SCAPI: The Secure Computation Application Programming Interface[*]

Yael Ejgenberg Moriya Farbstein Meital Levy Yehuda Lindell

Department of Computer Science
Bar-Ilan University, Israel

Library Website: `http://crypto.biu.ac.il/SCAPI`
Contact email for support, bug reports, etc.: `scapi@biu.ac.il`

November 7, 2012

Abstract

Secure two-party and multiparty computation has long stood at the center of the foundations of theoretical cryptography. Recently, however, interest has grown regarding the efficiency of such protocols and their application in practice. As a result, there has been significant progress on this problem and it is possible to actually carry out secure computation for non-trivial tasks on reasonably large inputs. Part of this research goal of making secure computation practical has also involved *implementations*. Such implementations are of importance for two reasons: first, they demonstrate the real efficiency of known and new protocols; second, they deepen our understanding regarding where the bottlenecks in efficiency lie. However, it is very hard to compare between implementations by different research groups since they are carried out on different platforms and using different infrastructures. In addition, most implementations have been carried out without the goal of code reuse, and so are not helpful to other researchers. The difficulty of beginning implementation projects is further compounded by the fact that existing cryptographic libraries (like openSSL, Bouncy Castle, and others) are tailored for tasks like encryption, authentication and key-exchange, and not for secure computation. We have developed SCAPI in order to address these problems. SCAPI is an *open-source* general library tailored for secure computation implementations. Our aim in developing SCAPI has been to provide a flexible and efficient infrastructure for secure computation implementations, that is both easy to use and robust. Great care has been taken in the design of the library, in writing clean code, and in documentation. We hope that this library will be useful to the community interested in implementations of secure protocols, and will help to promote the goal of making secure computation practical.

[*]The SCAPI library was developed through the support of the ERC starting grant “LAST”.

1 Introduction

In the setting of secure multiparty computation, a set of two or more parties wish to compute a joint function of their inputs, while preserving a number of security properties. The most central of these properties are *privacy* (meaning that the parties learn the output $f(x, y)$ but nothing else), *correctness* (meaning that the output received is indeed $f(x, y)$ and not something else), and *independence of inputs* (meaning that neither party can choose its input as a function of the other party's input). Such protocols must maintain their security properties in the presence of adversarial behavior. Three models of relevance are the *semi-honest model* (where the adversary follows the protocol specification exactly but tries to learn more than it should by inspecting the protocol transcript), the *malicious model* (where the adversary can follow any arbitrary polynomial-time strategy), and the *covert model* (where the adversary can follow any arbitrary polynomial-time strategy, but is guaranteed to be caught with good probability if it cheats). Secure computation has broad applicability to electronic voting, auctions, privacy-preserving data mining, private information retrieval, private database queries, and much more.

In the late 1980s and 1990s, the focus of research on secure computation was feasibility [22, 10, 3, 6] and definitional issues [11, 17, 1, 5]. Powerful results were obtained proving that essentially any multiparty computation task can be solved, in a variety of settings. These results showed that it is possible to achieve such secure protocols, but did not demonstrate how to do so efficiently, where by efficiency we mean a protocol that can be implemented and run in practice. This is especially the case when considering malicious adversaries.

Recently, the question of efficiency in secure computation and its potential use in practice has gained much interest, and many works in the past 5 years have focused on this question with impressive success. The aforementioned feasibility results for secure computation are all based on the circuit representation of the function being computed. For many years, it was believed that such protocols belong to the realm of pure theory and cannot be useful in practice. This belief was changed by the Fairplay system [16] which was the first software implementation of the two-party protocol of [22]. Ever since, many protocols were implemented, with security for semi-honest, covert and malicious adversaries; see [2, 4, 13, 14, 15, 21, 18] for just a few examples. The current state-of-affairs today is such that extremely fast implementations exist for semi-honest and covert adversaries, and security in the presence of malicious adversaries is even practical for some non-trivial tasks like private AES computation. These achievements are actually very surprising since until recently, many held the belief that secure computation is a purely theoretical field that is unlikely to be of use in practice. Today, we have protocols that can carry out complex computations on datasets of reasonable size in time that suffices for some practical applications.

In addition to the above progress within the research community, governments and industry have shown much interest recently in the "technology" of secure multiparty computation as a potential tool for their needs (governments wishing to balance the needs of homeland security and privacy, and industry wishing to maximize the utility of their datasets while keeping within privacy law and without provoking customer anger at privacy breaches). Thus, one can argue that "secure computation has come of age", and the SCAPI project was initiated in this context.

Current implementations and SCAPI. To the best of our knowledge, existing implementation projects for secure computation aim at solving a specific problem more efficiently, with higher security and so on. As such, the implementations are specifically tailored to the task being studied, and the code is optimized and tailored to that problem. One of the problems that is arising is that many different research groups are implementing secure computation protocols, with each one working on a different platform with a different setup and completely different code. Thus, it is hard to compare different results, and implementations carried out by one group cannot be used by others. The aim of SCAPI is to provide a general platform of cryptographic primitives for secure computation implementations. As such, primitives have been implemented with the explicit aim of being modular and suitable for general use. In addition, the design is such that primitives can be easily replaced with others. Our primary focus in this project is to provide a *service* that others can use. Thus, the software has been written with the understanding that others will be using it, and so an emphasis has been placed on clean design and coding, documentation, and so on. Finally, SCAPI is intended to be a *long-term open-source project*. Thus, we will continue to update it, by adding additional primitives, improving existing implementations, incorporating additional low-level libraries, fixing bugs that we and others find, and so on. We will also welcome contributions, as long as they meet the software engineering and documentation standards of the library. Finally, we will provide *support* and help to those who wish to use SCAPI. To the best of our knowledge, this makes SCAPI a unique project in the landscape of secure computation implementations.

Licensing. SCAPI is free and is licensed under an adaptation of the MIT license. SCAPI also uses the Crypto++, Miracl, NTL and Bouncy Castle libraries. Please see those projects for any further licensing issues. We kindly request an appropriate citation from any work that uses SCAPI.

Current release – disclaimer. As we have mentioned, SCAPI is a long-term project and we have a long-term commitment to this project. The current release is not perfect. Most outstandingly, only two of the layers are currently in the release, and the third layer that is arguably the most significant (see Section 3) will be released in a few months. In addition, not all of the code has undergone the level of QA that we desire. Nevertheless, we have decided to release what we have so far as a "beta-type release", since our experience shows that it is already very useful. The primitives that will be included in the next release are described in Section 3.3; please also report any bugs discovered so that we can fix them.

Website and support. The SCAPI library can be found at `http://crypto.biu.ac.il/SCAPI`. The complete source code can be found there, as well as documentation (highly detailed design documents, Javadoc documentation of the entire library, and installation notes). Support questions regarding installation, usage, or anything else relevant, as well as bug reports can be sent by email to `scapi@biu.ac.il`.

Contributions. We will be very happy to receive contributions of new code that extends SCAPI (under the condition that it is cleanly written, well documented, and fits into the SCAPI design paradigm). Until the release of the third layer, we will accept contributions relevant only to the first and second layers (this is due to the fact that the design of the third layer components is not yet finalized). Contributions can be small (e.g., a wrapper of AES in the Miracl library and a wrapper of the Intel hardware-based AES), medium-size (e.g., an implementation of a new encryption or signature scheme), and large (e.g., wrapping a new low-level library for existing primitives, or providing new low-level primitives like those provided by the Miracl bilinear maps library).

2 Design Principles

2.1 Basic Principles

The main driving principles behind the design of SCAPI are *flexibility*, *extendibility*, *efficiency*, and *ease of use*. We describe these goals in more detail below. In short, *flexibility* means that protocols implemented using SCAPI use primitives and subprotocols in an abstract way so that they can be easily changed and replaced; *extendibility* means that new implementations of primitives and subprotocols that are added to SCAPI can be used by protocols even if they were previously implemented; *efficiency* means that protocols implemented using SCAPI will have the support of highly efficient low-level libraries; finally, *ease of use* simply means that the library is easy to use, and "speaks the language" of cryptographers.

Before describing the above principles in more details, we begin by explaining why we chose to implement SCAPI in Java. First, Java combines the properties that it is suitable for the development of large projects, and also enables easy and quick development (due to features such as a very large standard library, garbage collection, and no pointer logic). Second, Java is portable and can be run on multiple operating systems and platforms. One setting of importance in this respect regarding secure computation is the ability to run a secure protocol between a mobile device and a PC. Third, there exist very good cryptographic libraries implemented in Java which can be integrated to our API; one example of such a library is Bouncy Castle (`www.bouncycastle.org`). Finally, using the JNI framework, it is possible to integrate existing applications and libraries written in native code.[1] This is of great importance to both flexibility and efficiency, as will be described below.

Flexibility. Protocols for secure computation tasks are typically described as using arbitrary primitives of a lower-level type. For example, a protocol may use oblivious transfer, commitments, and pseudorandom functions, and is secure for any instantiation of these primitives. SCAPI is designed so that in such a case, the protocol is also implemented using *any* oblivious transfer, *any* commitment scheme, and *any* pseudorandom function. The choice of which concrete oblivious transfer, commitment and so on to actually use is determined later on by the high -level application calling the protocol. This flexibility has many advantages. First, it enables a very easy way of comparing the efficiency impact of using different lower-level primitives. For example, the same protocol code can be run with an elliptic curve group and with a prime subgroup of $\mathbb{Z}_p^*$. This is extremely useful, as we demonstrate in Section 4. Furthermore, it should be possible to run the same protocol on a mobile device which supports Java only (by using low-level primitives that are implemented in pure Java), and with higher efficiency on a PC (by using low-level primitives that are implemented in C or C++). In addition, it has ramifications to extendibility and efficiency (e.g., since a new more efficient oblivious transfer protocol can improve the running-time of already-implemented protocols without changing any code). Finally, it is important for ease of use, since cryptographers are used to working at this level of abstraction, and don't have to make decisions about which concrete primitives to use when this can depend on other factors not yet determined.

[1]The JNI framework, or "Java Native Interface" enables Java code running in a Java virtual machine to call native applications (i.e., applications that are specific to a hardware and operating system, like C or C++ code compiled on a certain platform). JNI is typically used for platform-specific features that are not supported in Java, or in order to achieve higher efficiency than is possible with Java code.

This flexibility is achieved by working with *interfaces* that define the functionality of the primitive. For example, a protocol that can use any pseudorandom permutation (PRP) will receive in its constructor any object that implements the PRP interface. Then, an application that uses this protocol, and so calls the constructor for the protocol class, first instantiates an object of one of the classes that implements the PRP interface, and then passes that object to the function (it may be AES, tripleDES or any other PRP that is implemented). We remark that it is also possible to implement a protocol that uses AES specifically. However, once again, here there is flexibility since it may use AES that is implemented in Java from Bouncy Castle or in native code from another library.

Extendibility. An important feature of any infrastructure of this type is the ability to later add different implementations of existing primitives and protocols. By having each algorithm/primitive/protocol type be represented by an interface, we allow future implementations of a given type to be easily used in existing protocols that call it. This is useful for a number of reasons. We illustrate this through two examples. Oblivious transfer is a protocol that is widely used as a building block in many higher-level protocols for secure computation. Until recently, oblivious transfer (with security in the presence of malicious adversaries) was very inefficient. This situation was dramatically changed with the protocols of [20, 12] that require only a small constant number of exponentiations. The design of SCAPI ensures that any new implementations of oblivious transfer that are even more efficient (or advantageous in other ways) can be utilized in all existing protocols that use oblivious transfer. The only change that needs to be made is in the application that calls the protocol, which should now instantiate an object from the new oblivious transfer class and pass it to the constructor of the protocol. The second example relates to the use of new low-level libraries. SCAPI wraps many primitives from existing libraries and enables their use with a common interface. Currently we have wrapped implementations from Bouncy Castle (Java), Crypto++ (C++) and Miracl (C). However, some specific primitives may be more efficiently implemented in other existing libraries, or in the future a better library may become available. In such a case, all protocols implemented above SCAPI can immediately benefit from a new library, once a SCAPI wrapping has been implemented. No existing code needs to be changed and the efficiency or other advantages are immediately utilized by all existing implementations. We will discuss this in more depth in Section 3.1.

Efficiency. One of the main reasons that researchers currently implement new protocols is to demonstrate their efficiency and feasibility in practice. As such, any infrastructure for implementing protocols for secure computation must place a strong focus on efficiency. SCAPI achieves this by wrapping highly-efficient low-level implementations using JNI. For example, elliptic curve operations coded in Java can actually be run using the extremely efficient Miracl library written in C. It is important to stress that protocols implemented in SCAPI are unlikely to be as efficient as specifically tailored and highly optimized low-level code. However, our tests have shown that the difference should not be significant. We believe that in most cases, the benefit here far outweighs the cost; it is much quicker and easier to implement a protocol using SCAPI, and the result is code that has far more flexibility and extendibility, as described above, and is still highly efficient.

Ease of use. There is a significant barrier to begin implementing secure computation protocols. This is due to a number of reasons. First, existing cryptographic libraries focus on the tasks of encryption, key exchange and authentication, and often do not expose the functionality needed for secure computation protocols. Second, these libraries often speak a very different language to that used by cryptographers, and this is a significant obstacle in implementation. Third, existing implementations of secure protocols are typically focused on specific tasks and the code is usually not amenable to reuse by others. SCAPI is designed explicitly for secure computation protocols, but with no specific goal in mind. In fact, we have made every effort to provide as much generality as possible and to not limit ourselves to types of protocols that we currently have in mind.[2] An emphasis was also placed on writing clear code, comments throughout, and in preparing good documentation. Finally, we will provide support to users within the cryptography research community, as much as our resources allow. We believe that these steps will make SCAPI easy to use.

2.2 Security Levels

In many cases, a cryptographic primitive is not just "secure" or "insecure". Rather, it may meet some notion of security and not another. A classic example is encryption, where a scheme can be secure in the presence of eavesdropping adversaries, in the presence of chosen-plaintext attacks or in the presence of chosen-ciphertext attacks. These three levels of security also form a hierarchy (any scheme that is secure in the presence of chosen-ciphertext attacks is secure against chosen-plaintext attacks and so on). The choice of which level of security to require therefore depends on the application. SCAPI includes a mechanism that allows a protocol to work with *any* encryption scheme, on the one hand, but to require that the scheme be CCA-secure, for example, on the other. Likewise, it is possible to implement a protocol using any discrete-log group, and to require that the DDH assumption be hard in that group (or CDH, or just the discrete log problem itself). As a final example, it is possible to construct a protocol that uses a commitment scheme and oblivious transfer, and to require that the commitment scheme be perfectly hiding and the oblivious transfer be UC-secure. This ensures that any application instantiating the protocol will not be able to pass the constructor primitives that do not provide the necessary security guarantees.[3]

This mechanism works by defining a hierarchy of security-level interfaces for the different primitives implemented in SCAPI. For example, a cryptographic hash function can be "collision resistant" or "target collision resistant" (where the latter is implied by the former), and a discrete-log group has security levels DDH, CDH and DLOG (where security under DDH implies security under CDH which implies security under DLOG). Now, the constructor of a protocol that uses a group in which the CDH assumption is hard, can simply verify that the object that it receives (which is guaranteed to be a dlog group) is an "instance of CDH". We remark that since the security-level interfaces form a hierarchy, any dlog group that extends the DDH interface automatically extends the CDH interface and will therefore be accepted by this constructor. Some examples of security level hierarchies are provided in Figure 1.

[2]This does not mean that SCAPI currently implements everything. In the first stage, we have chosen to focus on the tools that are typically used in protocols that are secure without an honest majority. The tools that are used in the setting of an honest majority, like secret sharing and so on, will be considered in future versions.

[3]This is typically the way secure protocols are described in academic papers. One describes the protocol with respect to a commitment scheme and oblivious transfer. Then, the theorem stating security states that "if the commitment scheme is perfectly-hiding and the oblivious transfer protocol is UC-secure, then the protocol meets the specified definition of security".

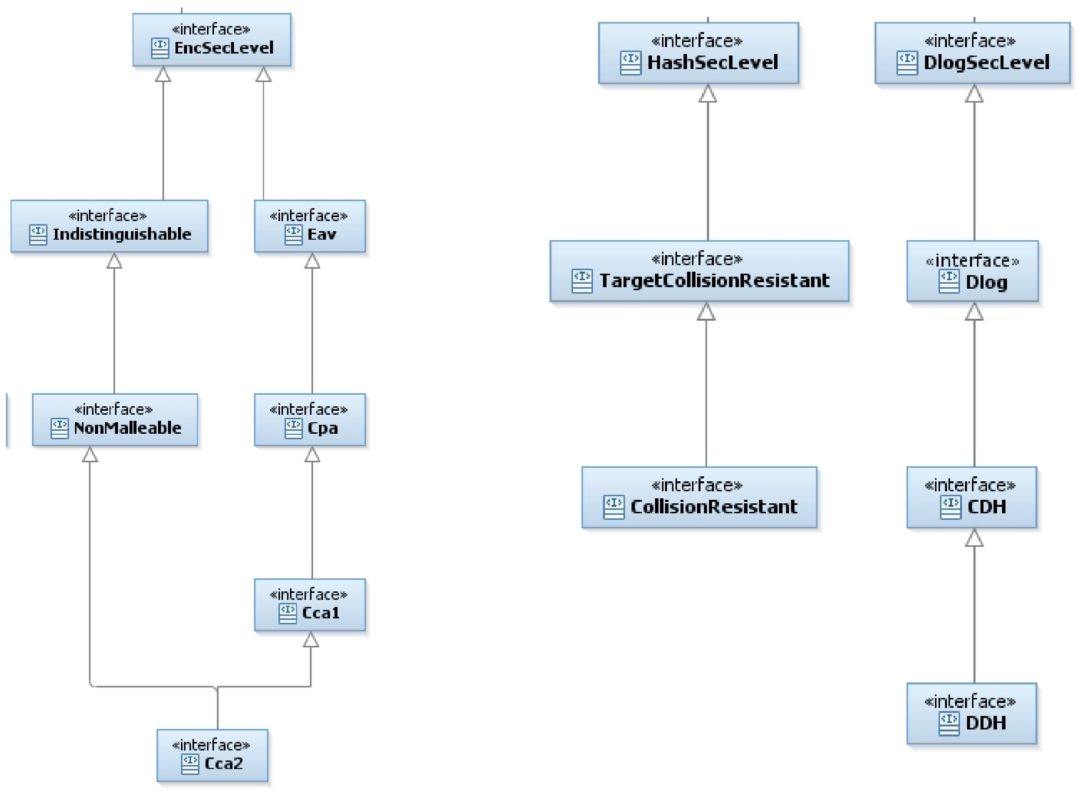

Figure 1: Security Levels Hierarchy for Encryption, Hash Functions and Discrete Log Groups

3 Layers and Primitives

The SCAPI library is divided into three layers. The first layer consists of *low-level primitives* like pseudorandom functions, hash functions and so on. The second layer is comprised of non-interactive schemes, mainly encryption and MAC/signatures. Finally, the third layer contains basic protocols that are commonly used in secure computation, like oblivious transfer, a variety of commitment schemes, Sigma protocols and zero-knowledge, and more. This final layer is the most powerful in some sense. However, in reality, the most important part of the library lies in its design at the lower layers. For example, it is very easy to implement the oblivious transfer protocol of [20] given the discrete-log group functionality in the first layer. In addition to these three main layers, there is an orthogonal communication layer that is used for setting up communication channels and sending messages.

3.1 Layer 1 – Basic Primitives

The first, lowest layer of SCAPI contains basic cryptographic primitives. Most of our code at this level consists of wrapping code from other libraries into a unified format (the exceptions are primitives like HKDF, universal hash function and Luby-Rackoff that are not implemented in standard crypto libraries). Despite this, the basic primitive layer plays a very important role in SCAPI since it provides a common interface for different low-level libraries. This enables higher-level protocols to be designed and implemented independently of any specific low-level library.

Although we have already mentioned this above, we describe the advantages to this approach in more detail here:

1. A single implementation can run on different platforms based on different low-level libraries that are available for that platform.

2. A new low-level library that is more efficient can be wrapped by SCAPI and then all higher-level protocols that are implemented with SCAPI will automatically gain from the efficiency improvement. For example, new Intel chips have AES in hardware and also enable finite field operations in hardware that can speed up elliptic curve operations. By wrapping this capability, any secure computation protocol that uses such functions will immediately run much faster; the only change that needs to be made is in defining the provider for AES and for the elliptic curve operations.

3. The interface of this layer provides different levels of abstraction, as desired. For example, it is possible to implement a protocol that uses *any* pseudorandom function, any pseudorandom permutation, the AES function, or even the AES function as implemented in some specific low-level library (where this latter choice is not desired since it leaves no flexibility). At a later stage, when actually instantiating the protocol, a concrete pseudorandom function must be chosen (like AES). However, since the protocol implementation refers to any pseudorandom function or maybe permutation, the function can be chosen to be anything that meets the definition without modifying the code at all. This level of abstraction is very suitable to the way that cryptographers typically speak and design protocols. In addition, it enables easy comparison of the efficiency ramifications of using different primitives. An example of this is given in Section 4.

The primitives implemented in this layer are: pseudorandom functions and permutations (of multiple types with fixed input and output length, varying input and output length, and so on), cryptographic hash functions, universal hash functions, trapdoor permutations, pseudorandom generators, key derivation functions (a.k.a. extractors), and discrete log groups. The discrete log group is the most significant since it exposes the functionality that cryptographers expect from a generic group of this kind, without linking it to the specific needs of encryption and signing as can be found in most libraries. In addition, the same functionality is provided for groups based on $\mathbb{Z}_p^*$ and elliptic curve groups of different types; see Section 4 for an example of this. The functionality provided by the discrete log implementation is novel with respect to other existing libraries, and we view its contribution as significant in and of itself.

3.2 Layer 2 – Non-Interactive Schemes

The second layer consists of non-interactive cryptographic schemes. Specifically, this layer contains symmetric and asymmetric encryption, message authentication codes and digital signatures.[4] Regarding asymmetric encryption, SCAPI supports RSA-OAEP (from Bouncy Castle and from Crypto++), El-Gamal (over any discrete log group), Cramer-Shoup [7] (over any discrete log group), and Damgård-Jurik additively homomorphic encryption [8] (which is an extension of Paillier [19]). We remark that both ElGamal and Cramer-Shoup can receive group elements or byte

[4]Commitment schemes are part of the third layer since some are interactive and some are non-interactive.

arrays as plaintext; the former case is often needed in protocols where the algebraic structure of the ciphertext is needed for efficiently proving statements in zero knowledge.

Regarding symmetric encryption, message authentication codes and digital signatures, standard schemes are currently supported (e.g., AES with CBC and counter modes, CBC-MAC, DSA and RSA signatures, etc.). However, according to the design paradigm of the library, protocols that use encryption should work at the abstract level of what is required. For example, a protocol requiring CCA2-secure asymmetric encryption should accept any asymmetric encryption scheme, and then check that its security level is CCA2. This enables the incorporation of different schemes once implemented in SCAPI.

The security levels for encryption provided are Eavesdropping, CPA, CCA1, and CCA2, and orthogonally indistinguishability and non-malleability. For example, one can specify non-malleable under CPA, or indistinguishability under CCA1; needless to say, any scheme that is CCA2 is automatically both indistinguishable and non-malleable; see Figure 1.

3.3 Layer 3 – Interactive Protocols (not in the current release)

The third layer of SCAPI contains interactive protocols and schemes that are widely used in protocols for secure computation. The main schemes are:

- *Sigma protocols:* SCAPI contains over 10 common Sigma protocols (e.g., discrete log, Diffie-Hellman tuple, etc.). In addition, the following operations on *arbitrary* Sigma protocols are included: AND of multiple statements, OR of two or many statements, transformation to zero-knowledge, transformation to zero-knowledge proof of knowledge, Fiat-Shamir transformation to non-interactive zero-knowledge, and transformation to UC-secure zero-knowledge proof of knowledge.

- *Commitments:* SCAPI includes Pedersen commitments, ElGamal commitments, Hash-based commitments, equivocal commitments, extractable commitments, fully trapdoor commitments, homomorphic commitments, non-malleable commitments and UC-secure commitments.

- *Oblivious transfer:* Many oblivious transfer protocols are implemented in SCAPI, with security in the presence of semi-honest and malicious adversaries. For the case of malicious adversaries, protocols achieving privacy only, one-sided simulation, full simulation-based security, and UC-security are included. Oblivious transfer extension for semi-honest adversaries is also included.

- *Garbled circuit:* A number of Yao garbled-circuit constructions are implemented. There is a basic construction that can work with any double-encryption scheme, and some more optimized constructions (e.g., using the free XOR technique). The design is also such that a circuit can be broken up into layers and processed in parts, if desired.

- *Miscellaneous:* In addition to the above, SCAPI also has protocols for tossing a single bit and a string (with full simulatability or one-side simulatability), for pseudorandom function evaluation, and more.

The above will be included in the first release of the 3rd layer of SCAPI. Our aim is to increasingly add functionality as time goes on.

We stress that although this layer is the most significant in terms of the functionality it provides, the first layer is the most important in terms of providing the necessary infrastructure for implementing protocols for secure computation. It is very easy to implement oblivious transfer protocols, Sigma protocols and so on, given the first layer. The first release of SCAPI does *not* yet contain the third layer. We hope to release this within the coming few months.

3.4 The Communication Layer

SCAPI comes with a built in communication layer for setting up communication channels between two or more parties. In future releases, the communication layer will include advanced features like secure broadcast between multiple parties and more. Currently, it enables the generation of a plain, encrypted, authenticated, or encrypted-then-authenticated TCP channel between two parties.[5]

A communication channel has two main functions, send and receive, and these are used by the parties for "communicating". The main issue that arises in implementing such a channel is what can be sent over such a channel. Recall that cryptographic objects are often not simple types (e.g., a Cramer-Shoup ciphertext consists of group elements, and possibly also a byte array). Furthermore, at the level of abstraction that SCAPI works, a protocol wishing to send an object may not know what that object actually contains. For example, a group element can be a single BigInteger (in the case of a $\mathbb{Z}_p$-based group), or a pair of BigIntegers (in the case of an elliptic curve group).[6] We therefore defined a mechanism enabling any object in SCAPI to be sent or received. Fortunately, Java already comes with a built-in mechanism for sending objects that were generated by one VM to another VM. This mechanism is called serialization, and is used to convert a Java object into a form that can be stored or transmitted over a communication channel. Serialization converts an object into a stream of bytes containing all of the information necessary to recreate the object at a later time, e.g., after communication. In order to use this mechanism, all that needs to be done is to make all classes that contain data that may need to be sent serializable.[7]

Primitive types like int and byte, and basic classes like arrays, String and BigInteger are already serializable in Java. SCAPI objects that need to be sent include keys, plaintexts, ciphertexts, signatures, group parameters, group elements and trapdoor-permutation elements. These last two types cause a problem since they are only defined in the *context* of their associated group or trapdoor permutation (e.g., a point (x, y) may be in the group defined by one elliptic curve but not another). Due to this, SCAPI only allows the construction of a group element from within the class that represents the actual group; likewise, a RabinElement or RSAElement can only be constructed from within the actual corresponding permutation (specifically, the generateElement function needs to be called from within the relevant TPPermutation or dlog group). This design decision ensures an explicit connection between elements of the above type and the group or permutation that they are associated with.

Due to the above, SCAPI does not define the actual element object to be serializable (since this would either involve sending the group information explicitly with each element, which is wasteful,

[5]In actuality, the very first release only has a plain channel; the addition of encryption and authentication will be included in an additional release very soon.

[6]To be exact, a group element is neither a single BigInteger nor a pair of BigIntegers, but rather is a separate class containing this data.

[7]Many consider using serialization a security risk since the entire class contents are revealed. In the context of SCAPI, we do not consider this to be a concern. First, anything sent is supposed to be fully revealed to the recipient, or else it should not be sent. Second, if something should not be viewed by an eavesdropper on the channel, then an encrypted channel should be used.

or would require enabling the construction of an element independent of its context). Rather, a separate class containing the *data only* is used for communication. This design has another advantage in that it allows two communicating parties to use different providers. For example, consider a secure computation protocol that is run between a PC and a mobile device, where the latter only supports Java. If they communicated actual Java objects, then the PC would either have to use only Java-based classes (e.g., an Elliptic curve library from Bouncy Castle) which is much slower, or would have to explicitly convert every element received. By only sending data, each party can use whatever library it prefers. In order for this to work, all SCAPI objects which are not basic types have a function called generateSendableData() that returns a data object that is serializable, and a function for generating the appropriate object from the sendable-data object. For example, the code for sending a dlog group element is channel.send(myPoint.generateSendableData()) and the code for receiving it is myPoint = myDlogGroup.generateElement(channel.receive()).

We remark that this issue arises also with other types that can contain dlog elements and the like. Thus, asymmetric ciphertexts must also implement generateSendableData and construction from such data, since schemes like ElGamal and Cramer-Shoup contain dlog elements.

4 Example Usage

In this section, we present an example of how the SCAPI library works. The example that we use is the Cramer-Shoup encryption scheme [7]. This implementation is part of the second layer of the library, but also demonstrates how to *use* the library externally. The Cramer-Shoup scheme is a CCA2-secure public-key (asymmetric) encryption scheme. As such, the CramerShoup interface extends the asymmetric-encryption and CCA2 interfaces.[8]

```
public interface CramerShoupDDHEnc extends AsymmetricEnc, Cca2 {
}
```

This defines its type and enables a protocol/scheme that is secure when using *any* CCA2-secure public-key encryption scheme to use Cramer-Shoup.

Constructor: Cramer-Shoup encryption works over any group in which the DDH problem is assumed to be hard, and also uses a collision-resistant hash function. Thus, the constructor for the scheme receives a “discrete-log group” and a “cryptographic hash function” and verifies that the former has security level DDH and that the latter is collision resistant. This ensures that one cannot instantiate Cramer-Shoup with a weaker hash function, or with a group in which DDH is not hard (e.g., some Bilinear group). The constructor is shown below.

```
public CramerShoupAbs(DlogGroup dlogGroup, CryptographicHash hash, SecureRandom random){
  //The Cramer-Shoup encryption scheme must work with a Dlog Group that has DDH security level
  //and a Hash function that has CollisionResistant security level. If any of this conditions is not
  //met then cannot construct an object of type Cramer-Shoup encryption scheme; therefore throw exception.

  if(!(dlogGroup instanceof DDH)){
     throw new IllegalArgumentException("The Dlog group has to have DDH security level");
     }
```

[8]There are also `Indistinguishable` and `NonMalleable` interfaces for encryption; however, since any CCA2 scheme is both indistinguishable and non-malleable there is no need to extend these as well.

```
  if(!(hash instanceof CollisionResistant)){
     throw new IllegalArgumentException("The hash function has to have CollisionResistant security level");
     }

  // Everything is correct, then sets the member variables and creates object.
  this.dlogGroup = dlogGroup;
  qMinusOne = dlogGroup.getOrder().subtract(BigInteger.ONE);
  this.hash = hash;
  this.random = random;
}
```

Encryption: The code for the Cramer-Shoup encryption procedure can be found below. Observe that the operations on the discrete-log group are completely abstract, and are independent of its actual structure. This enables the same code to be used for groups based on $\mathbb{Z}_p^*$ or on elliptic curves, and so on. This is most evident in the `mapAnyGroupElementToByteArray` function which is needed to translate a group element into a string so that it can be input to the hash function (because the latter works only on byte arrays); since we have no information on the format of a group element, a special function is needed for this benign purpose.

Note that the encryption procedure here receives a plaintext that is a *group element* and not a byte array; there is a different class that is used for encrypting a byte array. This difference is of significance when the algebraic properties of the ciphertext need to be preserved (e.g., in order to prove statements in zero-knowledge regarding the plaintext).

```
public AsymmetricCiphertext encrypt(Plaintext plaintext){
 /*
  * Choose a random  r in Zq
  * Calculate   u1 = g1^r
  *             u2 = g2^r
  *             e  = (h^r)*msgEl
  * Convert u1, u2, e to byte[] using the dlogGroup
  * Compute alpha  - the result of computing the hash function on the concatenation u1+u2+e.
  * Calculate v = c^r * d^(r*alpha)
  * Create and return an CramerShoupCiphertext object with u1, u2, e and v.
  */
  if (!isKeySet()){
     throw new IllegalStateException("in order to encrypt a message this object must be initialized
     with public key");
     }
  if (!(plaintext instanceof GroupElementPlaintext)){
     throw new IllegalArgumentException("plaintext should be instance of GroupElementPlaintext");
     }
  GroupElement msgElement = ((GroupElementPlaintext) plaintext).getElement();

  BigInteger r = chooseRandomR();      //Choose a random value between 0 and q-1 (q = group order)
  GroupElement u1 = calcU1(r);         //Compute g1^r
  GroupElement u2 = calcU2(r);         //Compute g2^r
  GroupElement hExpr = calcHExpR(r);   //Computer h^r
  GroupElement e = dlogGroup.multiplyGroupElements(hExpr, msgElement);

  byte[] u1ToByteArray = dlogGroup.mapAnyGroupElementToByteArray(u1);
  byte[] u2ToByteArray = dlogGroup.mapAnyGroupElementToByteArray(u2);
  byte[] eToByteArray = dlogGroup.mapAnyGroupElementToByteArray(e);
  //Calculates the hash(u1 + u2 + e).
  byte[] alpha = calcAlpha(u1ToByteArray, u2ToByteArray, eToByteArray);

  //Calculates v = c^r * d^(r*alpha).
  GroupElement v = calcV(r, alpha);

  //Creates and return an CramerShoupCiphertext object with u1, u2, e and v.
  CramerShoupOnGroupElementCiphertext cipher = new CramerShoupOnGroupElementCiphertext(u1, u2, e, v);
  return cipher;
}
```

The functions `calcU1`, `calcU2`, and so on compute exponentiations, as expected. For example:

```
  protected GroupElement calcU1(BigInteger r) {
    return dlogGroup.exponentiate(publicKey.getGenerator1(), r);
  }
```

These are written as functions since there are two versions of Cramer-Shoup, one that receives a group element for encryption and one that receives a byte array, and writing these as functions prevents code duplication.

A demonstration of modularity. As we have described, the implementation of the Cramer-Shoup encryption scheme is completely independent of the actual discrete log group and hash function. Thus, it is very easy to write a simple loop that instantiates the scheme with different groups and measures the running time. We stress that this holds both with respect to different group types (e.g., the order-q subgroup of $\mathbb{Z}_p$ where $p = 2q + 1$ versus different elliptic curve groups), and with respect to the provider (e.g., elliptic curve routines of Bouncy Castle in Java, versus elliptic curve routines of Miracl that are written in C). This example clearly demonstrate our above discussion on flexibility and extendibility: this given Cramer-Shoup implementation will be able to work with any elliptic curve library that will be incorporated into SCAPI in the future. In addition, the same code should run on a mobile device (which only supports Java and so would use a Bouncy Castle based discrete log group) and on a PC where faster libraries like Miracle can be used. We now present the code for running this test and then the results.

We do not provide all of the code here, but rather just the main parts. The main function reads the parameters of the test from a configuration file and executes the test for each set of parameters. The main loop appears below:

```
public static void main(String[] args) throws FactoriesException {
   ...
   // Get parameters from config file:
   CramerShoupTestConfig[] config = readConfigFile();
   ...
   for (int i = 0; i < config.length; i++) {
      result = runTest(config[i]);
      out.println(result);
      System.out.println(result);
   }
   ...
}
```

Three example sets of parameters, as they appear in the configuration file, are as follows:

```
     dlogGroup = DlogZpSafePrime
     dlogProvider = CryptoPP
     algorithmParameterSpec = 1024
     hash = SHA-256
     providerHash = BC
     numTimesToEnc = 1000

     dlogGroup = DlogECFp
     dlogProvider = BC
     algorithmParameterSpec = P-224
     hash = SHA-1
     providerHash = BC
     numTimesToEnc = 1000

     dlogGroup = DlogECFp
     dlogProvider = Miracl
     algorithmParameterSpec = P-224
     hash = SHA-1
     providerHash = BC
     numTimesToEnc = 1000
```

As you can see, the parameters determine which "discrete-log group" to use (e.g., $\mathbb{Z}_p$-based group, elliptic curve group over a prime-order field, etc.), which provider of that group to use (e.g., Crypto++, Bouncy Castle (BC) or Miracl), and whatever additional parameters needed (the size of the prime p in $\mathbb{Z}_p$-based groups or the curve name for elliptic curve groups). Likewise, the hash function and provider are specified as well. We now present the function that carries out the actual work of encryption. This function also demonstrates how to generate an encryption object, a random key, a random group element, and so on.

```
static public String runTest(CramerShoupTestConfig config) throws FactoriesException{
   DlogGroup dlogGroup;
   //Create the requested Dlog Group object. Do this via the factory.
   //If no provider specified, take the SCAPI-defined default provider.
   if(config.dlogProvider != null){
      dlogGroup = DlogGroupFactory.getInstance().getObject(config.dlogGroup+
                                    "("+config.algorithmParameterSpec+")", config.dlogProvider);
   }else {
      dlogGroup = DlogGroupFactory.getInstance().getObject(config.dlogGroup+
                                    "("+config.algorithmParameterSpec+")");
   }

   CryptographicHash hash;
   //Create the requested hash. Do this via the factory.
   if(config.hashProvider != null){
      hash = CryptographicHashFactory.getInstance().getObject(config.hash, config.hashProvider);
   }else {
      hash = CryptographicHashFactory.getInstance().getObject(config.hash);
   }

   //Create a random group element. This element will be encrypted several times as specified in
   //config file and decrypted several times
   GroupElement gEl = dlogGroup.createRandomElement();

   //Create a Cramer Shoup Encryption/Decryption object. Do this directly by calling the relevant
   //constructor. (Can be done instead via the factory).
   ScCramerShoupDDHOnGroupElement enc = new ScCramerShoupDDHOnGroupElement(dlogGroup, hash);
   //Generate and set a suitable key.
   KeyPair keyPair = enc.generateKey();
   try {
      enc.setKey(keyPair.getPublic(),keyPair.getPrivate());
   }  catch (InvalidKeyException e) {
      e.printStackTrace();
   }

   //Wrap the group element we want to encrypt with a Plaintext object.
   Plaintext plainText = new GroupElementPlaintext(gEl);
   AsymmetricCiphertext cipher = null;

   //Measure the time it takes to encrypt each time. Calculate and output the average running time.
   long allTimes = 0;
   long start = System.currentTimeMillis();
   long stop = 0;
   long duration = 0;

   int encTestTimes = new Integer(config.numTimesToEnc).intValue();
   for(int i = 0; i < encTestTimes; i++){
      cipher = enc.encrypt(plainText);
      stop = System.currentTimeMillis();
      duration = stop - start;
      start = stop;
      allTimes += duration;
   }
   double encAvgTime = (double)allTimes/(double)encTestTimes;

   //Repeat for decryption...

   ...

   return result;
}
```

See Table 1 for the results of the test (the times are the average of 1000 encryptions and decryptions). This example demonstrates the power of SCAPI. In more advanced protocols, where the effect on efficiency of changing the underlying primitives may be unclear, this is a very important tool.

Dlog Group Type	**Dlog Provider**	**Dlog Param**	**Hash Function**	**Hash Provider**	**Encrypt Time (ms)**	**Decrypt Time (ms)**
DlogZpSafePrime	CryptoPP	1024	SHA-256	BC	6.072	3.665
DlogZpSafePrime	CryptoPP	2048	SHA-256	BC	43.818	26.289
DlogECFp	BC	P-224	SHA-1	BC	54.171	31.662
DlogECF2m	BC	B-233	SHA-1	BC	107.316	65.185
DlogECF2m	BC	K-233	SHA-1	BC	25.292	14.886
DlogECFp	Miracl	P-224	SHA-1	BC	6.571	3.929
DlogECF2m	Miracl	B-233	SHA-1	BC	5.819	3.652
DlogECF2m	Miracl	K-233	SHA-1	BC	2.753	1.787

Table 1: Running times for Cramer-Shoup using Different Groups and Libraries

References

[1] D. Beaver. Foundations of Secure Interactive Computing. In *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 377–391, 1991.

[2] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a System for Secure Multiparty Computation. In the *ACM Conference on Computer and Communications Security 2008*, pages 257–266, 2008.

[3] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In 20*th STOC,* pages 1–10, 1988.

[4] P. Bogetoft, D. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krigaard, J. Nielsen, J.B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. Secure Multiparty Computation Goes Live. In *Financial Crypto*, 2009.

[5] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[6] D. Chaum, C. Crépeau and I. Damgård. Multi-party Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.

[7] R. Cramer and V. Shoup, A Practical Public Key Cryptosystem Secure Against Adaptive Chosen Ciphertext Attacks. In *CRYPTO'98*, Springer (LNCS 1462), pages 13–25, 1998.

[8] I. Damgård and M. Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. *Public Key Cryptography* (PKC), Springer (LNCS 1992), pages 119–136, 2001.

[9] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications.* Cambridge University Press, 2004.

[10] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In 19*th STOC,* pages 218–229, 1987. For details see [9].

[11] S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90,* Springer-Verlag (LNCS 537), pages 77–93, 1990.

[12] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*, Springer 2010.

[13] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. *USENIX Security Symposium*, 2011.

[14] B. Kreuter, A. Shelat and C.H. Shen. Towards Billion-Gate Secure Computation with Malicious Adversaries. *IACR Cryptology ePrint Archive 2012*, report 179, 2012.

[15] Y. Lindell, B. Pinkas and N.P. Smart. Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries. In *SCN 2008*, Springer (LNCS 5229), pages 2–20, 2008.

[16] D. Malkhi, N. Nisan, B. Pinkas and Y. Sella. Fairplay – A Secure Two-Party Computation System. Proceedings of *Usenix Security Symposium*, pages 287–302, 2004.

[17] S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 392–404, 1991.

[18] J.B. Nielsen, P.S. Nordholt, C. Orlandi and S.S. Burra. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO 2012*, Springer (LNCS 7417), pages 681–700, 2012.

[19] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT 1999*, Springer (LNCS 1592), pages 223–238, 1999.

[20] C. Peikert, V. Vaikuntanathan and B. Waters. A Framework for Efficient and Composable Oblivious Transfer. In *CRYPTO 2008*, Springer-Verlag (LNCS 5157), pages 554–571, 2008.

[21] B. Pinkas, T. Schneider, N.P. Smart, S.C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT 2009*, Springer (LNCS 5912), pages 250–267, 2009.

[22] A. Yao. How to Generate and Exchange Secrets. In 27*th FOCS*, pages 162–167, 1986.