied to various cryptographic primitives (e.g., signature schemes, encryption schemes, oblivious transfer protocols) and other protocols that employ a trusted party (e.g., key retrieval, key distribution).

Categories and Subject Descriptors

E.3 [**Data Encryption**]: public key cryptosystems

General Terms

Security

Keywords

Secure two-party computation, automatic generation of protocols, threshold cryptography

*Bell Labs, Lucent Technologies, Murray Hill, NJ, USA; philmac@research.bell-labs.com
†Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA; alina@cs.cmu.edu
‡Electrical & Computer Engineering and Computer Science Departments, Carnegie Mellon University, Pittsburgh, PA, USA; reiter@cmu.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'03, October 27–31, 2003, Washington, DC, USA.
Copyright 2003 ACM 1-58113-738-9/03/0010 ...$5.00.

1. INTRODUCTION

A central notion in cryptographic key management is that of (k,n)-*secret sharing* [32], by which a secret key is divided into n *shares* with the property that only sets of shares of size k or larger can be used to reconstruct the key. During the past fifteen years, (k,n)-secret sharing has been extended to (k,n)-*function sharing* (or *threshold cryptography*) (e.g., [9, 13, 14, 17]), whereby k parties, each holding a share, collaborate to perform a computation with the key without exposing their shares (or reconstructing the key). A special case of interest in this paper is two-party (i.e., $(2,2)$) function sharing, in which the key is split between two parties whose collaboration is necessary and sufficient to compute the function with the key.

In this paper we present the design and implementation of a compiler for the automatic generation of two-party function sharing protocols. Our work is primarily motivated by the current gap between generic secure two-party computation protocols and efficient hand-tuned two-party protocols for particular primitives. We propose the first fully automated tool for generating two-party protocols that are efficient enough to be used in practical applications. The input to our compiler is a description of a cryptographic function (e.g., a digital signature algorithm, or decryption algorithm) written in a high-level language of our own design. The output of our compiler is source code implementing each side of a two party protocol for computing that function. At the time of this writing, the input functions for which our compiler can generate two-party protocols are restricted to functions in which secrets and randomly generated values (e.g., permanent and ephemeral keys) are elements of the field $\mathbb{Z}_q$ of integers modulo a public prime q, and that are comprised of field operations in $\mathbb{Z}_q$ and operations in the multiplicative subgroup G_q of order q in $\mathbb{Z}_p^*$ with public generator g and public prime p, $q|p-1$. Computations on public values (e.g., hashing a message in the course of generating a digital signature) are also supported. These operations suffice for numerous useful cryptographic algorithms, including various public key cryptosystems (e.g., [10]) and signature schemes (see [20] for a survey) based on the difficulty of computing discrete logarithms in G_q. An area of ongoing work is extending our protocol to broader sets of computations.

The protocols generated by our compiler are provably secure against arbitrary misbehavior by either party. More specifically, the security of a protocol generated by our compiler can be reduced to the security of the cryptographic function that is the input to the compiler, using a simulation argument. Se-

curity additionally relies on other assumptions, which in our present implementation includes both random oracles [4] and cryptographic assumptions such as decision composite residuosity [29]. Further, for any function with a secret input parameter (typically a long-term private key), the two-party protocol output from our compiler requires that shares of this secret parameter be predistributed to the two parties that are to run the protocol. That is, our compiler generates two party protocols that assume a trusted third party for sharing long-term secrets; removing this assumption is an area of ongoing work.

Another motivation of our work is the prior research toward preventing misuse of a cryptographic key by requiring the consent of a partially [21, 8] or fully [16] trusted server before the key can be used. A common approach to make the server non-bypassable is to share the key between its intended user and the server, and for the user and server to jointly compute cryptographic operations using $(2, 2)$-function sharing. A goal of this research is to provide a tool by which a new cryptographic function (e.g., any of the literally hundreds of signature schemes in [20]) can be protected in this sense through the automatic, and even dynamic, generation of a server-assisted protocol for the function. More generally, we intend to explore compilation as a means for making two-party (and, in the future, multiparty) computation a "commodity tool" for other cryptographic applications and scientific research.

A precursor to our compilation process is the assembly of a repertoire of "building block" protocols for basic two-party computations, e.g., addition of shared secrets (yielding a shared result), multiplication of shared secrets, multiplicative inversion of a shared secret, and so forth. Additional building blocks are included to translate shares in one representation to another, i.e., from additive shares to multiplicative shares and back. These translations are required since, for example, the most efficient protocol for adding shared secrets employs additive shares, whereas the most efficient protocol for inverting a shared secret employs multiplicative shares. The compilation process itself begins with parsing the input function into a tree with nodes being operations and edges representing data flows between operations. In addition, each edge is annotated with an indication of whether the value on that edge is to be public or kept secret. Additional nodes are inserted to translate shared secrets in one representation to the other as necessary, and to reveal (i.e., make public) values that are allowed to be revealed. Finally, the tree is traversed to emit building block protocols that together perform a two-party computation of the input function.

2. RELATED WORK

Techniques for generic secure two-party computation (e.g., [35]) provide a recipe for designing secure two-party protocols for a much larger class of functions than our compiler can accommodate. However, the protocols that result from these general techniques are too inefficient to be useful in practice, and we know of no efforts to implement these techniques (or more specialized techniques) within a tool for generating protocol implementations automatically. The goal of our compiler is to generate practical protocols for a restricted but interesting class of functions, and to explore optimizations that make them useful even in performance-sensitive applications.

As discussed in Section 1, we are primarily motivated by the generation of two-party protocols for computing digital signatures or decrypting ciphertexts of a public key cryptosystem. As such, the protocols our compiler generates fall within the general framework of *threshold cryptography*, which explores specialized protocols for implementing shared signing or decryption more efficiently than via general secure multiparty computation (e.g., [9, 13, 14, 17]). Within this category of work, proposed two-party protocols include those for performing two-party RSA computations [9] (see also [3]), two-party DSA [22] and two-party Schnorr [25], for example. At the time of this writing, our compiler can be used to *automatically* generate provably secure protocols for two-party DSA and Schnorr signing, albeit at some loss in performance over prior hand-tuned proposals, and for all other discrete-log based signature schemes of which we are aware (notably the hundreds of variations surveyed in [20]). In the future, we plan to extend our compiler to automatically generate two-party protocols for RSA-based operations (e.g., [31, 11]).

Implementations of specific threshold cryptosystems and signature schemes have been developed in the context of several systems (e.g., [30, 36], and specifically [21, 8, 25] for the two-party case) and have been built into general-purpose toolkits for broader use [34, 1]. We know of no such toolkit for two-party threshold cryptography, much less any prior work on compilers for automatically generating two-party protocols.

More distantly related is work in the automatic generation of protocols for establishing a shared cryptographic key between two parties, possibly with the assistance of a third party (e.g. [33]). We emphasize that this work involves *key exchange* protocols using cryptographic primitives (encryption, signatures) as abstract building blocks, without consideration of the specific construction of these primitives. In contrast, here we show how to automatically generate two-party protocols for implementing specific digital signing or decryption algorithms, as well as other cryptographic algorithms of interest.

3. GOALS

As described in Section 1, the goal of our compiler is to automatically generate efficient, provably secure two-party protocols for an interesting class of cryptographic functions. In our current version of the compiler, the *input function* is restricted to computations expressed using selection of random elements from $\mathbb{Z}_q$ for q a public prime; multiplication, multiplicative inversion, addition, and additive inversion over $\mathbb{Z}_q$; and the generation and multiplication of elements of the multiplicative subgroup G_q of order q in $\mathbb{Z}_p^*$ with public generator g and public prime p, $q | p - 1$. In addition, the computation is permitted to (and will generally) input one or more "secret" parameters that are elements of $\mathbb{Z}_q$, and public auxiliary inputs (e.g., a message to be signed in a digital signature computation). These computations include, for example, many "discrete-log based" signature schemes and encryption schemes, i.e., schemes whose security is based on the difficulty of computing discrete logarithms to base g in G_q. The compiler assumes that all inputs to the computation (except for g, p and q) and all intermediate results are to be secret; only the values output from the computation (or that are derivable from that output) are permitted to be public.

We denote the two participating parties in our protocol by A and B. The output of the compiler is two algorithms (source

code), one for A and one for B. The input to each party consists of the public parameters of the system, its own public and secret parameters, public parameters of the other party and commitments of the secret parameters of the other party. More formally, the A algorithm expects the following inputs:

- common public inputs, including g, q, p, and the auxiliary inputs;
- private input $x_{\mathsf{A}} \in \mathbb{Z}_q$ per "secret" parameter x to the input function;
- private/public key pair $(sk_{\mathsf{A}}, pk_{\mathsf{A}})$ of a semantically secure encryption scheme $\mathcal{E}$ (with additional properties; see Section 4.2);
- B's public key pk_{B} of $\mathcal{E}$;
- and B's commitment $\bar{x}_{\mathsf{B}}$ for each secret parameter x to the input function, which in our case is an encryption of x_{B} under pk_{B}, i.e., $\bar{x}_{\mathsf{B}} \leftarrow E_{pk_{\mathsf{B}}}(x_{\mathsf{B}})$.

B expects an analogous set of inputs.

The execution of the two party protocol is started by A. This initiates an interactive protocol in which each of the parties sends messages that contain partial results of its computation to the other party. The protocol runs until either one of the parties aborts or A outputs the result of the input computation. In the case of two-party signature generation, the output value is the signature generated jointly by A and B. If A and B run to completion without aborting, the output produced by A is the same as the input function to the compiler would have produced, provided that for each secret input x, either $x_{\mathsf{A}} + x_{\mathsf{B}} \equiv_q x$ or $x_{\mathsf{A}} x_{\mathsf{B}} \equiv_q x$.

We emphasize that A and B require x_{A} and x_{B} as inputs, respectively, for each input parameter x marked as "secret" to the function input to the compiler. In addition, B is given pk_{A}, sk_{B}, and $\bar{x}_{\mathsf{A}}$, whereas A is given pk_{B}, sk_{A}, and $\bar{x}_{\mathsf{B}}$. The generation and distribution of these values is outside the scope of our compiler, and must be achieved via some other means, presumably by a trusted third party. Eliminating reliance on a third party for initialization is a planned area of future work.

4. COMPILER DESIGN

This section describes the design of our compiler and some implementation choices, using the DSA signature scheme as an example. Figure 1 presents a high-level overview of the compiler structure. The output of the compiler (the two algorithms for A and B) is generated from a collection of component protocols, called *building blocks*, and from an input file specified by the user. The component protocols correspond, intuitively, to arithmetic operations, and they are further decomposed into *primitive protocols* that are protocols with at most one interaction between the two parties. The input file contains the specification of a computation that will be transformed into a two-party protocol. Below, we take a bottom-up approach in detailing the compilation process.

4.1 Compiler Input

The input to our compiler is the specification of a computation given in a high level language that was created for this purpose. The language consists of keywords p, q, g, PARAMS, START, RETURN, SECRET, PUBLIC, RANDOM, MULT and ADD,

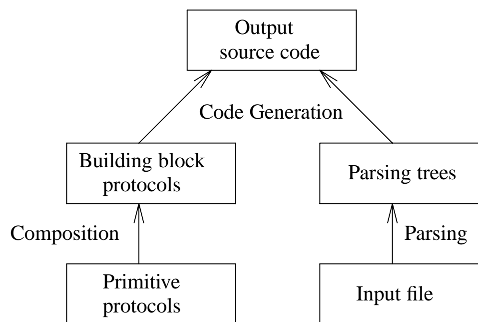

Figure 1: Compiler structure

and operators $+$ (addition in $\mathbb{Z}_q$), $*$, $\sim$ (multiplicative inversion in $\mathbb{Z}_q$), $\wedge$ (exponentiation in G_q with public base), $\%$ (reduction mod a public parameter) and $=$ (assignment). The binary operator $*$ is overloaded and can be used to multiply elements in $\mathbb{Z}_q$ or in G_q (it can be inferred from the context in which group the multiplication is done). Keywords MULT and ADD specify the type of sharing of a secret (multiplicatively or additively, respectively) and are optional. We require that keywords PARAMS, START and RETURN appear in the description file of each function in this exact order. For example, the input file for generating the two-party DSA signature scheme is given in Figure 2(a).

By convention, every variable is assumed to be secret, unless it is declared PUBLIC or returned in RETURN. In the example in Figure 2(a), k is secret, whereas r and s are public as being output of the computation. In addition, every input parameter is assumed to be an element of $\mathbb{Z}_q$, and RANDOM generates a random element of $\mathbb{Z}_q$. The computation itself begins after START and ends at RETURN. It can consist of arbitrarily many steps, with the restriction that each expression is generated by the context-free grammar from Figure 2 (b) (id here replaces any valid variable name using standard conventions).

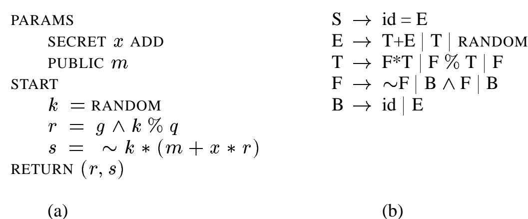

Figure 2: (a) Input file for DSA signature, (b) Grammar for generating expressions

4.2 Primitive protocols

The output of our compiler, i.e., the two-party protocol implemented by two algorithms for A and B, is constructed using a collection of primitive protocols. Intuitively, these primitive protocols define the "instructions" that are composed to generate the compiler's output. Each primitive protocol consists of at most one message, and has an *initiator*. The initiator's input to a primitive protocol includes one or more of its secret values, its commitments to those secret values, the other party's

commitments to its secret values, and possibly public values. The other party can employ corresponding values of its own, as well, though in most primitive protocols, this party works only with commitments from the initiator and public values; in all but two protocols, its own secrets are not employed.

A table specifying each of the primitive protocols is shown in Figure 3. The terminology we used for protocol names is as follows: Add, Mult, Inv, Exp, ModQ refer to the arithmetic operations performed; Add2Mult and Mult2Add refer to the conversion from additive to multiplicative sharing of a secret and the reverse; Rev means that the protocol has a public output; S/P stands for one of the input parameters being secret/public and we can have combinations of the form SS, SP if the protocol has two parameters; Dup means that the protocol has no functionality (its output is equal to its input).

This table specifies the primitive protocols in which A is the initiator. Those when B is the initiator are symmetric. For each protocol, the number of messages, either zero or one, is shown; when a message is sent it is always sent by the initiator. The next columns specify each party's inputs to the protocol, and the preconditions that it must know to be true in order to execute the protocol. Note that in some cases, the preconditions cannot be directly verified by the party who must be convinced of them. For example, B may be required to know that $D_{sk_A}(\bar{x}_A)$ (i.e., the decryption of $\bar{x}_A$ under private key sk_A) is less than some value, even though B does not know sk_A. This simply means that this fact will need to be *proven* to B (via a zero-knowledge proof) within another protocol (e.g., ModQ) if B cannot otherwise infer it, before B will execute the protocol for which this is a precondition. Each primitive protocol gives each party additional outputs, also shown, and enables that party to conclude the specified postconditions for those outputs. As a simple example, consider the second protocol listed in the table, called AddSS(x_A, y_A) (adds two secrets). In this protocol, A begins with values x_A and y_A with commitments (ciphertexts) $\bar{x}_A$ and $\bar{y}_A$; B also knows these commitments. Moreover, both A and B know that $D_{sk_A}(\bar{x}_A) + D_{sk_A}(\bar{y}_A) \leq R$ for some value R (more on this below). After this protocol (involving no messages), A will possess a new value $z_A = x_A + y_A$ and commitment $\bar{z}_A$ for it, and B will also possess $\bar{z}_A$. Moreover, each of them will know that $D_{sk_A}(\bar{z}_A) \leq R$.

The commitments of shares are generated using a semantically secure public key encryption scheme $\mathcal{E}$ with encryption algorithm E and decryption algorithm D that in addition have a homomorphic property. For a given public key pk, we denote by M_{pk} the plaintext space, which we assume is a range of integers $\{0, 1, \ldots, t\}$ with $t > q^l$. (Typically $l = 6$, e.g., for q of 160 bits and the public key of 1024 bits.) We use C_{pk} to denote the ciphertext space. We require that there exists an efficient additive operation $+_{pk} : C_{pk} \times C_{pk} \rightarrow C_{pk}$ and a multiplicative operation $\times_{pk} : C_{pk} \times M_{pk} \rightarrow C_{pk}$ such that:

- $\forall m_1, m_2 \in M_{pk}$: $m_1 + m_2 \in M_{pk} \Rightarrow$ $D_{sk}(E_{pk}(m_1) +_{pk} E_{pk}(m_2)) = m_1 + m_2$
- $\forall m_1, m_2 \in M_{pk}$: $m_1 m_2 \in M_{pk} \Rightarrow$ $D_{sk}(E_{pk}(m_1) \times_{pk} m_2) = m_1 m_2$

We observe that the existence of $+_{pk}$ implies the existence of $\times_{pk}$. Several examples of cryptosystems supporting these additional operations exist [5, 24, 28, 29]. In our implementation, we use the Paillier cryptosystem [29] whose security is based on the composite residuosity problem, but any of the above-mentioned cryptosystems could be used instead.

In Figure 3, R denotes the largest number in $M_{pk_A} \cap M_{pk_B}$ and is used to ensure that some values are in the plaintext range of both the encryption schemes used.

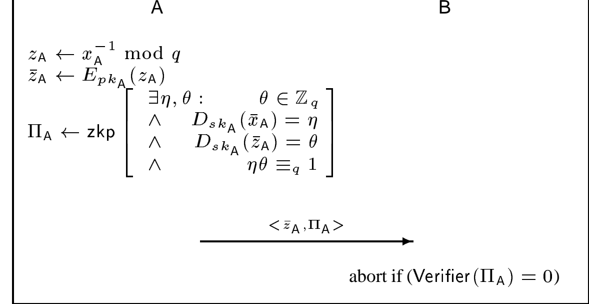

Figure 4: InvS protocol

For illustration, we present an example of a primitive protocol: InvS(x_A) (inverts a secret) in Figure 4. In the protocol, A's input is a value x_A and commitment $\bar{x}_A = E_{pk_A}(x_A)$, whereas B's input is only $\bar{x}_A$. After executing the protocol, A will have a share $z_A = x_A^{-1} \in \mathbb{Z}_q$ and its commitment $\bar{z}_A = E_{pk_A}(z_A)$. B's output will consist only of the commitment $\bar{z}_A$. A generates a zero knowledge proof Π_A, in which it proves to B that $D_{sk_A}(\bar{z}_A) = (D_{sk_A}(\bar{x}_A))^{-1} \bmod q$. B verifies the proof using the polynomial time algorithm Verifier. The proof Π_A is omitted due to space limitations, but will be included in the full version of this paper.

4.3 Building Block Protocols

The fine granularity of the primitive protocols described in Section 4.2 offers many opportunities for experimenting with how to construct the most efficient two-party protocols for a given input computation. As an initial exploration into this space, however, our work so far has focused on only one simple way of combining them to reach the given input computation. The technique we have explored thus far is to compose primitive protocols into larger two-party *building block* protocols that implement certain operations on shared secrets. Then, our compiler emits its output using building blocks, rather than emitting instances of primitive protocols directly.

Working with building blocks is more intuitive—and easier to prove things about—than working with primitive protocols directly, since each building block corresponds to a basic operation on shared secrets such as addition, multiplication, inversion, modular reduction, exponentiation or generation of random secrets in the input computation. Two additional building blocks perform conversion from additive shares to multiplicative shares of a secret and the other way around. These are necessary as each building block requires the input secret(s) to be shared in the proper way (either additively or multiplicatively). Building blocks are constructed via composition of primitive protocols:

DEFINITION 4.1. *Let P_1 and P_2 be two-party protocols. Then $P = P_1 \parallel P_2$ is a two-party protocol in which protocols P_1 and P_2 are executed sequentially, starting with P_1. We call P the sequential composition of P_1 and P_2.*

Protocol	Msgs	Party	Input	Preconditions	Output	Postconditions
Generate()	1	A	-		$z_A, \bar{z}_A$	$D_{sk_A}(\bar{z}_A) = z_A$ $z_A \in \mathbb{Z}_q$
		B	-		$\bar{z}_A$	
AddSS(x_A, y_A)	0	A	$x_A, \bar{x}_A$ $y_A, \bar{y}_A$	$D_{sk_A}(\bar{x}_A) = x_A$ $D_{sk_A}(\bar{y}_A) = y_A$ $x_A + y_A \leq R$	$z_A, \bar{z}_A$	$z_A = x_A + y_A$ $D_{sk_A}(\bar{z}_A) = z_A$
		B	$\bar{x}_A$ $\bar{y}_A$	$D_{sk_A}(\bar{x}_A) + D_{sk_A}(\bar{y}_A) \leq R$	$\bar{z}_A$	$D_{sk_A}(\bar{z}_A) = D_{sk_A}(\bar{x}_A) + D_{sk_A}(\bar{y}_A)$
AddSP(x_A, y)	0	A	$x_A, \bar{x}_A$ y	$D_{sk_A}(\bar{x}_A) = x_A$ $x_A + y \leq R$	$z_A, \bar{z}_A$	$z_A = x_A + y$ $D_{sk_A}(\bar{z}_A) = z_A$
		B	$\bar{x}_A$ y	$D_{sk_A}(\bar{x}_A) + y \leq R$	$\bar{z}_A$	$D_{sk_A}(\bar{z}_A) = D_{sk_A}(\bar{x}_A) + y$
MultSS(x_A, y_A)	1	A	$x_A, \bar{x}_A$ $y_A, \bar{y}_A$	$D_{sk_A}(\bar{x}_A) = x_A$ $D_{sk_A}(\bar{y}_A) = y_A$ $x_A y_A \leq R$	$z_A, \bar{z}_A$	$z_A = x_A y_A \bmod q$ $D_{sk_A}(\bar{z}_A) = z_A$
		B	$\bar{x}_A$ $\bar{y}_A$	$D_{sk_A}(\bar{x}_A) D_{sk_A}(\bar{y}_A) \leq R$	$\bar{z}_A$	$D_{sk_A}(\bar{z}_A) \equiv_q D_{sk_A}(\bar{x}_A) D_{sk_A}(\bar{y}_A)$
MultSP(x_A, y)	0	A	$x_A, \bar{x}_A$ y	$D_{sk_A}(\bar{x}_A) = x_A$ $x_A y \leq R$	$z_A, \bar{z}_A$	$z_A = x_A y$ $D_{sk_A}(\bar{z}_A) = z_A$
		B	$\bar{x}_A$ y	$D_{sk_A}(\bar{x}_A) y \leq R$	$\bar{z}_A$	$D_{sk_A}(\bar{z}_A) = D_{sk_A}(\bar{x}_A) y$
InvS(x_A)	1	A	$x_A, \bar{x}_A$	$D_{sk_A}(\bar{x}_A) = x_A$	$z_A, \bar{z}_A$	$z_A = (x_A)^{-1} \bmod q$ $D_{sk_A}(\bar{z}_A) = z_A$
		B	$\bar{x}_A$	$D_{sk_A}(\bar{x}_A) < R/2q^3$	$\bar{z}_A$	$D_{sk_A}(\bar{z}_A) = (D_{sk_A}(\bar{x}_A))^{-1} \bmod q$
RevExp(x_A, y)	1	A	$x_A, \bar{x}_A$ y	$D_{sk_A}(\bar{x}_A) = x_A$	z_A	$z_A = y^{x_A} \bmod p$
		B	$\bar{x}_A$ y		z_A	$z_A = y^{D_{sk_A}(\bar{x}_A)} \bmod p$
Add2Mult(x_A, x_B)	1	A	$x_A, \bar{x}_A$ $\bar{x}_B$	$D_{sk_A}(\bar{x}_A) = x_A$ $D_{sk_B}(\bar{x}_B) + x_A < R/q^3$	$z_A, \bar{z}_A$ $\bar{z}_B$	$z_A \in \mathbb{Z}_q$ $D_{sk_A}(\bar{z}_A) = z_A$ $z_A D_{sk_B}(\bar{z}_B) \equiv_q x_A + D_{sk_B}(\bar{x}_B)$
		B	$x_B, \bar{x}_B$ $\bar{x}_A$	$D_{sk_B}(\bar{x}_B) = x_B$	$z_B, \bar{z}_B$ $\bar{z}_A$	$D_{sk_B}(\bar{z}_B) = z_B$ $D_{sk_A}(\bar{z}_A) z_B \equiv_q D_{sk_A}(\bar{x}_A) + x_B$
Mult2Add(x_A, x_B)	1	A	$x_A, \bar{x}_A$ $\bar{x}_B$	$D_{sk_A}(\bar{x}_A) = x_A$ $D_{sk_B}(\bar{x}_B) x_A < R/q^2$	$z_A, \bar{z}_A$ $\bar{z}_B$	$z_A \in \mathbb{Z}_q$ $D_{sk_A}(\bar{z}_A) = z_A$ $z_A + D_{sk_B}(\bar{z}_B) \equiv_q x_A D_{sk_B}(\bar{x}_B)$
		B	$x_B, \bar{x}_B$ $\bar{x}_A$	$D_{sk_B}(\bar{x}_B) = x_B$	$z_B, \bar{z}_B$ $\bar{z}_A$	$D_{sk_B}(\bar{z}_B) = z_B$ $D_{sk_A}(\bar{z}_A) + z_B \equiv_q D_{sk_A}(\bar{x}_A) x_B$
ModQ(x_A)	1	A	$x_A, \bar{x}_A$	$D_{sk_A}(\bar{x}_A) = x_A$	$z_A, \bar{z}_A$	$z_A = x_A \bmod q$ $D_{sk_A}(\bar{z}_A) = z_A$
		B	$\bar{x}_A$		$\bar{z}_A$	$D_{sk_A}(\bar{z}_A) = D_{sk_A}(\bar{x}_A) \bmod q$
Rev(x_A)	1	A	$x_A, \bar{x}_A$	$D_{sk_A}(\bar{x}_A) = x_A$		
		B	$\bar{x}_A$		x_A	$D_{sk_A}(\bar{x}_A) = x_A$
Dup(x_A)	0	A	$x_A, \bar{x}_A$		$z_A, \bar{z}_A$	$z_A = x_A$ $\bar{z}_A = \bar{x}_A$
		B	$\bar{x}_A$		$\bar{z}_A$	$\bar{z}_A = \bar{x}_A$

Figure 3: Primitive protocol specifications

In addition to the primitive protocols discussed in Section 4.2, we need some local computation protocols that perform operations only on public values, so we employ one such protocol for each arithmetic operation: Add2Pub (adds two public values), Mult2Pub (multiplies two public values), InvPub (computes the multiplicative inverse of a public value in $\mathbb{Z}_q$), ModQPub (computes the reduction mod q of a public value) and so forth. Our building blocks, enumerated in Figure 5, are compositions of primitive protocols and local computation protocols. The preconditions and postconditions of each building block protocol are accumulated from the preconditions and postconditions of the primitive protocols that comprise it.

4.4 Emitting the Two-Party Protocol

The emitted two-party protocol is a sequential composition of building blocks described in Section 4.3. In this section, we describe how the sequence of building blocks is determined by the compiler, given the specification of the input function.

The compilation process has three important phases: parsing, construction of the sequence of interactive building blocks and generation of the output source code for each party. Below we will detail each of the three steps.

4.4.1 Parsing

During the parsing process, the compiler checks that the input file has the required format, signaling for existing errors. In addition, for each step in the computation (these are given by expressions between keywords START and RETURN), it will generate an associated parsing tree as in Figure 6. The nodes in the parsing trees represent operations: GEN$_q$ (generation of random secrets in $\mathbb{Z}_q$), $+_q$ (addition of elements in $\mathbb{Z}_q$), $\times_q$ (multiplication of elements in $\mathbb{Z}_q$), -1_q (inversion of a $\mathbb{Z}_q$ element), $\wedge_p$ (exponentiation of an element in G_q to an exponent in $\mathbb{Z}_q$), $\%q$ (reduction $\bmod\ q$), Conv (conversion from multiplicative shares of a secret to additive shares of the same secret or the reverse), Rev (reveal a secret). Terminal nodes

Building block	Description	Composition
`Generate`	Generates a random shared secret	$z_A \leftarrow$ Generate() $z_B \leftarrow$ Generate()
`Add2Secrets`	Adds two secrets (x, y), additively shared	$z_A \leftarrow$ AddSS(x_A,y_A) $z_B \leftarrow$ AddSS(x_B,y_B)
`AddSecretPub`	Adds a public value (y) and a secret additively shared (x)	$z_A \leftarrow$ AddSP(x_A, y) $z_B \leftarrow$ Dup(x_B)
`Mult2Secrets`	Multiplies two secrets (x, y) multiplicatively shared	$z_A \leftarrow$ MultSS(x_A,y_A) $z_B \leftarrow$ MultSS(x_B,y_B)
`MultSecretPub`	Multiplies a public value (y) and a secret (x) multiplicatively shared	$z_A \leftarrow$ MultSP(x_A,y) $z_B \leftarrow$ Dup(x_B)
`InvSecret`	Inverts a secret (x), multiplicatively shared	$z_A \leftarrow$ InvS(x_A) $z_B \leftarrow$ InvS(x_B)
`RevealExp`	Generates $y^x \bmod p$ where y is public and x is an additively shared secret	$z_A \leftarrow$ RevExp(x_A, y) $z_B \leftarrow$ RevExp(x_B, y) $z \leftarrow$ Mult2Pub(z_A, z_B)
`Add2Mult`	Generates a multiplicative sharing of an additively shared secret (x)	$\langle z_A, z'_B \rangle \leftarrow$ Add2Mult(x_A, x_B) $z_B \leftarrow$ ModQ(z'_B)
`Mult2Add`	Generates an additive sharing of a multiplicatively shared secret (x)	$\langle z_A, z'_B \rangle \leftarrow$ Mult2Add(x_A, x_B) $z_B \leftarrow$ ModQ(z'_B)
`ModQ`	Reduces a secret mod q	$z_A \leftarrow$ ModQ(x_A) $z_B \leftarrow$ ModQ(x_B)
`RevealAdd`	Reveals a secret (x) additively shared	$x_A \leftarrow$ Rev(x_A) $x_B \leftarrow$ Rev(x_B) $x \leftarrow$ Add2Pub(x_A, x_B)
`RevealMult`	Reveals a secret (x) multiplicatively shared	$x_A \leftarrow$ Rev(x_A) $x_B \leftarrow$ Rev(x_B) $x \leftarrow$ Mult2Pub(x_A, x_B)

Figure 5: Building block protocols

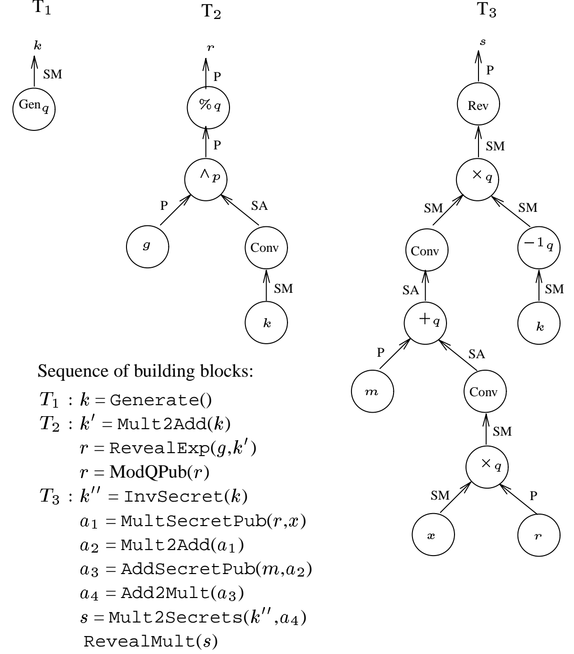

Figure 6: Parsing trees for generating a DSA signature

in the tree (leaves) are either input variables to the compiler or output variables of previous trees. Each node has one or two entering edges corresponding to the input of the building block the node represents and one leaving edge, corresponding to the output of the building block. The edges are labelled with the type of the corresponding variable: P (public), SM (secret shared multiplicatively) or SA (secret shared additively).

During the parsing phase, the compiler constructs a list of all secrets whose sharing type is unknown (it is not explicitly specified in the input file and it can not be determined from the execution of the protocol) and then for each possible assignment of sharing types to secrets, compute the number of convert protocols required in the protocol. Our compiler then chooses the assignment that minimizes the number of convert protocols, as the convert protocols are the most computationally expensive building blocks. This algorithm is exponential in the number of secrets considered, but this number is typically fairly small.

4.4.2 Construction of building block sequence

In the second phase, each arithmetic operation is replaced with the corresponding building block, e.g., addition with `Add2Secrets`, `AddSecretPub` or Add2Pub, depending on the type of operands. Convert protocols are inserted whenever secrets are not shared properly for subsequent building blocks. If the left hand side of an assignment is secret, but at the same time is part of the computation output, then a `Reveal` protocol must be executed. In addition, a `ModQ` protocol is executed whenever a share might exceed the range specified by a precondition of a subsequent building block. An example for constructing the sequence of building blocks for the DSA signature scheme is given in Figure 6.

4.4.3 Generation of Java code

In this phase, Java source code for A and B is automatically generated by our compiler, using the sequence of building block protocols determined previously. During the interactive execution of the protocol, the two parties can exchange messages of four types: PARAMS (one party sends the output of its computations to the other party), REQ-PROOF (a party requests a zero knowledge proof of correctness of a computation from the other party), PROOF (one party sends a zero-knowledge proof to the other party as a response to the REQ-PROOF message) and DONE (one party informs the other that it has finished the protocol). The model we adopt in the implementation is that of "pulling the proofs" (generating zero knowledge proofs on request) as opposed to the theoretical building blocks where proofs are generated immediately after each computation. We believe that this model can lead to optimizations in our protocol, by allowing aggregation of proofs and minimization of the number of interactions. We intend to experiment with this in future work.

5. SECURITY FOR TWO-PARTY SIGNATURE SCHEMES

The two-party protocols generated by our compiler preserve the security of the original input function to the compiler. The proofs of security are built on the simulatability property of the underlying building blocks (Theorem A.2 from Appendix A). The security of the two-party protocols additionally relies on the security of the encryption scheme used for the commitments of secret shares and on the security of the zero knowl-

edge proofs involved. In this section, we give an example of such a proof when the input function to the compiler is a signature scheme.

Let SIG be a generic signature scheme and S-SIG = $P_1 \parallel \cdots \parallel P_n$ be our two-party protocol S-SIG for generating a signature (a sequential composition of n building blocks). In the security analysis of S-SIG, we require that SIG is secure against chosen message attack [19] and the encryption scheme $\mathcal{E}$ is semantically secure [18]. We also require non-interactive zero-knowledge proofs, for which formal definitions can be found in [6, 12]. Informally, we remind the reader that a non-interactive zero-knowledge proof system for a language L is measured by its *soundness error* (the probability that a malicious prover convinces a verifier to accept a $w \notin L$) and its *simulation error* (for any simulator holding $w \in L$ and answering random oracle queries, the probability with which a verifier can distinguish the simulator from the real prover).

We consider two types of adversaries for S-SIG. An A-compromising adversary is a probabilistic polynomial-time adversary that is given the public key of SIG and access to the B oracle. The B oracle can be queried by invoking n oracles: $\mathsf{BQ}^1, \ldots, \mathsf{BQ}^n$, where a query to the ith oracle, BQ^i, corresponds to the execution of the P_i building block by B. The query to the BQ^1 oracle must have as input the message m to be signed. An A-compromising adversary has access in addition to all the secret computation of A. A B-compromising adversary is defined analogously.

DEFINITION 5.1. *A two-party signature scheme is secure against an* A*-compromising adversary $\mathcal{A}$ if $\mathcal{A}$ has a negligible probability in outputting a pair (m, σ) such that σ is a valid signature for m and the message m was not sent to the* B *oracle in a* BQ^1 *query.*

Below we state the security theorem of the S-SIG protocol against an A-compromising adversary. A similar definition of security of a two-party signature scheme against a B-compromising adversary and a proof of security of S-SIG against this type of adversary could be given.

THEOREM 5.2. *If an* A*-compromising adversary forges a signature in the* S-SIG *scheme with non-negligible probability, then:*

1. *There exists an attacker that forges a signature in* SIG *with non-negligible probability; or*
2. *There exists an attacker that breaks the semantic security of $\mathcal{E}$ with non-negligible probability; or*
3. *One of the zero knowledge proofs used in the* S-SIG *protocol has a non-negligible simulation error; or*
4. *One of the zero knowledge proofs used in the* S-SIG *protocol has a non-negligible soundness error.*

The proof of the theorem is given in Appendix A.

6. PERFORMANCE RESULTS

We implemented our compiler in Java. To optimize some of the expensive operations performed in our implementation (such as exponentiations or generation of random numbers) we used the gmp C library. In our system, A is a client of server B. A initiates the communication and B responds to A's messages until a signature is generated. The client and server communicate through sockets.

Scheme	Signature	A (sec)	B (sec)	sharing of x
EG I.1	$s \equiv_q k^{-1}(m + xr)$	1.019	1.151	additive
EG I.2	$s \equiv_q x^{-1}(m + kr)$	0.828	0.882	multiplicative
EG I.3	$s \equiv_q xr + km$	0.129	0.141	additive
EG I.4	$s \equiv_q xm + kr$	0.130	0.143	additive
EG I.5	$s \equiv_q x^{-1}(r + xm)$	0.837	0.896	multiplicative
EG I.6	$s \equiv_q k^{-1}(r + xm)$	1.022	1.144	additive

Figure 7: Performance results

We ran our implementation on a two-processor machine, each a 2.4 GHz Intel Xeon, and tested it on the various signature schemes proposed in [20]. Figure 7 shows the unoptimized computation time for both A and B for generating a two-party signature. The signature schemes used are the six basic types of signatures from [20]. All these schemes have the same key generation algorithm, they differ only in the signature generation and verification. Signature generation is of the form: $k \leftarrow_R \mathbb{Z}_q; r = g^k \bmod p; s = ...;$ output (r, s) with different relations for the computation of s. The table shows the specific equation for s for each of the schemes and the verification relation. Scheme EG I.1 is a variant of DSA in which r is not reduced mod q.

As already mentioned in [20], types 3 and 4 are the most efficient schemes, as they do not require the computation of an inverse mod q. Our experiments confirm that this is also true for the two party case, types 3 and 4 being nearly an order of magnitude more efficient than the other four types.

For each of the types, we have generated two party protocols utilizing both additive and multiplicative sharing of the secret key x. An interesting observation is that the most efficient protocol for DSA is one in which x is shared additively, but the hand-tuned two-party DSA [22] employs a multiplicative sharing of x.

We are planning to explore future optimizations in our compiler, such as parallelizing computation or using pre-computed tables for exponentiations with the same base [23]. While we expect that the protocols generated by our compiler will not be as efficient as hand-tuned approaches, the performance results are already promising.

7. APPLICATIONS

The initial focus of our compiler has been for transforming discrete-log based signature schemes and encryption schemes into secure two-party implementations. However, our compiler can be applied to a much broader range of applications that are also of interest. Here we list two such applications.

7.1 Password hardening

In [15], a *password hardening* protocol is described to enable a user to obtain a strong secret from a password π by interacting with a server (which does not learn the secret obtained). The protocol uses public primes p, q such that $p = 2q + 1$ and a public function f that maps passwords to elements in $\mathbb{Z}_q$. The server initially selects a secret value d for the user, and the user interacts with the server using her password π to retrieve the stronger secret $f(\pi)^d \bmod p$ as follows: (i) The

user picks $k \leftarrow_R \mathbb{Z}_q$, computes $r \leftarrow f(\pi)^k \bmod p$ and sends it to the server; (ii) the server computes $s \leftarrow r^d \bmod p$ and sends it to the user; and (iii) the user computes $s^{k^{-1}} \bmod p$, which is the strong secret.

The full paradigm proposed by [15] involves a client performing this password hardening protocol with multiple servers independently. An alternative that enables the service to be distributed transparently to the client (i.e., the client still communicates with only one server) would be to automatically implement a distributed server using our compiler, consisting of two servers that jointly compute s using shares of d. Client transparency is beneficial if the server must be distributed without changing legacy clients, for example.

7.2 Oblivious transfer

Some oblivious transfer (OT) protocols (e.g., [2, 27]) can be transformed into distributed oblivious transfer protocols [26], using our compiler. The goal of a 1-out-of-N OT protocol is to allow a receiver R to retrieve one of N secrets from a sender S so that S does not learn which of the N secrets R retrieved and so that R receives only one of the N secrets (and learns no information about the others). In a distributed OT protocol, the sender is distributed into multiple senders and each of them holds shares of the secrets. Our compiler enables the distribution of S into two senders automatically for the protocols mentioned above.

8. CONCLUSIONS

We have described the construction of a compiler that automatically generates two-party computation protocols starting from the specification of a computation involving one or more secret values. The two-party protocol is generated by composing building blocks for the fundamental arithmetic operations: addition, multiplication, exponentiation, inversion, modular reduction. Each building block protocol consists of two or more primitive protocols, that intuitively are "one-way" protocols (are initiated by one of the parties and the initiator can transmit at most one message to the receiver). As future work, we would like to explore further methods of combining directly the primitive protocols, parallelizing the computation.

The two-party protocols generated by our compiler are simulatable. This property allows us to prove that the security of the input function to the compiler is preserved by our two-party protocol against arbitrary misbehavior of either party. The security of the two-party protocol additionally relies on the security of the encryption scheme used for the commitments of secret shares and on the security of the zero knowledge proofs involved. For illustration, in Appendix A we show a proof of security in the case when the input function is a signature scheme. There are also other applications of our compiler in the context of protocols that require a trusted third party or trusted server (e.g. key distribution, fair exchange, key retrieval). The trusted party could be replaced by two parties, eliminating the single point of failure.

9. ACKNOWLEDGEMENTS

We would like to thank Ke Yang for his helpful comments and discussions.

10. REFERENCES

[1] B. Barak, A. Herzberg, D. Naor, and E. Shai. The proactive security toolkit and applications. In *Proc. 6th ACM Conf. Computer and Communications Security*, pp. 18–27, Nov. 1999.

[2] M. Bellare, S. Micali. Non-interactive oblivious transfer and applications. In *Proc. CRYPTO '89*, 1989.

[3] M. Bellare, R. Sandhu. The security of practical two-party RSA signature schemes. 2001.

[4] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc.* 1st *ACM Conf. Computer and Communications Security*, pp. 62–73, Nov. 1993.

[5] J. Benaloh. Dense probabilistic encryption. In *Workshop on Selected Areas of Cryptography*, pp. 120–128, 1994.

[6] M. Blum, A. DeSantis, S. Micali, and G. Persiano. Noninteractive zero-knowledge. *SIAM Journal of Computing* 20(6):1084–1118, 1991.

[7] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption systems. In *Proc. CRYPTO '98* (LNCS 1462), 1998.

[8] D. Boneh, X. Ding, G. Tsudik, and M. Wong. A method for fast revocation of public key certificates and security capabilities. In *Proc. 10th USENIX Security Symposium*, Aug. 2001.

[9] C. Boyd. Digital multisignatures. In H. J. Beker and F. C. Piper, editors, *Cryptography and Coding*, pp. 241–246. Clarendon Press, 1986.

[10] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO '98*, 1998.

[11] R. Cramer and V. Shoup. Signature schemes based on the strong RSA assumption. *ACM Transactions on Information and System Security* 3(3):161–185, 2000.

[12] A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano, and A. Sahai. Robust non-interactive zero knowledge. In *Proc. CRYPTO 2001* (LNCS 2139), pp. 566–598, 2001.

[13] Y. Desmedt. Society and group oriented cryptography: a new concept. In *Proc. CRYPTO '87* (LNCS 293), pp. 120–127, 1987.

[14] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Proc. CRYPTO '89* (LNCS 435), pp. 307–315, 1989.

[15] W. Ford, B. Kaliski. Server-assisted generation of a strong secret from a password. In *Proc. 5th International Workshop on Enterprise Security*, 2000.

[16] R. Ganesan. Yaksha: Augmenting Kerberos with public key cryptography. In *Proc. 1995 ISOC Network and Distributed System Security Symposium*, Feb. 1995.

[17] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In *Proc. EUROCRYPT '96* (LNCS 1070), pp. 354–371, 1996.

[18] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences* 28:270–299, 1984.

[19] S. Goldwasser, S. Micali and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*

17(2):281-308, Apr. 1988.

[20] P. Horster, H. Petersen, and M. Michels. Meta-ElGamal signature schemes. In *Proc. 2nd ACM Conf. Computer and Communications Security*, pp. 96–107, Nov. 1994.

[21] P. MacKenzie and M. K. Reiter. Networked cryptographic devices resilient to capture. DIMACS Technical Report 2001-19, May 2001. Extended abstract in *2001 IEEE Symposium on Security and Privacy*, May 2001.

[22] P. MacKenzie and M. K. Reiter. Two-party generation of DSA Signatures. In *Proc. CRYPTO 2001* (LNCS 2139), Aug. 2001.

[23] A. Menezes, P. van Oorschot, S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[24] D. Naccache and J. Stern. A new public-key cryptosystem. In *Proc. EUROCRYPT '97* (LNCS 1233), pp. 27–36, 1997.

[25] A. Nicolosi, M. Krohn, Y. Dodis, D. Mazieres. Proactive two-party signatures for user authentication. In *Proc. 10th ISOC Network and Distributed System Security Symposium*, Feb. 2003.

[26] M. Naor, B. Pinkas. Distributed oblivious transfer. In *Proc. ASIACRYPT 2000*, Dec. 2000.

[27] M. Naor, B. Pinkas. Efficient oblivious transfer protocols. In *Proc. SODA 2001*, Jan. 2001.

[28] T. Okamoto and S. Uchiyama. A new public-key cryptosystem, as secure as factoring. In *Proc. EUROCRYPT '98* (LNCS 1403), pp. 308–318, 1998.

[29] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT '99* (LNCS 1592), pp. 223–238, 1999.

[30] M. Reiter, M. Franklin, J. Lacy, R. Wright. The Ω key management service. *Journal of Computer Security* 4(4):267–297, 1996.

[31] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21(2):120–126, Feb. 1978.

[32] A. Shamir. How to share a secret. *Communications of the ACM* 22, 1979.

[33] D. Song, A. Perrig and D. Phan. AGVI — Automatic generation, verification, and implementation of security protocols. In *Proc. 13th Conference on Computer Aided Verification*, July 2001.

[34] T. Wu, M. Malkin, D. Boneh. Building intrusion tolerant applications. In *Proc. 8th USENIX Security Symposium*, pp. 79-91, Aug. 1999.

[35] A. Yao. Protocols for secure computation. In *Proc. 23 IEEE Symposium on Foundations of Computer Science*, pp. 160–164, 1982.

[36] L. Zhou, F. Schneider, R. van Renesse. COCA: A secure distributed on-line certification authority. In *ACM Transactions on Computer Systems* 20(4), Nov. 2002.

APPENDIX

A. SIMULATABILITY OF THE BUILDING BLOCK PROTOCOLS

In defining simulatability of a two-party building block protocol, we consider two cases, depending on the output parameter of the protocol being secret or public.

DEFINITION A.1. *(Simulatability for Building Block Protocols with Secret Output) Let* $P = P_1 \parallel P_2 \parallel \ldots P_i$ *be a composition of* i *building block protocols. The last building block protocol* P_i *with secret output is simulatable if given an adversary* $\mathcal{A}=(A_1, A_2)$ *that compromises one of the parties* C*, there exists a simulator* SIM_U *for the uncompromised party* U *such that:* $\text{Dist}_i^{\mathcal{A}}(\text{U}, \text{SIM}_\text{U}) = \Pr(\text{EXP}_{\mathcal{A},1}^{\text{ind-sr}} = 1) - \Pr(\text{EXP}_{\mathcal{A},0}^{\text{ind-sr}} = 1)$ *is negligible, where:*

$$\begin{aligned}&\text{EXP}_{\mathcal{A},b}^{\text{ind-sr}}:\\&\quad(\text{S}_u, \text{S}_c, \text{P}_s) \leftarrow I(1^\kappa)\\&\quad z \leftarrow A_1^{\text{U}(\text{S}_u,\text{S}_c,\text{P}_s)}(1^\kappa, \text{S}_c, \text{P}_s)\\&\quad b' \leftarrow A_2^{LR(\text{S}_u,\text{S}_c,\text{P}_s,L,b)}(z)\\&\quad \textit{output } b'\end{aligned}$$

Here S_u *denotes the secret parameters of the uncompromised party* U*,* S_c *the secrets of the corrupted party* C*,* P_s *the public parameters of the system,* κ *the security parameter,* I *the initialization algorithm;* $\text{U}(\text{S}_u, \text{S}_c, \text{P}_s)$ *is an oracle that given* $\text{S}_u, \text{S}_c, \text{P}_s$ *executes the uncompromised party in the protocol* P *up to building block* P_{i-1}*,* z *is the state of the adversary and* L *is a list of commitments produced by calls to the* U *oracle; the oracle* LR *is defined as* $LR(\text{S}_u, \text{S}_c, \text{P}_s, L, b) = \text{U}_{P_i}(\text{S}_u, \text{S}_c, \text{P}_s, L)$ *if* $b = 0$ *(*U_{P_i} *is the execution of the uncompromised party in building block* P_i*), and* $LR(\text{S}_u, \text{S}_c, \text{P}_s, L, b) = \text{SIM}_{\text{U},P_i}(\text{S}_c, \text{P}_s, L)$ *if* $b = 1$ *(*$\text{SIM}_{\text{U},P_i}$ *is the simulation of the uncompromised party in building block* P_i*).*

The definition of simulatability for building block protocols with public output is similar to this definition, with the only difference that the simulator SIM_U is also given as input the public output of the protocol.

Notation. Consider a protocol $P = P_1 \parallel P_2 \parallel \ldots P_i$. Throughout the rest of the paper we use the following notation: $\text{SIMERR}_{\Pi_{i\mathsf{B}}}$ is the simulation error of proof $\Pi_{i\mathsf{B}}$ used in the B protocol from P_i; $w_{i\mathsf{A}}$ denotes the set of A's commitments of its secret shares from protocol P_i; $\text{SERR}_{\Pi_{i\mathsf{A}}}(w_{i\mathsf{A}})$ is the soundness error of the proof $\Pi_{i\mathsf{A}}$ used in the A protocol from P_i; $\text{SIM}_{\Pi_{i\mathsf{B}}}$ is the simulator for the proof $\Pi_{i\mathsf{B}}$; $\mathsf{Adv}_{\mathcal{E}}(\mathcal{A})$ is the advantage of $\mathcal{A}$ for the encryption scheme $\mathcal{E}$, as defined in [7, Property IND-CPA]. B′ denotes the protocol B, except that in addition to checking all the zero knowledge proofs generated by A, it decrypts A's commitments (having access to A's secret key) and checks explicitly the predicates of the zero knowledge proofs.

The general simulatability theorem for an A-compromising forger is given below (the theorem for a B-compromising forger is similar to this).

THEOREM A.2. *Let* $\mathcal{A}$ *be an* A*-compromising attacker for* $P = P_1 \parallel \cdots \parallel P_i$ *. Then there is a simulation* SIM-B *of the* B *protocol in* P *and a distinguisher algorithm* D *such that:*

1. *If P_i is one of* `Generate`, `Mult2Secrets`, `InvSecret`, `ModQ`, `Add2Mult` *or* `Mult2Add` *then:*

   $\mathrm{Dist}_i^D(\mathsf{B},\text{SIM-B}) \leq \mathsf{Adv}_{\mathcal{E}}(\mathcal{A}) + \text{SIMERR}_{\Pi_{i\mathsf{B}}} + \sum_{j=1}^{i} \text{SERR}_{\Pi_{j\mathsf{A}}}(w_{j\mathsf{A}})$

   $\mathrm{Dist}_i^D(\mathsf{B}',\text{SIM-B}) \leq \mathsf{Adv}_{\mathcal{E}}(\mathcal{A}) + \text{SIMERR}_{\Pi_{i\mathsf{B}}}$

2. *If P_i is one of* `Add2Secrets`, `AddSecretPub` *or* `MultSecretPub` *then:*

   $\mathrm{Dist}_i^D(\mathsf{B}, \text{SIM-B}) \leq \mathsf{Adv}_{\mathcal{E}}(\mathcal{A}) + \sum_{j=1}^{i} \text{SERR}_{\Pi_{j\mathsf{A}}}(w_{j\mathsf{A}})$

   $\mathrm{Dist}_i^D(\mathsf{B}',\text{SIM-B}) \leq \mathsf{Adv}_{\mathcal{E}}(\mathcal{A})$

3. *If P_i is one of* `RevealMult`, `RevealAdd` *or* `RevealExp` *then:*

   $\mathrm{Dist}_i^D(\mathsf{B}, \text{SIM-B}) \leq \text{SIMERR}_{\Pi_{i\mathsf{B}}} + \sum_{j=1}^{i} \text{SERR}_{\Pi_{j\mathsf{A}}}(w_{j\mathsf{A}})$

   $\mathrm{Dist}_i^D(\mathsf{B}', \text{SIM-B}) \leq \text{SIMERR}_{\Pi_{i\mathsf{B}}}$

The proof of this theorem is omitted due to space limitations, but will appear in the full paper. We have now all the tools to complete the proof of Theorem 5.2:

Proof of Theorem 5.2. Assume an A-compromising forger F forges a signature in the S-SIG scheme with probability ϵ. Let $(\mathsf{BQ}_1^1, \ldots, \mathsf{BQ}_{q_{1\mathsf{B}}}^1), \ldots, (\mathsf{BQ}_1^n, \ldots, \mathsf{BQ}_{q_{n\mathsf{B}}}^n)$ denote all the queries made to the B oracle, and let q_B denote the total number of queries to the B oracle. Hence q_B must be polynomial in the security parameter and $\sum_{i=1}^n q_{i\mathsf{B}} = q_\mathsf{B}$.

We construct a simulation SIM of S-SIG that takes as input a SIG public key, a corresponding signature oracle, a public key/secret key pair $(pk_\mathsf{A}, sk_\mathsf{A})$ of the encryption scheme $\mathcal{E}$ and a public key pk_B for $\mathcal{E}$. SIM responds to query BQ^i to the B oracle as follows:

1. If $P_i = P_1$, then the signature oracle is queried on message m and the output signature $\sigma = (\sigma_1, \ldots, \sigma_t)$ is saved. Each σ_j corresponds to the output of one public output building block P_{i_j}, $j = 1, \ldots, t$. (We made the convention that only values that are part of the computation output are public.)
2. If P_i is a secret output building block, use the simulator SIM-B for P_i with input all the input parameters for SIM and the public output and commitments of all $P_{i'}$, $i' < i$, and output whatever the simulator outputs.
3. If $P_i = P_{i_j}$ is a public output building block, then use the simulator SIM-B for P_{i_j} with input all the input parameters for SIM, the public output and commitments of all $P_{i'}$, $i' < i$, and σ_j and output whatever the simulator outputs.

Now consider a forger F^* that takes as input a SIG public key and corresponding signature oracle, generates a public key/secret key pair $(pk_\mathsf{A}, sk_\mathsf{A})$ and a public key pk_B for $\mathcal{E}$, runs SIM using these parameters as inputs, and outputs whatever F outputs. If F produces a forgery with probability at least $\frac{\epsilon}{3}$ in SIM, F^* produces a forgery in the underlying SIG signature scheme with probability at least $\frac{\epsilon}{3}$.

Otherwise F produces a forgery with probability less than $\frac{\epsilon}{3}$ in SIM. Then, an algorithm D that distinguishes between the execution of S-SIG and SIM with probability greater than $\frac{2\epsilon}{3}$ can be constructed.

Let $\overline{\text{S-SIG}}$ be an intermediate protocol that differs from the original S-SIG in that in addition to verifying the zero knowledge proofs generated by A, it decrypts A's commitments (having access to A's secret key) and checks explicitly the predicates of the zero knowledge proofs. Then, $\overline{\text{S-SIG}} = \mathsf{B}'$. If we denote by $S \subseteq \{1, \ldots, q_\mathsf{B}\}$ the subset of building blocks in which party A generates a zero knowledge proof, then the distinguishing probability between S-SIG and $\overline{\text{S-SIG}}$ depends only on the soundness error of the proofs from S. We define ρ to be $\mathrm{Dist}_n^D(\text{S-SIG}, \overline{\text{S-SIG}}) = \sum_{i \in S} \text{SERR}_{\Pi_{i\mathsf{A}}}(w_{i\mathsf{A}})$.

We have thus $\frac{2\epsilon}{3} \leq \mathrm{Dist}_n^D(\text{S-SIG},\text{SIM}) \leq \mathrm{Dist}_n^D(\text{S-SIG}, \overline{\text{S-SIG}}) + \mathrm{Dist}_n^D(\overline{\text{S-SIG}},\text{SIM}) \Rightarrow \mathrm{Dist}_n^D(\overline{\text{S-SIG}},\text{SIM}) \geq \frac{2\epsilon}{3} - \rho$.

This implies that there is at least one building block P_i such that D distinguishes between B' and the simulation of $P_1 \parallel \cdots \parallel P_i$, SIM-B, with probability greater than $\frac{2\epsilon}{3q_\mathsf{B}} - \frac{\rho}{q_\mathsf{B}}$. To prove this, we use a hybrid argument: we construct a series of simulators SIM_j, $j = 0, \ldots, q_\mathsf{B}$ such that in SIM_j the first j building blocks are executed as in the protocol $\overline{\text{S-SIG}}$ and all the other building blocks are executed as in the simulation SIM. SIM_0 corresponds to SIM and $\text{SIM}_{q_\mathsf{B}}$ corresponds to $\overline{\text{S-SIG}}$. We can write two inequalities for the distinguishing probability between $\overline{\text{S-SIG}}$ and SIM: $\frac{2\epsilon}{3} - \rho < \mathrm{Dist}_n^D(\overline{\text{S-SIG}},\text{SIM}) \leq \mathrm{Dist}_n^D(\text{SIM}_{q_\mathsf{B}},\text{SIM}_{q_\mathsf{B}-1}) + \cdots + \mathrm{Dist}_n^D(\text{SIM}_1,\text{SIM}_0)$. From this, it follows that there is at least one $i, 1 \leq i \leq q_\mathsf{B}$ such that $\mathrm{Dist}_n^D(\text{SIM}_i,\text{SIM}_{i-1}) > \frac{2\epsilon/3-\rho}{q_\mathsf{B}}$. But $\mathrm{Dist}_n^D(\text{SIM}_i,\text{SIM}_{i-1}) = \mathrm{Dist}_i^D(\mathsf{B}',\text{SIM-B}) > \frac{2\epsilon}{3q_\mathsf{B}} - \frac{\rho}{q_\mathsf{B}}$.

We further distinguish three cases:

1. P_i is one of: `Generate`, `Mult2Secrets`, `InvSecret`, `ModQ`, `Add2Mult` or `Mult2Add`

   Then, from theorem A.2: $\frac{2\epsilon}{3q_\mathsf{B}} - \frac{\rho}{q_\mathsf{B}} \leq \mathrm{Dist}_i^D(\mathsf{B}',\text{SIM-B}) \leq \mathsf{Adv}_{\mathcal{E}}(\mathcal{A}) + \text{SIMERR}_{\Pi_{i\mathsf{B}}}$ and it follows that $\mathsf{Adv}_{\mathcal{E}}(\mathcal{A}) \geq \frac{\epsilon}{3q_\mathsf{B}} - \frac{\rho}{2q_\mathsf{B}}$ or $\text{SIMERR}_{\Pi_{i\mathsf{B}}} \geq \frac{\epsilon}{3q_\mathsf{B}} - \frac{\rho}{2q_\mathsf{B}}$. A simple analysis shows that $\mathsf{Adv}_{\mathcal{E}}(\mathcal{A}) \geq \frac{\epsilon}{6q_\mathsf{B}}$ or $\text{SIMERR}_{\Pi_{i\mathsf{B}}} \geq \frac{\epsilon}{6q_\mathsf{B}}$ or $\rho \geq \frac{\epsilon}{3}$. If the last is true, then there exists at least one $j \in S$ such that $\text{SERR}_{\Pi_{j\mathsf{A}}}(w_j) \geq \frac{\epsilon}{3q_\mathsf{B}}$.
2. P_i is one of: `Add2Secrets`, `AddSecretPub` or `MultSecretPub`

   Then, from theorem A.2: $\frac{2\epsilon}{3q_\mathsf{B}} - \frac{\rho}{q_\mathsf{B}} \leq \mathrm{Dist}_i^D(\mathsf{B}',\text{SIM-B}) \leq \mathsf{Adv}_{\mathcal{E}}(\mathcal{A})$ and it follows that either $\mathsf{Adv}_{\mathcal{E}}(\mathcal{A}) \geq \frac{\epsilon}{3q_\mathsf{B}}$ or $\rho \geq \frac{\epsilon}{3}$.
3. P_i is one of: `RevealMult`, `RevealAdd` or `RevealExp`

   Then, from theorem A.2: $\frac{2\epsilon}{3q_\mathsf{B}} - \frac{\rho}{q_\mathsf{B}} \leq \mathrm{Dist}_i^D(\mathsf{B}',\text{SIM-B}) \leq \text{SIMERR}_{\Pi_{i\mathsf{B}}}$ and it follows that either $\text{SIMERR}_{\Pi_{i\mathsf{B}}} \geq \frac{\epsilon}{3q_\mathsf{B}}$ or $\rho \geq \frac{\epsilon}{3}$.

The conclusion of the theorem follows from the three cases.