Charm: A Framework for Rapidly Prototyping Cryptosystems

Joseph A. Akinyele* Matthew Green* Avi Rubin*

Abstract

We describe Charm, an extensible framework designed for rapid prototyping of cryptographic systems that utilize the latest advances in cryptography, such as identity and attribute-based encryption, as well as the traditional cryptographic functions. Charm is designed to minimize code complexity, promote code re-use, and to automate interoperability, while not compromising on efficiency.

Charm was designed from the ground up to support the implementation of advanced cryptographic schemes. It includes support for multiple cryptographic settings, an extensive library of re-usable code, and a *protocol engine* to aid in the development of interactive protocols. Our framework also provides a series of specialized tools that enable different cryptosystems to interoperate.

We implemented over twenty cryptographic schemes using Charm, including some new ones that to our knowledge have never been built in practice. This paper describes our modular architecture, which includes a built-in benchmarking module that we use to compare the performance of primitives written in Python to comparable C implementations. We show that in many cases our techniques result in a potential order of magnitude decrease in code size, while inducing an acceptable performance impact.

Keywords: applied cryptography, protocols, software

1 Introduction

Recent advances in cryptography utilize new settings, such as bilinear groups and lattices. These advances include new paradigms for securely processing and protecting access to sensitive information such as identity and attribute-based encryption, privacy-preserving primitives such as group signatures and anonymous credentials, and techniques for computing on encrypted data. The new cryptographic mechanisms are increasingly complex. As these mathematically-based schemes become more intricate, adoption by the systems security community is likely to decrease, unless developers are provided with proper code libraries.

While there have been some efforts to produce *research* prototype implementations of the new primitives, there is a need for a robust, modular, efficient and usable framework for developers to fully utilize the capabilities of the latest results from the theoretical cryptography community. The framework should provide developers with the proper interface to shelter them from the low-level details of the algorithms and the mathematical operations, while providing a powerful and flexible API to rapidly implement new protocols. We believe that developers should utilize high-level languages with advanced type checking and memory management features.

In performance-intensive fields such as animation and game development the trend seems to be towards a hybrid architecture, where an optimized, native engine centralizes performance-critical operations, and developers are encouraged to implement the remaining code in a high-level language (*e.g.,* [4, 34]). Similarly, a framework for implementing security systems based on advanced (and standard) cryptographic protocols and algorithms should be designed this way.

There have been several elegant implementations of a small number of new primitives [52, 43, 11] as well as some tools for protocol development [45, 5, 44, 32, 37, 39]. These systems serve their special purposes well, but are not interoperable by design, and so developers wishing to build a system using multiple different primitives must write awkward *glue* code to piece their implementation together.

*Johns Hopkins University, {jakinye3,mgreen,rubin}@cs.jhu.edu

In this paper, we present Charm [1], a new, extensible and unified framework for *rapidly prototyping* experimental cryptosystems. Our system is built around a well-supported high level language (Python), and is designed to reduce development time and code complexity while promoting component re-use. While Charm is based on an interpreted language, all of the performance-intensive mathematical operations are implemented as native modules. We show that for some applications, even though the protocol logic is written with the advantages of higher level languages, performance can be comparable to C implementations.

As shown in Figure 1, Charm provides a number of components to facilitate the development of cryptographic schemes and protocols.

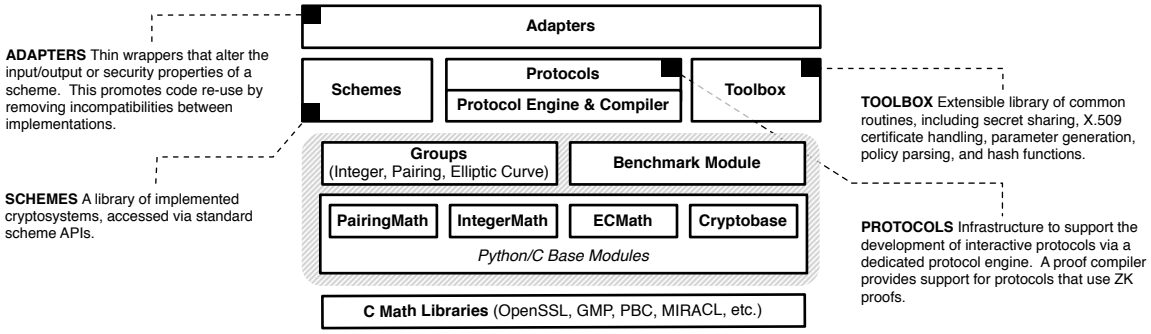


Figure 1: Overview of the Charm architecture.

Extensible code library. Charm provides an extensive library of cryptographic building blocks. Supported operations include secret sharing, interactive and non-interactive zero-knowledge proof protocols, key and ciphertext serialization, certificate handling, and parameter generation. The library is designed to be easily extended when new primitives are invented.

Protocol engine and protocol compilers. Charm contains all of the necessary infrastructure to rapidly implement interactive cryptographic protocols. The Charm *protocol engine* embeds necessary logic required to execute a protocol, including message serialization, data transmission, state transitions, error handling, and the execution of subprotocols. For protocols involving zero-knowledge proof statements, Charm also embeds a compiler that converts Camenisch-Stadler form proof statements into Python subprotocols that can be dynamically executed by the interpreter.

Adapter architecture. To address the incompatibilities between different schemes, cryptosystems are annotated with details of their implementations. We use annotations and introspection to 'label' cryptographic schemes with properties including input and output specifications (*e.g.,* plaintext and ciphertext space), performance, and even security-related properties such as the complexity assumptions used in a security proof. Adapters are thin, object-oriented wrappers that provide for conversion among incompatible schemes based on their labels.

Benchmarking and profiling tools. The framework includes sophisticated benchmarking tools, particularly useful for researchers, to measure the bandwidth and time complexity of implementations (in terms of operations conducted or computation time). The benchmarking module can be extended to record custom measurements, *e.g.,* custom subroutines.

Application embedding. For interoperability, constructions implemented in our framework can be easily incorporated into C/C++ programs by embedding the Python interpreter and the necessary supporting modules.

We implemented several recent cryptographic results from the research literature using Charm. In addition to building protocols for which we believe there are no existing implementations, for the purpose of comparison, we also coded several constructions that have existing C implementations. Our programs are

dramatically simpler than prior ones; in some cases the code size is an order of magnitude smaller, with only a slight performance penalty. We believe that for many applications, even a more significant decrease in the speed of the running code would be a worthwhile price to pay for a large reduction in the size and complexity of the code, both from a security perspective as well as code maintainability.

2 Architecture

This section describes the architecture of Charm and provide details about its components, as shown in Figure 1.

Components. We now describe the building blocks of the Charm framework. The lower level components, at the bottom of Figure 1 are optimized for efficiency, while the ones at the top focus on ease of use and interoperability. One of the primary drivers of our architecture is our objective to simplify the code written by developers who utilize the framework. Our modular component architecture reflects this.

Base Modules. Charm contains four base modules that implement the core cryptographic routines. For performance reasons these modules are written in C and include integermath, ecmath (elliptic curve subgroups), and pairingmath.[1] The cryptobase module provides efficient implementations of basic cryptographic primitives such as hash functions and block ciphers. These modules include code from standard C libraries including libgmp, OpenSSL, libpbc and PyCrypto [30, 52, 43, 42].[2] To maximize code readability, the module interfaces employ language features such as operator overloading. Finally, Charm provides high-level Python interfaces for constructs such as algebraic groups and fields.

Cryptographic Toolbox. The base modules implement only those lower level routines where implementation in C is crucial for performance. Charm also provides an extensive Toolbox of useful Python routines including secret sharing, encryption padding, group parameter generation, certificate handling, message encoding, and ciphertext parsing. We are continuously adding routines to the Toolbox, and we hope that future releases will include contributions from external developers.

Scheme Interfaces. To facilitate code re-use, Charm provides a set of APIs for common cryptographic primitives such as digital signatures, bit commitment, encryption, and related functions. Schemes with identical APIs are identified and are interchangeable in our framework. Thus, for example, DSA can be used instead of RSA-PSS within a larger protocol with a simple, almost trivial change to the code.

Scheme interfaces are implemented using standard object-oriented programming techniques. The current Charm interface hierarchy appears in Figure 2. This list is sufficient for the schemes we have currently implemented (see Figure 5), but we expect it to expand with the addition of new cryptosystems.

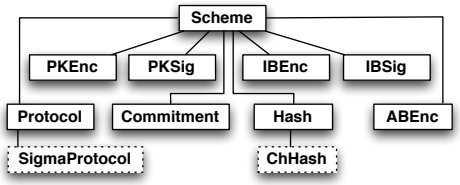


Figure 2: Listing of scheme types defined in Charm. Subtypes are indicated with dotted lines.

[1] A dedicated module to support lattice-based cryptography is in preparation for a future release.

[2] We plan to remove the dependencies on libgmp and PyCrypto and to add optional support for the MIRACL library [50] in an upcoming release.

Scheme Annotation and Adapters. In practice, implementations of different cryptosystems may be incompatible even if their APIs are the same. For example, two systems might have different input and output requirements. Consider that many public key encryption schemes require plaintexts to be pre-encoded as elements of a cyclic group $\mathbb{G}$, or as strings of some fixed size. These requirements frequently depend on how the scheme is configured, *e.g.,* depending on parameters used. Different developers are unlikely to make all of the same choices in their implementations, so even if they build their code with a standard API template, their systems are unlikely to interoperate cleanly.

More subtle incompatibilities may arise when schemes of a given class provide differing security guarantees: for example, public-key encryption schemes may provide either IND-CPA vs. IND-CCA2 security. These properties become relevant whenever the scheme is used as a building block for a more complex protocol.

To address these issues, Charm provides a mechanism to annotate schemes with meta-information that describes their relevant characteristics, including (but not limited to) input/output space, security definition, complexity assumptions, computational model, and performance characteristics. Wherever possible this information is derived automatically, *e.g.,* from the scheme type or function definitions. Other characteristics must be specified by the developer via a standard annotation interface. This information can be programmatically evaluated at any point.

Capability matching. Charm uses this meta-information to facilitate compatibility among schemes. First, it provides tools to programmatically interrogate a scheme in order to determine whether the scheme satisfies certain criteria. This makes it easy to substitute schemes into a protocol at runtime, since the protocol can simply specify its requirements (*e.g.,* EU-CMA signature scheme) and Charm will ensure that they are met. To make this workable, Charm includes a dictionary of security definitions and complexity assumptions, as well as the implications between them.[3]

Adapters. Charm includes *adapters* to handle incompatibilities between schemes. Adapters are code wrappers implemented as thin classes. They permit developers to bridge the gap between primitives with disparate message/output spaces or security requirements. In our experience so far, the most common use of adapters is to convert an input type so that a scheme can be used for a specific application. For example, we use adapters to encode messages or in the case of hybrid encryption, to expand the message space of a public key encryption scheme.

Adapters can perform even more sophisticated functions, such as modifying a scheme's security properties. In Figure 3 we illustrate an adapter using a hash function to perform a conversion from a selectively-secure IBE into one that is adaptively secure (note here that the hash function is modeled as a random oracle).

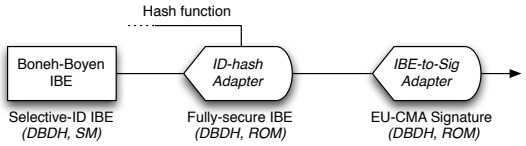


Figure 3: **Example of an adapter chain converting the Boneh-Boyen selective-ID secure IBE [14] into a signature scheme using Naor's technique [16]. The scheme carries meta-information including the complexity assumptions and computational model used in its security proof.**

Adapters can also combine schemes to produce entirely different cryptosystems. Thus, there are *implicit* schemes in Charm that do not physically appear in the scheme library.

Protocol Engine Interactive protocols implementations must include network communications, data serialization, error handling and state machine transition. Charm simplifies development by providing all of these features as part of a re-usable *protocol engine*. Thus, an implementation in our framework consists

[3]Thus a protocol that requires only an EU-CMA signature scheme will be satisfied if instantiated with an SU-CMA signature, but not vice versa.

of a list of parties, a description of states and transitions, and the core logic for each state. Serialization, transmission and error handling are handled at the lower levels and are available for free to the developer.

Our protocol engine provides native support for the execution of sub-protocols and supports recursion. We have found subprotocols to be particularly useful in constructions that use interactive proofs of knowledge.

Given a protocol implementation, an application executes it by selecting a party type and optional initial state, and by providing a collection of socket connections to the remote parties. Sockets in Python are an abstract interface and can be extended to support various communication mechanisms.

2.1 Benchmarking System

To measure the performance of a prototype implementation, Charm incorporates a native benchmark module to collect information on a scheme's performance. This module collects and aggregates statistics on a set of operations defined by the user. All of the operations in the core modules are instrumented separately, allowing for detailed profiling including total operation counts, average operation time for various critical operations, and network bandwidth (for interactive protocols). Users can define their own measurements within a given implementation (*e.g.,* a scheme or subroutine). When these measurements involve timing, the benchmarking module automatically performs and collects timing information. Many of our experiments in Section 4 were performed using the benchmarking system. The benchmarking system is easy to switch on or off and has no impact on the system when it is not in use. An example of using the benchmarking system is provided in Section 3.3.

2.2 ZKP Compiler

Many advanced cryptographic protocols (*e.g.,* [20, 15, 21]) employ zero-knowledge or witness-indistinguishable proofs as part of their protocol structure. The notation of Camenisch and Stadler [22] has become the de facto standard in the Crypto literature. This notation, while elegant, stands in for a complex interactive or non-interactive subprotocol that must be derived before the base protocol can be implemented.

To handle such complex protocols, Charm includes an automated compiler for common ZK proof statements. Such compilers are not new, and have been implemented by Meiklejohn *et al.* (ZKPDL) [45] and Bangerter *et al.* (CACE) [8]. Our compiler interprets Camenisch-Stadler style proof descriptions at runtime and derives an executable honest-verifier protocol. At present our compiler handles a limited set of discrete-log statements, and is not currently as rich as ZKPDL or CACE. However, it offers some advantages over those systems.

First, as Python is an interpreted language, we do not require a custom interpreter for the compiled proofs, as ZKPDL does. Instead, we exploit Python's ability to dynamically generate and execute code at runtime. We employ this feature to convert Camenisch-Stadler proof statements into Charm code, which we feed directly to the interpreter and protocol engine.[4] Second, since our compiler has access to the public and secret[5] variables at compile time, Charm can use introspection to determine the variable types, settings and parameter sizes. This information forms the bulk of what is provided in a ZKPDL or CACE PSL program. Thus, from a developer's perspective, executing a ZK proof is nearly as simple as writing a Camenisch-Stadler statement. We provide an example in Section 3.7.

3 Implementation

In this section, we describe our Python implementation and provide more detail on certain components of our architecture. In Section 3.5 below, we reference an example comparing a protocol description from the literature to one implemented in our system. The code fragment shown in Figure 4 is a good overall

[4] In practice, we first compile to bytecode, then execute. This reduces overhead for proofs that will be conducted multiple times.

[5] Clearly the verifier does *not* have access to the secret variables. We address this in Section 3.7.

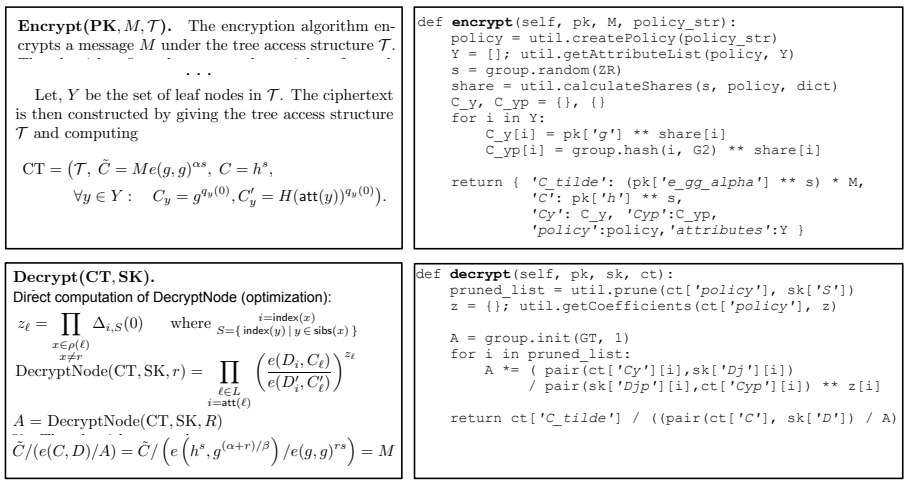


Figure 4: Encryption and decryption in the Bethencourt, Sahai, Waters ABE scheme [12]. The Charm toolbox provides several utility routines that are shared by different ABE schemes.

example of using Charm and is worth skimming at this point to understand our approach. After reading the remainder of the implementation section, our code should be easier to understand.

3.1 Language Features

Python provides many useful features that simplify development for programmers using Charm. Benefits include support for object-oriented programming, dynamic typing, overloading of mathematical operators, and automatic memory allocation and garbage collection.

The language also provides useful built-in data structures such as tuples and dictionaries (essentially, key-value stores) useful for common tasks such as storing ciphertexts and public keys. These values can be automatically serialized and deserialized, eliminating the need for custom parsing code. To read legacy files with a specific binary format we use the python `struct` module, which performs packing and unpacking of binary data. Our decision to use Python is supported by the fact that much of the effort in a typical C implementation relates to defining and serializing data structures.

Python also supports dynamic generation of code. This feature is particularly useful in constructing a Zero-Knowledge proof compiler (see Section 3.7) The features discussed here are not unique to Python and can be found in other high-level languages.[6] However Python has a large and devoted user base and provides a good balance between usability, stability and performance.[7]

3.2 Low-level Python/C Modules

As discussed in Section 2, for performance reasons, our implementation of Charm depends on a few open-source C math libraries including OpenSSL [52], GMP [30] and the Stanford Pairing-Based Crypto library [43]. We provide Python/C extensions for these libraries.

Our base modules expose arithmetic operations using standard mathematical operators such as $*$, $+$ and $**$ (exponentiation).[8] Besides group operations, our base modules also perform essential functions such as element serialization and encoding.

[6] Nor are we the first to import cryptographic operations into Python. See for example [25].

[7] It is also well-supported. Our experiments show that there have been significant performance improvements between Python 2.x and 3.x. For this reason Charm supports the more recent version.

[8] For consistency, group operations are always specified in multiplicative notation, thus $*$ is used for EC point addition and $**$ for point multiplication. This makes it easy to switch between group settings.

In addition to the base modules just described, we provide a cryptobase module that includes fast routines for bitstring manipulation, evaluation of block ciphers, MACs and hash functions. Supported ciphers include AES, DES and 3DES. Moreover, this module implements several standard modes of operation (drawn from PyCrypto [42]) that facilitate encryption of arbitrary amounts of data.

3.3 Benchmark Module

As described in Section 2.1, we provide a benchmark module for measuring computation time and counting operations, such as exponentiation and multiplication, in a given snippet of code at runtime. Our benchmark module provides a consistent interface that developers can use to perform these measurements. Each base module inherits the *benchmark* interface and is incorporated into a cryptographic scheme as follows:

```
bID = InitBenchmark()
# select benchmark options
StartBenchmark(bID, [RealTime, Exp, Mul, Add, Sub])
... code ...
EndBenchmark(bID)
# obtain results
print(GetBenchmark(bID))
```

As stated earlier, benchmarking can be easily removed or disabled after measurements are complete and introduces negligible overhead.

3.4 Algebraic Groups and Fields

While our base modules provide low-level numerical functions, there are still differences in how each module handles serializing elements, encoding messages, and generating group parameters. For instance, for the ecmath module we employ subgroups of elliptic curves over a finite field, whereas the integermath implements integer groups, rings and fields. To reconcile these differences, we provide a thin Python interface to encapsulate differences in group/field parameter generation, serialization, message encoding, and hashing. This interface allows us to standardize calls to the underlying base modules from a developer's perspective.

With this approach, cryptographers are able to adjust the algebraic setting (standard EC, integer or pairing groups) on the fly without having to re-implement the scheme. For instance, our implementations of DSA, El Gamal and Cramer-Shoup [46, 28, 26] can be instantiated in any group with an appropriate structure.

3.5 Schemes

To demonstrate the potential of our framework, we implemented a number of standard and experimental cryptosystems. For space reasons we provide only a partial listing of these in Figure 5, however the full collection can be obtained from [1]. The complete list includes a variety of encryption schemes, signatures, commitments and several interactive protocols. Most of the implementations required fewer than 100 lines of code.

We provide several examples to illustrate code in Charm. Figure 4 shows the encryption and decryption algorithms for the Bethencourt, Sahai and Waters [12] CP-ABE scheme along with the corresponding Charm code. We provide the remaining algorithms, along with some additional examples, in Appendix A. We note that our framework was designed to minimize the differences between published algorithms and code, in the hope of lowering the barriers to implementation.

3.6 Protocol Engine

Every protocol implementation in Charm is a subclass of the `Protocol` base class. This interface provides all of the core protocol functionality, including functions to support protocol implementations, a database for maintaining state, serialization, network I/O, and a state machine for driving the protocol progression.

Scheme	Type	Setting	Comp. Model	Lines
Encryption				
RSA-OAEP [9]	Public-Key Encryption	Integer	ROM	46
CS98 [26]	Public-Key Encryption	EC/Integer	Standard	50
CHK04 [24]	Public-Key Encryption	-	Standard	27
ElGamal [13]	Public-Key Encryption	EC/Integer	Standard	48
Paillier99 [51]	Public-Key Encryption	Integer	Standard	47
BF01 [16]	Identity-Based Encryption	Pairing	ROM	58
BB04 [14]	Identity-Based Encryption	Pairing	Standard	45
Waters05 [53]	Identity-Based Encryption	Pairing	Standard	77
SW05 [48]	Fuzzy Identity-Based Encryption	Pairing	Standard	79
BSW07 [12]	Attribute-Based Encryption	Pairing	ROM*	63
LSW09 [40]	Attribute-Based Encryption	Pairing	ROM*	72
Waters08 [54]	Attribute-Based Encryption	Pairing	ROM*	67
LW10 [41]	MA Attribute-Based Encryption	Pairing	ROM*	84
Digital Signatures				
Schnorr [18]	Signature	Integer	Standard	27
IBE-to-Signature [16]	Signature	-	Standard	27
RSA-PSS [10]	Signature	Integer	ROM	53
DSA [46]	Signature	EC/Integer	*n/a*	34
BLS03 [17]	Short signature	Pairing	ROM	25
HW09 [33]	Signature	Integer	Standard	71
BBS04 [15]	Group signature	Pairing	ROM	47
Miscellaneous				
GS07 [31]	Commitment	Pairing	Standard	41
Pedersen [47]	Commitment	EC/Integer	Standard	17
AdM05 [6]	Chameleon Hash	Integer	ROM	33
RSA HW09 [33]	Chameleon Hash	Integer	Standard	44
Protocols				
Schnorr91 [49]	Zero-Knowledge proof	EC/Integer	Standard	54
ECMQV [38]	Key Agreement	EC	ROM	95
CNS07 [21]	Oblivious Transfer	Pairing	Standard	160

Figure 5: A selected (due to space considerations) listing of some of the cryptographic schemes we implemented. "Code Lines" indicates the number of lines of Python code used to implement the scheme (excluding comments and whitespace), and does not include the framework itself. ROM indicates that a scheme is secure in the Random Oracle Model. A "-" indicates a generic transform (adapter). * indicates a choice made for efficiency reasons.

Creating a new interactive protocol is straightforward. The implementation provides a description of the parties, protocol states and transitions (including error transitions for caught exceptions), as well as the core functionality for each state. State functions accept and return Python dictionaries containing the passed parameters — socket I/O and data serialization is handled transparently before and after each state function runs. Developers have the option to implement their own serialization functionality for protocols with a custom message format. Public parameters may either be passed into the protocol or defined in the `init` function. Finally, we provide templates for some common protocol types (such as Σ-protocols). Figure 6 contains an example of a machine-generated `Protocol` subclass.

Executing protocols and subprotocols. From an application's perspective, executing a protocol consists of two calls to the `Protocol` interface. First, the application calls `Setup()` to configure the protocol with an identifier of one of the parties in the protocol, optional initial state, public parameters, a list of remote parties, and a collection of open sockets. It then calls `Execute()` to initiate communication.

We also provide support for the execution of *subprotocols*. Launching a subprotocol is simpler than an

initial execution, since the protocol engine already has information on the remote parties. The caller simply identifies for the server the role played by each of the parties in the subprotocol (*e.g.,* the Server party may be remapped to be the Prover for the subprotocol), and instructs the protocol engine to run the subprotocol via the `Execute()` method.

Our engine currently supports only synchronous operation. Asynchronous protocol runs must be handled by the application itself using Python's threading capabilities. Callback functions may be supplied by passing function pointers as part of the public parameters. We plan to provide more complete support for asynchronous execution in future releases.

3.7 ZK compiler

Zero-knowledge proofs allow a Prover to demonstrate knowledge of a secret without revealing it to a Verifier. Such proofs are common in privacy-preserving protocols such as the idemix anonymous credential system and Direct Anonymous Attestation [23, 19]. These proofs may be interactive or non-interactive (via the Fiat-Shamir heuristic, or using new bilinear-map based techniques [29, 31]). Regardless of the underlying mechanism, it has become common in the literature to describe such proofs using the notation of Camenisch and Stadler [22]. For instance,

$$\mathsf{ZKPoK}\{(x, y) : h = g^x \ \wedge \ j = g^y\}$$

denotes a proof of knowledge of two integers x, y that satisfy both $h = g^x$ and $j = g^y$. All values not enclosed in parentheses are assumed known to the verifier.

Converting these statements into working protocols is challenging, even for expert developers. To assist implementation, Charm borrows from the techniques of ZKPDL and CACE [45, 5], providing native support for honest verifier Schnorr-type proofs via an automated protocol compiler.

Our compiler, implemented in Python itself, outputs Python code. The interface to the compiler closely resembles a Camenisch-Stadler proof statement. The caller provides two Python dictionaries containing the public and secret parameters, as well as a string describing the proof goal. In some cases, such as when configuring the Verifier portion of an interactive proof, the secret values are not available. We currently deal with this by providing "dummy" variables of the appropriate type. Our runtime compiler can examine the variables and automatically generate appropriate code on the fly. The compiler produces one of two possible outputs: a routine for computing a non-interactive protocol via the Fiat-Shamir heuristic, or a subclass of `Protocol` describing the Prover and Verifier interactions, in the case of interactive protocols.

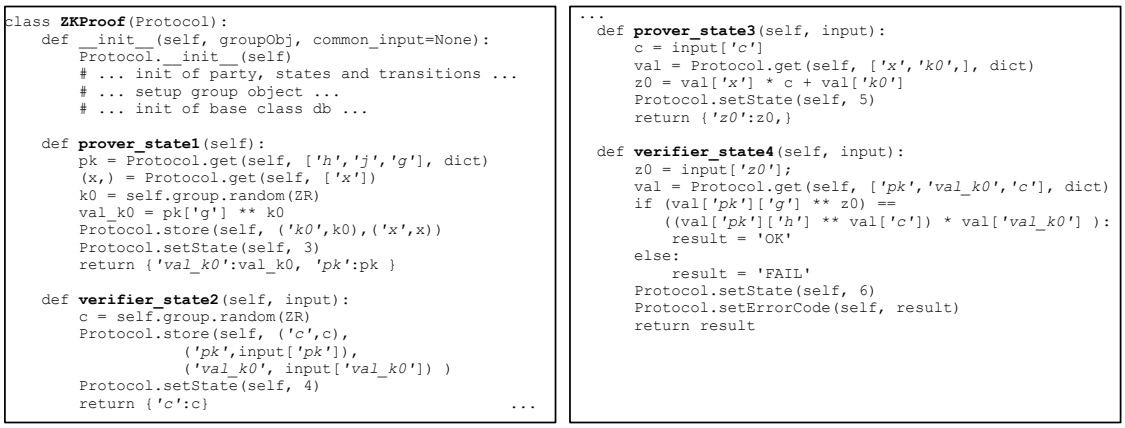
In the interactive case, we provide support routines to generate the class definition, compile the generated code into Python bytecode, initialize communication with sockets provided by the caller, and execute the proof of knowledge. The code below illustrates a typical interactive proof execution from the Prover:

```
# prover
pub = {'h':g ** x, 'g':g, 'j':g ** y}
sec = {'x':x, 'y':y}
result = executeIntZKProof(pub, sec, "(h = g^x) and (j = g^y)", party_info)
```

Figure 6 shows a generated `Protocol` subclass for the proof goal $h = g^x$.

We believe that the runtime technique will be extremely useful for developers who require compact, readable code. However, we note that since our protocol produces Python code, it can also be used to compile static protocol code which may be added to a project. We provide some performance numbers on the compilation process in Section 4.2.

At present our compiler is intended as a proof of concept because it lacks support for many types of statement (e.g. Boolean-OR) and proof settings. Our compiler is less sophisticated than CACE and ZKPDL. For example, in addition to supporting more complex conjunctions and statement types, CACE includes formal verification of proofs. We believe that our approach is complementary to these projects, and we hope to establish collaborations to extend Charm's capabilities in future versions.



```
class ZKProof(Protocol):
    def __init__(self, groupObj, common_input=None):
        Protocol.__init__(self)
        # ... init of party, states and transitions ...
        # ... setup group object ...
        # ... init of base class db ...

    def prover_state1(self):
        pk = Protocol.get(self, ['h','j','g'], dict)
        (x,) = Protocol.get(self, ['x'])
        k0 = self.group.random(ZR)
        val_k0 = pk['g'] ** k0
        Protocol.store(self, ('k0',k0),('x',x))
        Protocol.setState(self, 3)
        return {'val_k0':val_k0, 'pk':pk }

    def verifier_state2(self, input):
        c = self.group.random(ZR)
        Protocol.store(self, ('c',c),
                    ('pk',input['pk']),
                    ('val_k0', input['val_k0']) )
        Protocol.setState(self, 4)
        return {'c':c}                                  ...
```

```
...
 def prover_state3(self, input):
     c = input['c']
     val = Protocol.get(self, ['x','k0',], dict)
     z0 = val['x'] * c + val['k0']
     Protocol.setState(self, 5)
     return {'z0':z0,}

 def verifier_state4(self, input):
     z0 = input['z0'];
     val = Protocol.get(self, ['pk','val_k0','c'], dict)
     if (val['pk']['g'] ** z0) ==
        ((val['pk']['h'] ** val['c']) * val['val_k0'] ):
         result = 'OK'
     else:
         result = 'FAIL'
     Protocol.setState(self, 6)
     Protocol.setErrorCode(self, result)
     return result
```

Figure 6: A partial listing of the generated protocol produced by our Zero-Knowledge compiler for the honest-verifier proof $\mathsf{ZKPoK}\{(x) : h = g^x\}$.

3.8 Meta-information and Adapters

Charm provides the ability to label schemes so that they carry meta-information about their input/output space and security definitions. Optionally, developers can provide other details such as the complexity assumption and computational models used in the scheme's security proof. This information allows developers to compare and check compatibility between schemes.

All schemes descend from the `Scheme` class, which provides tools to record and evaluate meta-information. Developers use the `setProperty()` method to specify important properties. For example, the `init` function of an Identity-Based Encryption scheme might include a call of this form:

```
# Set the scheme's security definition, ID space,
# and message space.
setProperty(self, secdef=IND_ID_CPA, id=str, msg=str)
```

Schemes with more restrictive parameters, *e.g.,* group elements and/or strings of limited length, can specify these requirements as well. In some cases, evaluation of a scheme depends on the scheme's public key. Once each scheme is labeled with the appropriate metadata when defined, we can programmatically extract this information at run-time to verify a given set of criteria.

Adapter example. To illustrate how this functionality works in practice, we consider the process of constructing *adapters* between different schemes. In Section 2 we proposed an adapter chain to convert the Boneh-Boyen IND-sID-CPA-secure signature scheme [14] into an EU-CMA signature (see Figure 3). This transformation requires two adapters: one to convert the selectively-secure IBE scheme into an adaptively-secure IBE scheme (in the random oracle model), and another to transform the resulting IBE into a signature using the technique of Naor [16]. Let us now describe the core functionality of these adapters.

The Hash Identity adapter has an explicit and implicit function. Explicitly, it applies a hash function to the Boneh-Boyen IBE, which accepts identities in the group $\mathbb{Z}_r$,[9] thus altering the identity-space to $\{0,1\}^*$. Implicitly, it converts the security definition of the resulting IBE scheme from IND-sID-CPA to the stronger IND-ID-CPA definition and updates the meta-information to note that the security analysis is in the random oracle model.[10] The adapter itself is implemented as a subclass of `IBEnc` (see Figure 12 in Appendix A). It accepts the Boneh-Boyen IBE (also an `IBEnc` class) as input to its constructor. At construction time, the

[9]The value r is typically a large prime.

[10]On a call to `encrypt` or `keygen` the adapter simply hashes an arbitrary string into an element of $\mathbb{Z}_r$, then passes the result to the underlying IBE scheme. This technique and its security implications are described in [14].

adapter must verify the properties of the given scheme using the `checkProperty()` call. It then advertises its own identity space and security information. This code is contained within the adapter's `init` routine and appears as follows:

```
...
if IBEnc.checkProperty(scheme, {'scheme':IBEnc, 'secdef':IND_sID_CPA, 'id':ZR}):
   self.ibenc = scheme
   IBEnc.setProperty(self, secdef=IND_ID_CPA, id=str, secModel=ROM)
   ...
```

The IBE-to-Sig adapter converts any adaptively-secure IBE scheme into an EU-CMA signature.[11] This adapter is implemented as a subclass of `PKSig`. It accepts an object derived from IBEnc and verifies that it advertises at least IND-ID-CPA security (IND-sID-CPA is not sufficient, hence our use of the previous adapter) and possesses an appropriate message space. With this check satisfied, this adapter inherits the security model of the underlying IBE, adopts the IBE's identity space as the message space for the signature, and advertises the EU-CMA security definition.

In future versions of the library, we hope to significantly extend the usefulness of this meta-data, to include detailed information on performance (gathered through automatic testing). We also intend to provide tools for *automatically* constructing useful adapter chains based on specific requirements. At present adapters must be manually configured by the application.

3.9 Type checking and conversion

Python programs are dynamically typed. In general, we believe that this is a benefit for a rapid prototyping system: dynamic typing makes it possible to assemble and modify complex data structures (*e.g.,* ciphertexts) "on the fly" without the need for detailed structure definitions.

Of course, the lack of static typing has disadvantages. For one thing, type errors may not be detected until runtime. Furthermore, it can limit the utility of adapters that depend on having *a priori* knowledge about a scheme's input or output characteristics.

To address these issues Charm provides optional support for static typing using the Python annotation interface. When it is provided, Charm uses this type information to validate the inputs provided to a cryptographic algorithm and, in cases where the inputs are of the wrong type, to automatically convert them. For the latter purpose, Charm provides a standard library designed to encode values to and from a variety of standard types, including bit strings and various types of group element. An example of the Charm typing syntax is provided below:

```
pk_t = { 'g1' : G, 'g2' : G, 'c' : G, 'd' : G, 'h' : G }
c_t = { 'u1' : G, 'u2' : G, 'e' : G, 'v' : G }
def encrypt(self, pk : pk_t, M : str) -> c_t: ...
```

We believe that support for explicit typing also provides a foundation for adding formal verification techniques to Charm, though we leave such verification to future work.

4 Performance

Charm is primarily intended for rapid prototyping, with an emphasis on compactness of source code and similarity between standard protocol notation and code. These properties all favor the developer and are qualities designed to produce more correct, robust and secure code. We recognize that our approach to achieving these properties is likely to involve some tradeoff in performance.

This section describes some representative performance measurements that we performed using Charm's built-in benchmarking system. For comparison purposes we also conducted experiments on two existing C

[11] Naor [16] observed that adaptively-secure IBE can be converted into a signature scheme by using the IBE key extraction algorithm for signing.

implementations. We observe that the performance cost of using Charm is variable, and depends significantly on the nature of the scheme being implemented. Finally, we conducted some experiments to demonstrate the capabilities of our benchmark module, and to examine the performance of our ZK proof compiler.

4.1 Comparison with C Implementations

We conducted detailed timing experiments on two of the cryptosystems we implemented: ECDSA and a CP-ABE scheme due to Bethencourt, Sahai, Waters [12]. We chose these because of available C implementations that we could compare against. Our experiments comprise two different points on a spectrum: our ECDSA experiment considers Charm's performance in an algorithm with very fast operation times, and our CP-ABE experiment considered a scheme with a high computational burden (to stress this, we instantiated the scheme with a 50-element policy).

Experimental setup. We used the benchmark module to collect timings for our Charm implementation of the ECDSA Sign and Verify algorithms. This provided us with total operation time for both algorithms. We then collected total operation times for OpenSSL's implementation of the same algorithms using the built-in `speed` command.

For CP-ABE we again used benchmark to collect measurements for our ABE key generation, encryption and decryption implementations (omitting the setup routine). For key generation, we extracted a key containing 50 attributes $(1, \ldots, 50)$. We next encrypted a random message (in the group $\mathbb{G}_T$) under a policy consisting solely of AND gates: (1 AND 2 AND ... AND 50). Finally, we decrypted the message using the extracted key. For each experiment we measured total time and number of numerical operations. We repeated these experiment using John Bethencourt's library (available from [2]).

We conducted our experiments on a Macbook Pro with an 2.4Ghz Intel i5 with 4GB of RAM running Mac OS 10.6. All of our experiments were performed on a single core of the processor. For all experiments (Charm and C) we used either OpenSSL v1.0.0d library or libpbc 0.5.11 to perform the underlying mathematical operations. Our ECDSA experiments used the standard NIST P-192 elliptic curve. For CP-ABE we used a 512-bit supersingular curve (with embedding degree $k = 2$) from libpbc. All of our timing results are the average of ten experimental runs.

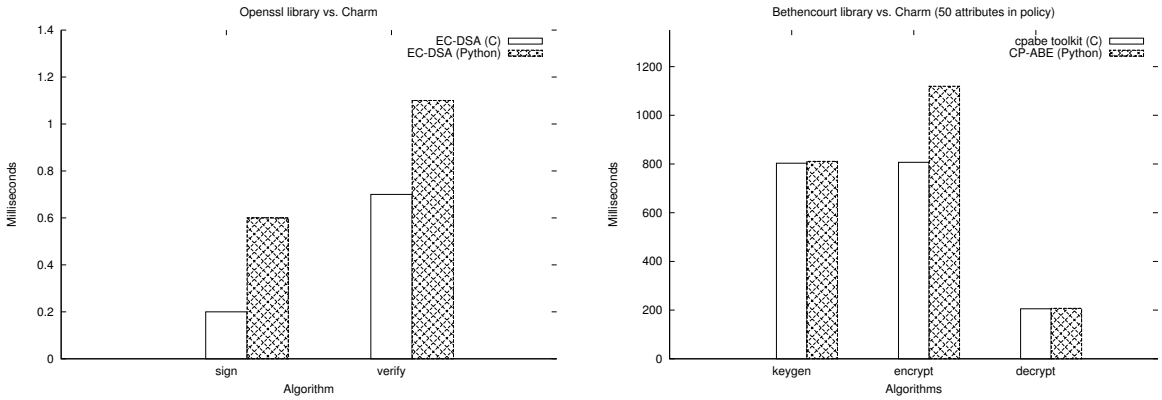
Commentary. The results of our experiments are presented in Figure 7. Unsurprisingly, our Charm implementation of ECDSA suffered a substantial performance penalty when compared to the OpenSSL version. This is unavoidable given the relatively low overall time required for ECDSA operations — even small interpretation inefficiencies add up to a large percentage of the total cost. However, some of this extra time can be attributed to inefficiencies in our module implementation, which should be optimized to reduce the amount of C-Python interaction. Even with this overhead, the overall performance is still acceptable for many applications.

Our results with CP-ABE (and 50 attributes) are encouraging. For key generation and decryption, Charm is competitive with the C implementation. For encryption we noticed a significant additional overhead in the Charm implementation. We believe that this is due to an inefficient policy parser, which we intend to replace in future versions.

4.2 Additional Performance Measurements

We also conducted timing experiments on two additional schemes for which we did not have C implementations. The schemes we tested include the Cramer-Shoup scheme [26] (in Integer groups with 1024-bit keys), and a more recent CP-ABE scheme (with 50 leaves in policy tree) due to Waters [54].

Finally, to provide some insight into the performance impact of our ZK proof compiler we compiled a protocol for the proof $\mathsf{ZKPoK}\{(x, y) : h = g^x \text{ and } j = g^y\}$, and measured the time for our compiler, as well as the time required by Python to compile the resulting protocol to bytecode. We present the results of our benchmarks in Figure 8.



(a) Comparison to OpenSSL

(b) Comparison to Beth-cpabe toolkit

Figure 7: For EC-DSA, we select the NIST P-192 elliptic curve and for CP-ABE [12], we measure 50 attributes for keygen and 50 leaves in the policy tree for encrypt and decrypt.

Scheme	Time	Pairs	Exps	Muls	w/o
CS98-Enc	9.3ms	·	5	3	8.4ms
CS98-Dec	7.1ms	·	3	3	6.4ms
Waters08-Enc	1532ms	·	152	345	1531ms
Waters08-Dec	415ms	101	100	296	414ms

Protocols	Gen.	Comp.	Exec.
ZK Compiler	2.6ms	3.5ms	22.2ms

Figure 8: At top, benchmark results for Cramer-Shoup and the Waters09 CP-ABE. From left, results include total time, number of pairings, exponentiations and multiplications, and time with benchmark module deactivated. At bottom, time required to generate, compile, and execute (excluding network communications) the proof $\mathsf{ZKPoK}\{(x, y) : h = g^x \ \wedge \ j = g^y\}$.

4.3 Application Example

To demonstrate the usefulness of Charm in more complex applications, we integrated Charm into TLS Lite [3]. TLS Lite is an efficient Python-based implementation of the TLS protocol, complete with support for a variety of ciphersuites. We first modified the standard TLS Lite handshake to replace its existing (PyCrypto) RSA encryption and signature verification routines with the Charm equivalents.[12] We also prepared another version that swaps both schemes with standard-model alternatives: Cramer-Shoup [26] and the Waters signature scheme [53].[13]

We first measured the standard handshake using the original (unmodified) version of TLS Lite with client certificates enabled, using self-signed certificates and in all cases averaging over 10 experimental runs. Within each run we recorded the total encryption and decryption time on server and client, then summed these. We repeated the same process to obtain a total time for signing and verification. We then measured our modified TLS Lite, which is identical to the standard version but configured to use Charm for RSA encryption and signing. Finally, we measured our version with Cramer-Shoup and Waters signatures (using appropriately generated certificates).[14] Figure 9 presents the results of our experiments.

[12]Because Charm does not support the older RSA-PKCS #1v1.5 encryption and signature padding schemes used by TLS Lite, we replaced them with RSA-OAEP and RSA-PSS respectively.

[13]Note that this simple substitution does not produce a "standard model-secure" version of TLS along the lines of [35] — such a modification would require substantially greater changes to the TLS handshake. Rather, this informal experiment was intended to demonstrate the usefulness of Charm in experimenting with different cryptosystems.

[14]We implemented Cramer-Shoup in the NIST p192 elliptic curve, and the Waters signature in a 224-bit MNT elliptic curve

Version	Total Enc/Dec	Total Sig/Ver
Unmodified TLS Lite (RSA/RSA)	4.15ms	4.05ms
Charm TLS Lite (RSA/RSA)	3.8ms	3.62ms
Charm TLS Lite (CS98/Waters05)	6.0ms	17.0ms

Figure 9: Timing results for our modified versions of TLS Lite. Total time measurements represent the sum of time measurements on the server and client.

5 Related Work

Our work builds upon previous efforts to provide software libraries for people wishing to develop systems that use cryptography. We describe four different types of libraries below.

Cryptographic (primitive) libraries

The first widely available general purpose library for cryptographic functions was Jack Lacey's CryptoLib [36]. This software package provided programmers with an API to call many of the commonly used cryptographic functions in 1993, at the time when CryptoLib was released. The package included a big arithmetic function for performing operations on integers larger than can be represented using standard programming languages. It also included Chinese Remainder Theorem speedup of exponentiation, and many other efficiently implemented primitives for such functions as DES, RSA, El Gamal, and MD5. The package was designed to be portable and cross platform with some sacrifices made to efficiency for the sake of portability. By linking to CryptoLib, people interested in implementing cryptographic functions could focus on their protocols and to not worry about the low level primitives. The package was supported and improved for several years by its author.

Following CryptoLib, many other packages were developed by different people, including Peter Guttman's similarly named CryptLib (`http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`), RSA's Bsafe Crypto-C (`http://www.rsa.com/rsalabs/node.asp?id=2301`), and more recently JAVA libraries such as Cryptix (`http://www.cryptix.org/`) and BouncyCastle (`http://www.bouncycastle.org/`). There are too many API's available today for standard cryptographic functions to list them all here.

There have not been as many implementations of ABE and its related primitives. Of note is the implementation by Bethencourt, Sahai and Waters [12], which provides an API for ciphertext policy ABE. This package is part of the Advanced Crypto Software collection (`http://acsc.cs.utexas.edu/`) at the university of Texas [2], which in addition to the ABE code, includes packages for the Paillier public key cryptosystem, forward-secure signatures, pairing-based cryptography (implemented in C), and several other application-based primitives.

Math libraries

The Gnu Multiple Precision Arithmetic Library (GMP) [30] is a free, high-precision mathematics library, specifically optimized for speed of cryptographic algorithms. It was initially released in 1991 and, new improved versions have been released on a yearly basis. Pairing-Based Cryptography library (PBC) [43] is a free library written in C and built on top of GMP. PBC was built for speed and portability with the goal of enabling implementation of pairing-based cryptosystems.

The Multiprecision Integer and Rational Arithmetic Library (MIRACL) [50] is free and is written in C and C++. It provides an API for big number arithmetic needed to implement cryptographic operations. It supports elliptic curve cryptography, AES, SHA-160/256/384/512. In addition to the API, MIRACL exposes some of the lower level functions with the aim of allowing programmers to develop new cryptographic functions from low level primitives.

using SHA256 is the hash function. A wrinkle in using the Cramer-Shoup scheme is that the plaintext space of Cramer-Shoup is dramatically smaller than that of RSA-1024, hence we used a shorter pre-master secret.

Cryptographic compilers and frameworks

Ben Laurie's *Stupid* programming language [37] compiles into C and Haskell and is intended for things like ciphers and hash functions. Cryptol [39] compiles to a VHDL circuit for use with an FPGA. Yehuda Lindell *et. al.* at Bar Ilan University is building a framework for rapid prototyping of cryptographic primitives.[15] At the time of this writing, they have not yet released nor published about their system.

Protocol and Secure Function Evaluation compilers

The authors of the Zero Knowledge Proof Descriptive Language (ZKPDL) [45] offer a language and an interpreter for developers who wish to implement privacy-preserving protocols. Their example application is electronic cash, but their descriptive language is more general. A similar approach is provided by FairPlay [44], which provides a language-based system for secure multi-party computations. The authors of FairPlay provide a Secure Function Definition Language (SFDL), which can be used by programmers to specify code for multi-party computations.

The Computer Aided Cryptography Engineering (CACE) project has also developed a system that specifies a language for zero knowledge proofs [8, 7]. In this system, a compiler translates zero knowledge protocol specifications into JAVA code or Latex statements that can be incorporated into a research paper. In a similar vein, a software package called Tool for Automating Secure Two-Party Computations (TASTY) [32] allows protocol designers to specify a high-level description of a computation that is to be performed on encrypted data. TASTY then generates protocols based on the specification, and compares the efficiency of different protocols.

6 Conclusion and Future Work

This paper describes a *programmer*-focused software development methodology for cryptographic systems. We designed and built the Charm framework to reduce the load on the cryptographer. Low-level mathematical code, often a performance bottleneck, is written in C, and is called from the high level Python code. Developers build their protocols in Python and enjoy benefits of the built in features of that high level language, as well as the framework Toolbox and other mechanisms provided by Charm.

Charm contains a protocol engine that takes care of the communications, serialization and other housekeeping that is integral to implementing a multi-party protocol. Thus, developers are shielded from the minutia that is not relevant to the cryptographic theory in their protocol. We show in detail how a cryptographer can build a system implementing a ZK proof system based on a specification in the most commonly found academic notation.

Charm is extensible, and we are continuing to add schemes. We believe that the framework is simple enough to understand that we hope to develop an active user group. It is our aim to encourage others to develop schemes and to contribute them to the framework, and we will support a user community, if we are successful. In the coming months we plan to expand the capabilities of our library, and to optimize the existing code which should produce significant performance benefits while reducing dependencies.

This work leaves us with a number of open problems. We plan to investigate mechanisms for *automatically* discovering and configuring adapter chains according to constraints provided by the application. We also intend to enhance our zero knowledge proof compiler to support more complex proof statements, as well as bilinear Groth-Sahai proofs [31] (which to our knowledge have not been implemented in an open library). Additionally, we believe that there may be other applications of dynamic code generation in a cryptographic framework, including inline compilation of Secure Multiparty Computation circuits (as in [44, 32]).

Finally, we accept that there may be instances where development requirements cannot support Python. To address this we plan to examine the possibility of compiling Charm code directly to C, using tools such as Shedskin [27].

[15] personal communications

References

[1] Charm. URL removed for submission.

[2] The Advanced Crypto Software Collection. `http://acsc.cs.utexas.edu/`.

[3] TLS Lite. Available from `http://trevp.net/tlslite/`.

[4] Pygame. Available from `http://www.pygame.org/wiki/index.html`, July 2011.

[5] Almeida, J. B., Bangerter, E., Barbosa, M., Krenn, S., Sadeghi, A.-R., and Schneider, T. A certifying compiler for zero-knowledge proofs of knowledge based on Σ-protocols. In *Proceedings of the 15th European conference on Research in computer security* (Berlin, Heidelberg, 2010), ESORICS'10, Springer-Verlag, pp. 151–167.

[6] Ateniese, G., and de Medeiros, B. On the key exposure problem in chameleon hashes. In *SCN '04* (2004), vol. 3352 of LNCS, Springer, pp. 165–179.

[7] Bangerter, E., Barzan, S., Sadeghi, A., Schneider, T., and Tsay, J. Bringing zero-knowledge proofs of knowledge to practice. *17th International Workshop on Security Protocols* (2009).

[8] Bangerter, E., Camenisch, J., Krenn, S., Sadeghi, A.-R., and Schneider, T. Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471, 2008. `http://eprint.iacr.org/`.

[9] Bellare, M., and Rogaway, P. Optimal asymmetric encryption padding — how to encrypt with rsa. In *EUROCRYPT '94* (1994), pp. 92–111.

[10] Bellare, M., and Rogaway, P. The exact security of digital signatures: How to sign with RSA and Rabin. In *EUROCRYPT '96* (1996), U. Maurer, Ed., vol. 1070 of LNCS, Springer-Verlag.

[11] Bethencourt, J. Libpaillier, July 2006.

[12] Bethencourt, J., Sahai, A., and Waters, B. Ciphertext-policy Attribute-Based Encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007), IEEE Computer Society, pp. 321–334.

[13] Blakley, G., Chaum, D., and ElGamal, T. *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, vol. 196. Springer Berlin / Heidelberg, 1985, pp. 10–18.

[14] Boneh, D., and Boyen, X. Efficient selective-ID secure Identity-Based Encryption without random oracles. In *EUROCRYPT '04* (2004), vol. 3027 of LNCS, pp. 223–238.

[15] Boneh, D., Boyen, X., and Shacham, H. Short group signatures. In *CRYPTO '04* (2004), vol. 3152 of LNCS, pp. 45–55.

[16] Boneh, D., and Franklin, M. K. Identity-based encryption from the Weil Pairing. In *CRYPTO '01* (2001), vol. 2139 of LNCS, pp. 213–229.

[17] Boneh, D., Lynn, B., and Shacham, H. Short signatures from the weil pairing. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology* (London, UK, 2001), ASIACRYPT '01, Springer-Verlag, pp. 514–532.

[18] Brassard, G., and Schnorr, C. *Efficient Identification and Signatures for Smart Cards*, vol. 435. Springer Berlin / Heidelberg, 1990, pp. 239–252.

[19] Brickell, E., Camenisch, J., and Chen, L. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 132–145.

[20] Camenisch, J., and Lysyanskaya, A. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *EUROCRYPT '01* (2001), vol. 2045 of LNCS, Springer, pp. 93–118.

[21] Camenisch, J., Neven, G., and abhi shelat. Simulatable adaptive oblivious transfer. In *EUROCRYPT '07* (2007), vol. 4515 of LNCS, pp. 573–590.

[22] Camenisch, J., and Stadler, M. Efficient group signature schemes for large groups. In *CRYPTO '97* (1997), vol. 1296 of LNCS, pp. 410–424.

[23] Camenisch, J., and Van Herreweghen, E. Design and implementation of the idemix anonymous credential system. In *Proceedings of the 9th ACM conference on Computer and communications security* (New York, NY, USA, 2002), CCS '02, ACM, pp. 21–30.

[24] Canetti, R., Halevi, S., and Katz, J. Chosen-ciphertext security from Identity Based Encryption. In *EUROCRYPT '04* (2004), vol. 3027 of LNCS, pp. 207–222.

[25] Condra, G. pypbc. Available from `http://www.gitorious.org/pypbc`.

[26] Cramer, R., and Shoup, V. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO '98* (London, UK, 1998), Springer, pp. 13–25.

[27] Dufour, M. Shedskin. Available from `http://code.google.com/p/shedskin`, July 2009.

[28] El Gamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of Crypto '84* (1984), pp. 10–18.

[29] Fiat, A., and Shamir, A. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO '86* (1986), vol. 263 of LNCS, pp. 186–194.

[30] GNU. The GNU Multiple Precision Arithmetic Library. Available from `http://www.gmplib.org`.

[31] Groth, J., and Sahai, A. Efficient non-interactive proof systems for bilinear groups. In *EUROCRYPT '08* (2008), vol. 4965 of LNCS, Springer, pp. 415–432.

[32] Henecka, W., K ögl, S., Sadeghi, A.-R., Schneider, T., and Wehrenberg, I. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 451–462.

[33] Hohenberger, S., and Waters, B. Realizing hash-and-sign signatures under standard assumptions. In *Advances in Cryptology – EUROCRYPT '09* (2009).

[34] II, J. W. World of Warcraft - 10,000,000 Lua users and growing! Available from `http://bluedino.net/luapix/lua-whitehead.pdf`, July 2008.

[35] Jager, T., Kohlar, F., Schäge, S., and Schwenk, J. A standard-model security analysis of tls. Cryptology ePrint Archive, Report 2011/219, 2011. `http://eprint.iacr.org/`.

[36] Lacy, J. B. CryptoLib: Cryptography in software. *USENIX Security Conference IV* (1993), 1–18.

[37] Laurie, B., and Clifford, B. The Stupid programming language. Source code available at `http://code.google.com/p/stupid-crypto/`.

[38] Law, L., Menezes, A., Qu, M., Solinas, J., and Vanstone, S. An efficient protocol for authenticated key agreement. *Designs, Codes and Cryptography 28*, 2 (2003-03-21), 119–134.

[39] Lewis, J. R., and Martin, B. CRYPTOL: High Assurance, Retargetable Crypto Development and Validation. Available from `http://www.galois.com/files/Cryptol_Whitepaper.pdf`, October 2003.

[40] Lewko, A., Sahai, A., and Waters, B. Revocation systems with very small private keys. *Security and Privacy, IEEE Symposium on 0* (2010), 273–285.

[41] Lewko, A., and Waters, B. Decentralizing attribute-based encryption. In *EUROCRYPT '11* (2011), K. G. Patterson, Ed., vol. 6632 of LNCS, Springer, pp. 568–588. `http://eprint.iacr.org/`.

[42] Litzenberger, D. C. PyCrypto - The Python Cryptography Toolkit. Available at `http://www.dlitz.net/software/pycrypto/`.

[43] Lynn, B. The Stanford Pairing Based Crypto Library. Available from `http://crypto.stanford.edu/pbc`.

[44] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay - a secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 287–302.

[45] Meiklejohn, S., Erway, C. C., Küpçü, A., Hinkle, T., and Lysyanskaya, A. ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 13–13.

[46] NIST. Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186, May 1994.

[47] Pedersen, T. P. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO '92* (1992), vol. 576 of LNCS, pp. 129–140.

[48] Sahai, A., and Waters, B. Fuzzy identity-based encryption. In *EUROCRYPT* (2005), pp. 457–473.

[49] Schnorr, C.-P. Efficient signature generation for smart cards. *Journal of Cryptology 4*, 3 (1991), 239–252.

[50] Scott, M. MIRACL library. Indigo Software. http://indigo.ie/∼mscott/#download.

[51] Stern, J., and Paillier, P. *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*, vol. 1592. Springer Berlin / Heidelberg, 1999, pp. 223–238.

[52] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. `www.openssl.org`, April 2010.

[53] Waters, B. Efficient Identity-Based Encryption without random oracles. In *EUROCRYPT '05* (2005), vol. 3494 of LNCS, pp. 114–127.

[54] Waters, B. Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. Cryptology ePrint Archive, Report 2008/290, 2008. `http://eprint.iacr.org/`.

A Code Examples

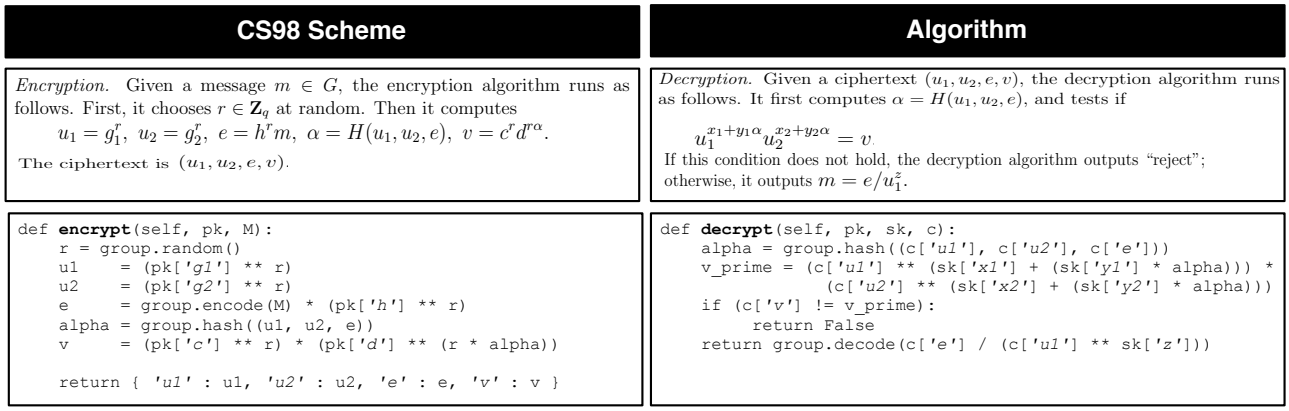


Figure 10: Encryption and Decryption in the Cramer-Shoup scheme [26]. We exclude group parameter initialization and keygen.

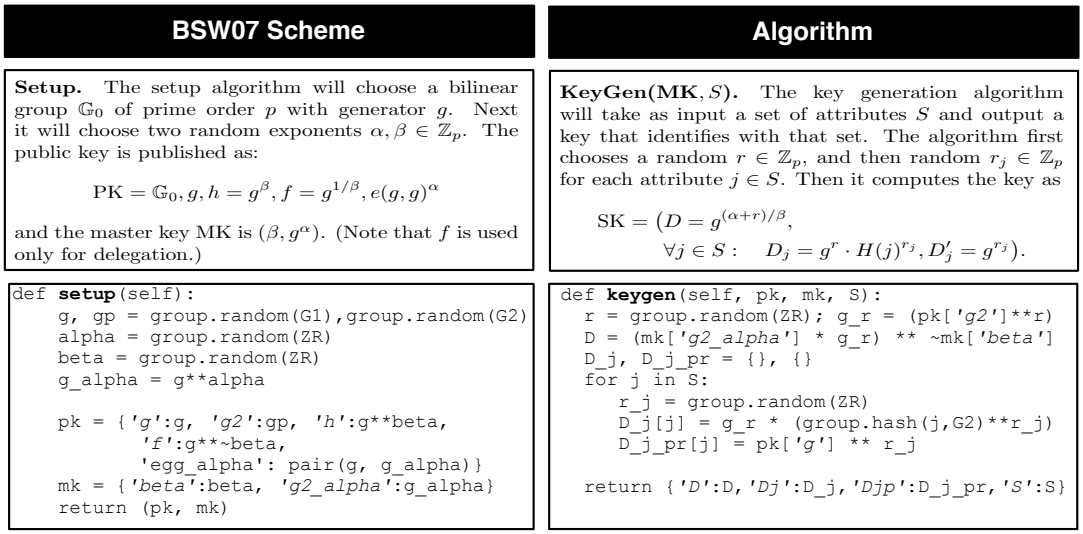


Figure 11: Setup and Keygen in the Bethencourt, Sahai, Waters scheme [12]. We exclude group parameter generation.

IBE-to-Sig Adapter

```
class Sig_Generic_ibetosig_Naor01(PKSig):
    def __init__(self, scheme, groupObj):
        PKSig.__init__(self)
        global ibe, group
        # ... verify scheme properties ...
        ibe = scheme; group = groupObj

    def keygen(self, secparam=None):
        (mpk, msk) = ibe.setup(secparam)
        return (mpk, msk)
```

```
    def sign(self, sk, message):
        return ibe.extract(sk, str(message))

    def verify(self, pk, m, sig):
        if hasattr(ibe, 'verify'):
            result = ibe.verify(pk, sig)
            if result == False: return False
        message = group.random(GT)

        C = ibe.encrypt(pk, sig['id'], message)
        if (ibe.decrypt(sig, C) == message):
            return True
        else:
            return False
```

Figure 12: The entire IBE to signature adapter scheme [16] implemented in Charm.