J Cryptogr Eng (2013) 3:111–128
DOI 10.1007/s13389-013-0057-3

REGULAR PAPER

Charm: a framework for rapidly prototyping cryptosystems

Joseph A. Akinyele · Christina Garman · Ian Miers · Matthew W. Pagano · Michael Rushanan · Matthew Green · Aviel D. Rubin

Received: 17 September 2012 / Accepted: 9 February 2013 / Published online: 7 March 2013

Abstract We describe Charm, an extensible framework for rapidly prototyping cryptographic systems. Charm provides a number of features that explicitly support the development of new protocols, including support for modular composition of cryptographic building blocks, infrastructure for developing interactive protocols, and an extensive library of re-usable code. Our framework also provides a series of specialized tools that enable different cryptosystems to interoperate. We implemented over 40 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 to compare the performance of Charm primitives to existing C implementations. We show that in many cases our techniques result in an order of magnitude decrease in code size, while inducing an acceptable performance impact. Lastly, the Charm framework is freely available to the research community and to date, we have developed a large, active user base.

J. A. Akinyele (✉) · C. Garman · I. Miers · M. W. Pagano · M. Rushanan · M. Green · A. D. Rubin
Department of Computer Science, Johns Hopkins University, 3400 N. Charles St., 21218 Baltimore, MD, USA
e-mail: akinyelj@cs.jhu.edu

C. Garman
e-mail: cgarman@cs.jhu.edu

I. Miers
e-mail: imiers@cs.jhu.edu

M. W. Pagano
e-mail: mpagano@cs.jhu.edu

M. Rushanan
e-mail: micharu1@cs.jhu.edu

M. Green
e-mail: mgreen@cs.jhu.edu

A. D. Rubin
e-mail: rubin@cs.jhu.edu

Keywords Applied cryptography · Protocols · Software · Privacy

1 Introduction

Recent developments in cryptography have the potential to greatly impact real world systems. Advances in lattices and pairings have driven new paradigms for securely processing and protecting sensitive information such as identity-based encryption [17,19,27,57,76] and attribute-based encryption [14,58,69,77], and privacy-preserving schemes such as ring signatures [22,36], group signatures [18,25] and anonymous credentials [29,30]. Without these kind of advances, a number of results in top security conferences would not be possible [15,64,79].

Unfortunately, many potentially useful and novel schemes exist only in research papers and have not actually been implemented. A few of these schemes find their way into isolated C libraries that are maintained purely by their creator, executed only as proof of concept and are operated solely in their own limited domain. While elliptic curves and lattices enabled some of these advances, they also substantially increased the complexity: writing software for cryptosystems no longer involves only number theory and modular arithmetic. This is doubly problematic because the size of typical C implementations makes bugs likely and audits hard. The barrier to usage, consequently, remains very high.

There have been a handful of elegant implementations of a small number of new primitives [13,61,73] as well as some tools for protocol development [5,49,55,56,62,63]. These systems serve their special purposes well, but are not interoperable, and so developers wishing to build a system using multiple primitives must write non-cohesive *glue* code to piece their implementations together.

In practice, libraries such as Sage [71], the Stanford Pairing-Based Crypto (PBC) [61] and MIRACL [70] fulfill an important role of providing implementations of advanced mathematics for algebra, number theory, and elliptic curves just to name a few. While these libraries provide a solid foundation for developing advanced cryptography, they were not designed with *usability* or *interoperability* in mind in terms of composing, structuring, and reusing cryptographic primitives. Although this may seem like an engineering detail, serious theoretical issues can arise from the improper combination of cryptographic primitives. Therefore, great care must be taken to accommodate the theoretical foundations of underlying primitives when designing a system that provides robust, composable, and modular cryptography.

Our contribution We present Charm[1] [4], a new, extensible and unified framework for *rapidly prototyping* experimental cryptographic schemes and leveraging them in system applications. Charm is built around the concepts of extensibility, composability, and modularity. The framework is implemented in Python, a well-supported high-level language, designed to reduce development time and code complexity while promoting component re-use. Computationally-intensive mathematical operations are implemented as native modules, enabling performant schemes and protocols while preserving the advantages of high-level languages for scheme implementations. Although Charm is written in a dynamically typed interpreted language, the concepts and abstractions developed in this paper can be realized in variety of programming languages.

The design goals of Charm are:

Enabling efficient, extensible numeric computation New primitives are invented and existing implementations of primitives are optimized on a regular basis. For example, the PBC library [61], one of the original libraries providing pairings, has been supplanted in terms of performance by alternative libraries such as MIRACL [70] and RELIC [6]. Similarly, lattice-based cryptographic operations are an increasingly desirable feature in scheme development. In practice, the math libraries supporting any given cryptographic operation are subject to change. The challenge is how to enable these changes without disrupting the higher-level scheme.

Supporting succinct cryptographic protocols Although cryptographic protocols only capture the mathematical formulas on paper, in practice network protocols must embed the necessary logic required for message serialization, data transmission, state transitions, error handling, and the execution of subprotocols. Protocols involving zero-knowledge proof statements are particularly problematic: concrete implementations require explicit information not usually present in an algorithmic sketch. The challenge is to provide an interface for wire protocols roughly equivalent to the way the protocols are specified in research papers.

Supporting scheme composition Composing cryptographic algorithms allows for the rapid creation of new schemes, protocols and facilitates code reuse. Not only does this make implementers more efficient, it improves the security of the system by ensuring there is one canonical version of a given scheme or technique. However, composability creates its own set of hurdles: schemes may use different plaintext and ciphertext spaces, security assumptions and security models. The challenge is abstracting away these differences while preserving the schemes' underlying security and functionality.

Providing measurement capability Benchmarking and profiling are particularly important, both from a theoretical perspective and an implementation standpoint for complex schemes (e.g., homomorphic encryption). Simple benchmarking allows quick prototyping and comparison of novel variations of naïve implementations of schemes. Profiling enables in-depth optimization of full-fledged schemes with fine-grained performance data. The difficulty is providing both seamless benchmarking and in-depth profiling while maintaining component modularity.

Allowing application embedding Rapid prototyping and ease of use require that the framework be written in a user-friendly, high-level language. If developers outside of the cryptographic community are to build applications with advanced cryptographic constructs, the choice of language is critical. The dilemma is how to provide a level of abstraction (or embedding API) to outside systems without unduly limiting the expressiveness of the framework.

Allowing cryptographic algorithm agility As noted by Acar et al. [2], cryptographic algorithms have a limited shelf life. For example, once exhaustive search rendered DES keys insecure, DES was replaced by AES. Similarly, MD5 and SHA1 were discovered to contain vulnerabilities [74,75]. A system must be designed such that algorithms can be replaced when necessary [3]. Cipher algorithm replacement must be done without compromising security, without breaking functionality, and if possible, without requiring keys to change.

2 Background

We briefly provide an overview of the concepts discussed in this work. Identity-based encryption (IBE) [17,19], first introduced notionally by Adi Shamir in 1984, is a form of public-key encryption where the public-key is a string. Attribute-based encryption (ABE) [14,58,77] (introduced by Sahai and Waters [69]) is a generalization of IBE where the public-key is a set of attributes. Users can only decrypt if the attributes associated with their private key matches certain attributes specified in the ciphertext.

[1] Project webpage: http://charm-crypto.com.

Practical implementations of these recent forms of encryption typically involve the use of bilinear maps (also called pairings). A pairing is an efficient mapping $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ over two multiplicative cyclic groups $\mathbb{G}$ and $\mathbb{G}_T$ of prime order p . Moreover, a pairing has two properties: bilinearity and non-degenerate maps. Bilinearity is that given a generator $g \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$ it holds that $e(g^a, g^b) = e(g, g)^{ab}$. Non-degenerate maps ensures that $e(g, g) \neq 1$. Lastly, cryptographic primitives that utilize lattices are an exciting area of research that hold promise for post-quantum cryptography. We briefly mention lattices in this paper, but defer to Regev's work [67] for an in-depth introduction.

We also discuss techniques for performing transformations over cryptographic primitives to achieve desired security properties. For example, Naor [40] proposed a technique for converting an IBE scheme into a public-key signature scheme. Canetti et al. [34] proposed a technique for transforming any IBE scheme into one that is secure against adaptive chosen-ciphertext attacks. In general, we refer to these types of techniques as *adapters* in this work.

Finally, we refer to Zero-knowledge Proofs of Knowledge [47], which allow one party to prove knowledge of a secret to another party without revealing the secret.

3 Approach

Charm realizes the afore-mentioned goals at the architectural level through various components and levels of modularization as depicted in Fig. 1.

We now describe the building blocks of the Charm framework. The lower-level components, at the bottom of Fig. 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 approach is our objective to simplify the code written by cryptographers who utilize the framework. Our modular component architecture reflects this.

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 can provide either IND-CPA or IND-CCA2 security. These properties become more relevant whenever the scheme is used as a building block for a more complex protocol.

Meta-information To address these issues, Charm must provide some mechanism to identify the pertinent information inherent in each scheme, including (but not limited to) input/output space, security definition, complexity assumptions, computational model, and performance characteristics. We defer the discussion of whether this should be done automatically or by the programmer to Sect. 4.

Capability matching Once this meta-information is collected, Charm uses it to facilitate compatibility among schemes. First, it provides tools to programmatically interrogate a scheme 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. 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. However, the implication can be bypassed in some cases, e.g., if EU-CMA is required and SU-CMA is not suitable for a given composition where re-randomizable signatures are required.

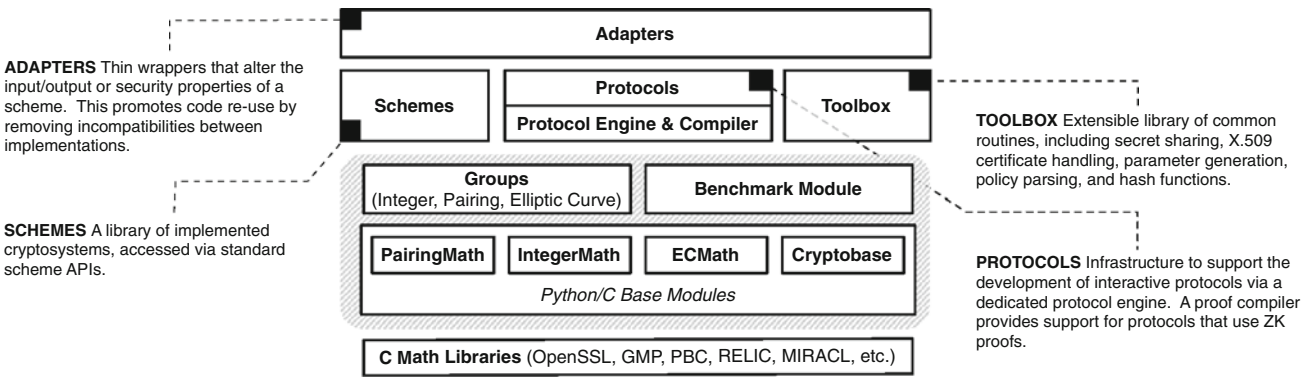


Fig. 1 Overview of the Charm architecture

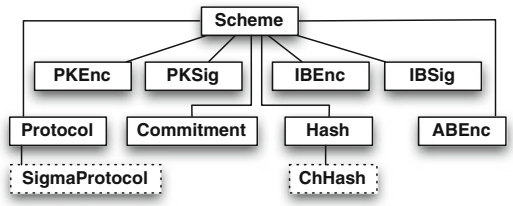


Fig. 2 Listing of scheme types defined in Charm. Subtypes are indicated with *dotted lines*

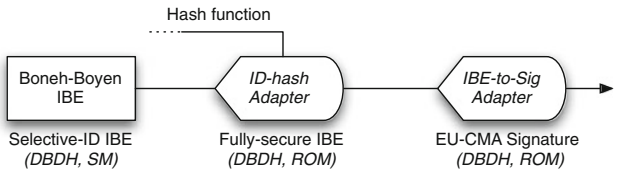


Fig. 3 Example of an adapter chain converting the Boneh–Boyen selective-ID secure IBE [17] into a signature scheme using Naor's technique [40]. The scheme carries meta-information including the complexity assumptions and computational model used in its security proof

Structured interfaces To facilitate scheme composition and reuse, 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. 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 Fig. 2. This list is sufficient for the schemes we have currently implemented (see Fig. 1), but we expect it to expand with the addition of new cryptosystems.

Adapters Since we now have enough information to safely and securely compose schemes, Charm includes *adapters* for this purpose and for handling mismatches between schemes. Adapters are code wrappers implemented as thin classes. For example, 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 Fig. 3, we illustrate an adapter using a hash function to perform a conversion from a selectively-secure IBE scheme into one that is adaptively secure (note here that the hash function is modeled as a random oracle).

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

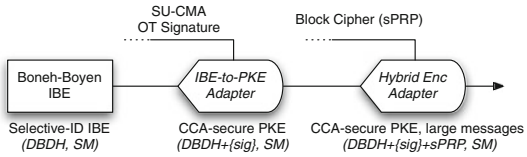


Fig. 4 Adapter chain converting the Boneh–Boyen selective-ID secure IBE [17] into a CCA-secure public-key hybrid encryption scheme via the CHK transform [34]. $\{sig\}$ stands for the complexity assumptions added by the signature scheme

library, demonstrating Charm's success at the goal of composability. Figure 4 provides another example of such a conversion.

Extensible numeric computation The mathematics underlying modern cryptography has changed considerably, driven by advances in lattices and pairings, and is sure to continue in this trend. It is fundamentally important that any system that wishes to maintain relevancy be able to incorporate these advances. By necessity, these libraries are implemented in C and require a certain specialty and expertise to implement correctly (e.g., elliptic curves). Charm provides domain separation by incorporating four base modules that implement the core cryptographic routines. This shelters developers from having to deal with very domain-specific concepts like elliptic curves. For performance reasons, these base modules are written in C/C++ and include integermath, ecmath (elliptic curve subgroups), and pairingmath.[2] 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 [46,60,61,73]. 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.

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, message encoding, and ciphertext parsing.We are continuously adding routines to the toolbox, and future releases will include contributions from external developers.

Protocol engine Interactive protocols often seem simple on paper, but in reality require a variety of different considerations. Zero-knowledge proofs are especially tricky as they often utilize information that is not specified in the documentation. General protocol 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 reusable *pro-*

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

tocol engine. An implementation in our framework consists 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 freely 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.

ZKP Compiler Zero-knowledge proofs of knowledge 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 [24,33]. These proofs may be interactive or non-interactive (via the Fiat–Shamir heuristic, or using new bilinear-map based techniques [43,48]). Regardless of the underlying mechanism, it has become common in the literature to describe such proofs using the notation of Camenisch and Stadler [32]. 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 to be 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 [5,63], providing native support for honest verifier Schnorr-type proofs via an automated protocol compiler.

Benchmarking system Performance is often critical when designing and implementing real-world cryptosystems. Therefore, developers are frequently interested in the efficiency of their schemes, both from a timing and computational perspective. They also might wonder how changes they make can affect these important aspects and how their schemes compare to others. In order to help developers 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 Sect. 6 were performed using the benchmarking system. The benchmarking system is easy to switch on or off and has minimal impact on the system when it is not in use. An example of using the benchmarking system is provided in Sect. 4.

4 Implementation

In this section, we describe our implementation and provide further details on components of our architecture. In Sect. 4.1 below, we reference an example comparing a protocol description from the literature to one implemented in our system. The code fragment shown in Fig. 5 is a good overall example of using Charm and is worth studying at this point to understand our approach.

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, 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 laboriously 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 Sect. 4.3). The features discussed here are not unique to Python and can be found in other high-level languages.[3] However, Python has a large and devoted user base and provides a good balance between usability, stability, and performance.[4]

Low-level Python/C modules As discussed in Sect. 3, for performance reasons, our implementation of Charm supports a variety of C math libraries including GMP [46], OpenSSL [73], RELIC [6], MIRACL [70] and the PBC library [61]. We provide Python/C extensions for these libraries.

Our base modules expose arithmetic operations using standard mathematical operators such as $*$, $+$ and $**$

[3] Nor are we the first to import cryptographic operations into Python. See, for example, [37,71].

[4] It is also well supported. Our experiments show that there have been significant performance improvements between Python 2.x and 3.x. Charm supports both versions for backwards compatibility with legacy applications.

CS98 Encryption

Encryption. Given a message $m \in G$, the encryption algorithm runs as follows. First, it chooses $r \in \mathbf{Z}_q$ at random. Then it computes

$$u_1 = g_1^r,\ u_2 = g_2^r,\ e = h^r m,\ \alpha = H(u_1, u_2, e),\ v = c^r d^{r\alpha}.$$

The ciphertext is (u_1, u_2, e, v).

```
def encrypt(self, pk, M):
    r     = group.random(ZR)
    u1    = (pk['g1'] ** r)
    u2    = (pk['g2'] ** r)
    e     = group.encode(M) * (pk['h'] ** r)
    alpha = group.hash((u1, u2, e))
    v     = (pk['c'] ** r) * (pk['d'] ** (r * alpha))

    return { 'u1' : u1, 'u2' : u2, 'e' : e, 'v' : v }
```

CS98 Decryption

Decryption. Given a ciphertext (u_1, u_2, e, v), the decryption algorithm runs as follows. It first computes $\alpha = H(u_1, u_2, e)$, and tests if

$$u_1^{x_1+y_1\alpha} u_2^{x_2+y_2\alpha} = v$$

If this condition does not hold, the decryption algorithm outputs "reject"; otherwise, it outputs $m = e/u_1^z$.

```
def decrypt(self, pk, sk, c):
    alpha = group.hash((c['u1'], c['u2'], c['e']))
    v_prime = (c['u1'] ** (sk['x1'] + (sk['y1'] * alpha))) *
              (c['u2'] ** (sk['x2'] + (sk['y2'] * alpha)))
    if (c['v'] != v_prime):
        return False
    return group.decode(c['e'] / (c['u1'] ** sk['z']))
```

Fig. 5 Encryption and decryption in the Cramer–Shoup scheme [38]. The *top box* shows the description of the algorithm in the published paper while the *bottom box* reflects the Charm code. Charm is designed to enable cryptographers to implement their schemes using mathematical notation that mirrors the paper description

(exponentiation).[5] Besides group operations, our base modules also perform essential functions such as element serialization and encoding.

In addition to the base modules, 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 such as CBC and CTR (drawn from PyCrypto [60] and libTomCrypt [39]) that facilitate encryption of arbitrary amounts of data.

Benchmark module As described in Sect. 3, we provide a benchmark module for measuring computation time and counting operations, such as exponentiations and multiplications, 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
msmtDict = GetGeneralBenchmarks(bID)
print(msmtDict[Exp])
# reset benchmarks
ClearBenchmark(bID)
```

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

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 module 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 [38,42,65] can be instantiated in any group with an appropriate structure.

4.1 Schemes

To demonstrate the potential of our framework, we implemented a number of standard and experimental cryptosystems. We provide a collection of implemented schemes that includes a variety of encryption schemes, signatures, commitments, and interactive protocols.[6] Most of the implementations consist of fewer than 100 lines of code (see Table 1 for a listing).

We provide several examples to illustrate code in Charm. Figure 5 shows the encryption and decryption algorithms for the Cramer-Shoup [38] scheme, and the corresponding Charm code. We provide the remaining algorithms, along with some additional examples, in Appendix. We note that our framework was designed to minimize the differences

[5] 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.

[6] For more scheme implementations, see http://jhuisi.github.com/charm/schemes.html.

Table 1 A listing of the cryptographic schemes we implemented

Scheme	Type	Setting	Comp. model	Lines
Encryption				
RSA-OAEP [10]	Public-key encryption	Integer	ROM	22
CS98 [38]	Public-key encryption	EC/Integer	Standard	40
ElGamal [16]	Public-key encryption	EC/Integer	Standard	34
Paillier99 [72]	Public-key encryption	Integer	Standard	31
BF01 [19]	Identity-based encryption	Pairing	ROM	51
BB04 [17]	Identity-based encryption	Pairing	Standard	45
Waters05 [76]	Identity-based encryption	Pairing	Standard	49
CKRS09 [27]	Identity-based encryption	Pairing	Standard	55
LSW08 [57]	Identity-based encryption	Pairing	ROM[a]	69
SW05 [69]	Fuzzy identity-based encryption	Pairing	Standard	68
BSW07 [14]	Attribute-based encryption	Pairing	ROM[a]	62
Waters08 [77]	Attribute-based encryption	Pairing	ROM[a]	61
LW10 [58]	MA Attribute-based encryption	Pairing	ROM[a]	67
FE12 [78]	DFA-based functional encryption	Pairing	Standard	71
HVE08 [53]	Hidden vector encryption	Pairing	Standard	104
Digital signatures				
Schnorr [23]	Signature	Integer	ROM	33
RSA-PSS [11]	Signature	Integer	ROM	32
EC-DSA & DSA [65]	Signature	EC/Integer	n/a	32
HW09 [51]	Signature	Integer	Standard	113
CHP [26]	Signature	Pairing	Standard	30
CL03 [29]	Signature	Integer	Standard	58
CL04 [30]	Signature	Pairing	ROM	25
HW [51]	Signature	Pairing	Standard	48
Hess [50]	Identity-based signature	Pairing	ROM	31
CHCH [35]	Identity-based signature	Pairing	ROM	31
Waters05 [76]	Identity-based signature	Pairing	Standard	43
Boyen [22]	Ring Signature	Pairing	CRS	65
CYH [36]	Ring Signature	Pairing	ROM	58
BLS03 [21]	Short signature	Pairing	ROM	23
BBS04 [18]	Group signature	Pairing	ROM	60
Adapters				
CHK04 [34], BCHK05 [20]	IBE-to-PKE	–	–	23, 63
IBE-to-Signature [19]	Signature	–	–	24
Hybrid ABE	Hybrid ABE	–	–	27
Hybrid DABE	Hybrid DABE	–	–	28
Hybrid KPABE	Hybrid KPABE	–	–	26
Hybrid IBE [34]	Hybrid IBE	–	–	27
IBE Identity Hash	IBE	–	–	35
Hybrid PKE	Hybrid PKE	–	–	30
Miscellaneous				
GS07 [48]	Commitment	Pairing	CRS	28
Pedersen [66]	Commitment	EC/Integer	Standard	16
AdM05 [7]	Chameleon hash	Integer	ROM	24
RSA HW09 [51]	Chameleon hash	Integer	Standard	29
VRF [52]	Verifiable random functions	Pairing	Standard	47

Table 1 continued

Scheme	Type	Setting	Comp. model	Lines
Protocols				
Schnorr91 [23]	Zero-knowledge proof	EC/Integer	Standard	53
CNS07 [31]	Oblivious Transfer	Pairing	Standard	147

"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, CRS indicates that a scheme is secure in the Common Reference String Model. A "–" indicates a generic transform (adapter)
[a] A choice made for efficiency reasons

between published algorithms and code (as shown in Fig. 5), in the hope of lowering the barriers to implementation.

4.2 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.

Creating a new interactive protocol is straightforward. The implementation must provide 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 are 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 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 references as part of the public parameters. We plan to provide more complete support for asynchronous execution in future releases.

4.3 ZKP compiler

Many advanced cryptographic protocols (e.g., [18,28,31]) employ zero-knowledge or witness-indistinguishable proofs as part of their protocol structure. The notation of Camenisch and Stadler [32] has become the de facto standard in the cryptography 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 have been implemented in the past by Meiklejohn et al. (ZKPDL) [63] and Bangerter et al. (CACE) [9]. 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.[7] Second, since our compiler has access to the public and secret[8] 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 Protocol Specification Language (PSL) program. Thus, from a developer's perspective, executing a ZK proof is nearly as simple as writing a Camenisch-Stadler statement.

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

[8] Clearly the verifier does *not* have access to the secret variables. We address this later in this section.

```
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
```

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

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 the 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
public = {'h':g ** x, 'g':g, 'j':g ** y}
secret = {'x':x, 'y':y}
result = executeIntZKProof(public, secret,
      ''(h = g^x) and (j = g^y)'',
       party_info)
```

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

The runtime technique is 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.

At present, our compiler is intended as a proof of concept because it lacks support for many types of statements (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.

4.4 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. Wherever possible, this information is derived automatically, e.g., from the scheme type or function definitions. Optionally, developers can provide other details such as the complexity assumption and computational models used in the scheme's security proof via a standard annotation interface. 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, messageSpace=str)
```

Schemes with more restrictive parameters, e.g., group elements and/or strings of limited length, can specify these

requirements as well.[9] Once each scheme is labeled with the appropriate metadata, 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 Sect. 3, we proposed an adapter chain to convert the Boneh-Boyen IND-sID-CPA-secure signature scheme [17] into an EU-CMA signature (see Fig. 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 [40].

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 $\mathbb{Z}_r$,[10] 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.[11] The adapter itself is implemented as a subclass of `IBEnc` (see Fig. 9a in Appendix). It accepts the Boneh-Boyen IBE (also an `IBEnc` class) as input to its constructor. At construction time, the 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(self, scheme,
    [('scheme','IBEnc'),('secDef',
    IND_sID_CPA),
      ('id',ZR)]):
   self.ibe = scheme
   IBEnc.updateProperty(self, scheme,
         secDef=IND_ID_CPA, id=str,
         secModel=ROM)
...
```

The IBE-to-Sig adapter converts any adaptively-secure IBE scheme into an EU-CMA signature.[12] 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, and 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.

4.5 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 example, 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 elements. 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}

@Input(pk_t, str)
@Output(c_t)
def encrypt(self, pk, M):
    ...
```

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.

4.6 Using Charm in C applications

To enable the use of Charm schemes in existing C applications, we provide an embed API for integrating Charm

[9] In some cases, evaluation of a scheme depends on the scheme's public key.

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

[11] 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 [17].

[12] Naor [40] observed that adaptively-secure IBE can be converted into a signature scheme by using the IBE key extraction algorithm for signing.

schemes without burdening developers. Our approach achieves two important goals. First, the embed API is easy-to-use, intuitive, and straightforward for developers to use a scheme based on its scheme type API (e.g., keygen, encrypt/decrypt). Second, the API allows C applications to interchange primitives of the same type with minimal modifications.

To embed a scheme, the application first calls the InitializeCharm() function to setup the Charm environment. Once Charm is setup, the application creates a group object for instantiating a scheme. This is accomplished by calling the group initialization function for a given setting such as InitPairingGroup(), InitIntegerGroup(), etc. Next, the application calls InitScheme() and includes the scheme file name, class name, and the group object handle returned from the previous call. To call any function within the scheme, the application uses the CallMethod() and supplies the arguments for the target function. Finally, we provide serialization methods (objectToBytes() and bytesToObject()) for converting Charm objects to/from base-64 encoded binary strings. We believe our simple embed API enables Charm to be seamlessly integrated into a variety of applications that require advanced cryptographic constructs. For a detailed example, see Fig. 8 in Appendix.

5 Goals and challenges

The goal of the Charm framework is to provide a usable, extensible, and modular architecture to support rapid prototyping of a variety of cryptographic primitives and protocols. Our implementation provides the necessary building blocks to achieve this central goal. For example, at the lowest level, we provide abstract C/C++ interfaces around the C math libraries to make them interchangeable at build time. This allows cryptographers to evaluate their scheme implementations against different libraries by only changing the Charm install configuration. With the pairingmath module, for instance, we can evaluate the performance of schemes against the PBC, MIRACL, and RELIC libraries without changing the scheme itself. It is relatively easy to extend our framework with new math libraries that adhere to our C/C++ abstract interface. Moreover, we are able to extend our platform to diverse environments with relatively low effort and without affecting the higher level components in Charm. Thus, all of these features enable Charm to provide a test bed for rapidly prototyping and evaluating advanced cryptosystems against any appropriate underlying C library.

While the Charm architecture addresses a number of issues to facilitate rapid implementations of modern cryptography, it did not come without technical challenges. Our first challenge was determining the interface that should be exposed in Python for building schemes and protocols in a way that is standard and comprehensive. The second challenge was conforming the math libraries to this interface. This was not a significant issue for well-established math libraries, such as GMP, OpenSSL, PBC, and MIRACL. However, for more recent research libraries such as RELIC, this presented challenges due to missing functionality (e.g., serialization) and the alpha software quality of the pairings interface. But given the optimizations available in RELIC for pairings, it has the potential to become the standard for pairing-based cryptography in the near future.

6 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 facilitate more semantically correct, robust, and secure code. However, we recognize that achieving these properties is likely to come at a tradeoff in performance.

As such, in this section, we report representative performance metrics collected through the use of Charm's built-in benchmarking system. These metrics are quantitatively compared against detailed timing experiments of two existing C cryptographic system implementations. We observe that the performance cost of using Charm is variable, and it is directly dependent on the nature of the scheme implementation.

6.1 Comparison with C implementations

We conducted detailed timing experiments on two of the cryptosystems we implemented: EC-DSA and a CP-ABE scheme due to Bethencourt, Sahai, and Waters [14]. We chose these two because of their available C implementations, thus realistic choices against which to compare. Our experiments comprise two different points on a spectrum: our EC-DSA 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).

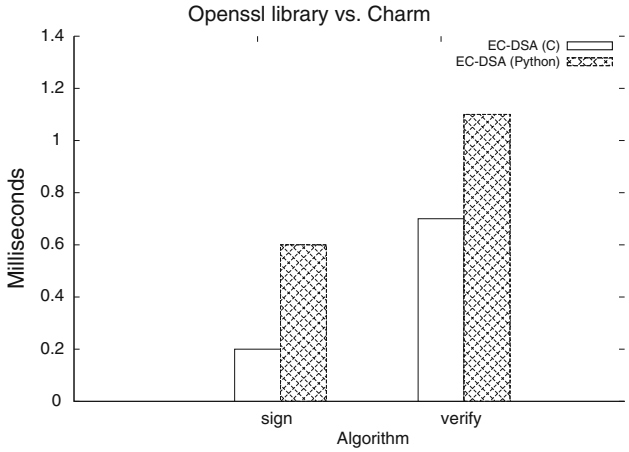
Experimental setup We used the benchmark module to collect timings for our Charm implementation of the EC-DSA Sign and Verify algorithms. This provided us with total operation time for both algorithms. We then collected total operation times for OpenSSL's implementation of the same algorithms using the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

For CP-ABE, we used benchmark again 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

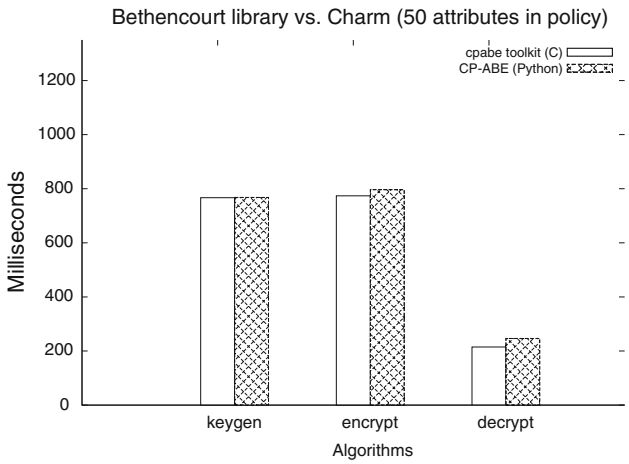
group $\mathbb{G}_T$) under a policy consisting solely of AND gates: (1 AND 2 AND ... and 50). Finally, we decrypted the message using the extracted key. For each experiment, we measured total time and repeated these experiments using John Bethencourt's library (available from [1]) to obtain the C time.

We conducted our experiments on a Macbook Pro with a 2.4Ghz Intel i5 with 8GB of RAM running Mac OS 10.7 and Python v3.2.3. All of our experiments were performed on a single core of the processor. For all experiments (Charm and C), we used either OpenSSL v1.0.1c library or libpbc 0.5.12 to perform the underlying mathematical operations. Our EC-DSA 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.

Experimental results The results of our experiments are presented in Fig. 7. Unsurprisingly, our Charm implementation of EC-DSA suffered a substantial performance penalty when compared to the OpenSSL version. This is unavoidable given the relatively low overall time required for EC-DSA operations—even small interpretation inefficiencies add up to a large percentage of the total cost. Our results with CP-ABE (and 50 attributes) are encouraging. For the CP-ABE algorithms, Charm is competitive with the C implementation. As a result, we believe that Charm can be a primary tool for cryptographers wishing to approximate the performance of their schemes or protocols in practice [68]. For additional performance measurements, see our technical report [4].



(a) Comparison to OpenSSL



(b) Comparison to Beth-cpabe toolkit

Fig. 7 For EC-DSA, we select the NIST P-192 elliptic curve and for CP-ABE [14], we measure 50 attributes for keygen and 50 leaves in the policy tree for encrypt and decrypt. Comparison to OpenSSL.Comparison to Beth-cpabe toolkit

7 Related work

Our work builds upon previous efforts to provide software libraries for developers who use cryptography. We describe four different types of libraries below.

Cryptographic (primitive) libraries The first widely available general purpose library for commonly used cryptographic functions was Jack Lacey's CryptoLib [54]. Following CryptoLib, many other packages were developed, including Peter Guttman's similarly named CryptLib[13], RSA's Bsafe Crypto-C[14], and more recently JAVA libraries such as Cryptix[15], BouncyCastle[16]. While these libraries have been useful for application developers, they were designed for specific and mostly isolated purposes. Moreover, they only implement commonly used and standardized cryptographic functions.

There have not been as many implementations of cryptosystems such as IBE, ABE, and related advanced primitives. Of note is the implementation by Bethencourt, Sahai and Waters [14], which provides an API for ciphertext policy ABE. This package is part of the Advanced Crypto Software collection (ACSC) [1], which, in addition to this ABE library, includes separate packages for other advanced application-based primitives such as forward-secure signatures and broadcast encryption. Our Charm architecture provides a comprehensive and unified framework that is both usable and developer friendly for rapid prototyping of advanced primitives.

Math libraries The GNU Multiple Precision Arithmetic Library (GMP) [46] is a free, high-precision mathematics library, specifically optimized for speed of cryptographic algorithms. The Stanford Pairing-Based Cryptography (PBC) library [61] is free, written in C, and built on top of GMP to provide abstractions for developing pairing algorithms. PBC was built for expressiveness, but not designed for usability or performance. RELIC [6], also an open source library which

[13] http://www.cs.auckland.ac.nz/~pgut001/cryptlib/.

[14] http://www.rsa.com/rsalabs/node.asp?id=2301.

[15] http://www.cryptix.org/.

[16] http://www.bouncycastle.org/.

relies on GMP, was built for speed and portability with support for big number arithmetic, traditional elliptic curves and pairings. While RELIC is highly configurable and supports a variety of cryptographic optimizations, it was not primarily built for usability.

The Multiprecision Integer and Rational Arithmetic Library (MIRACL) [70] is written in C/C++ and provides APIs for big number arithmetic, elliptic curve cryptography, block ciphers and hash functions. Similar to RELIC, MIRACL is a highly optimized library that is compatible with a variety of architectures and is quite expressive in terms of functionality. However, MIRACL places a secondary focus on usability. Using the library effectively requires knowledge of its inner workings. Our Charm framework shields developers from dealing with these libraries directly via layers of abstractions. Instead, cryptographers can utilize our abstractions to implement their schemes or protocols using standard notation and evaluate them against any of the math libraries supported in Charm.

Cryptographic compilers and frameworks Ben Laurie's *Stupid* programming language [55] compiles into C and Haskell and is intended for constructs like ciphers and hash functions. Cryptol [56] compiles to a VHDL circuit for use with an FPGA. More recently, Dan Bernstein's *NaCl* (or "salt") [12] software library in C/C++ provides an easy-to-use interface (e.g., encryption, decryption, signatures, etc.) to build higher-level cryptographic tools.

Protocol and secure function evaluation compilers The authors of the Zero Knowledge Proof Descriptive Language (ZKPDL) [63] offer a language and an interpreter for implementing privacy-preserving protocols. Their example application is electronic cash, but their descriptive language is more general. A similar approach is provided by FairPlay [62], 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. Charm takes a similar approach but with a focus on providing a simple language in the Camenisch and Stadler [32] notation for specifying high-level proof statements. From this proof statement, our compiler automatically generates the interactive protocol details.

A software package called Tool for Automating Secure Two-Party Computations (TASTY) [49] 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. Similarly, the Computer Aided Cryptography Engineering (CACE) project has also developed a system that specifies a language for zero knowledge proofs [9,8]. In this system, a compiler translates zero-knowledge protocol specifications into Java code or LaTeX statements. The CACE ZK compiler has many features, optimizations, and performance benefits. Our framework is certainly compatible with the CACE design and we intend on leveraging CACE as a building block in Charm.

8 Availability

The Charm framework is freely available at http://charm-crypto.com/Download.html with extensive documentation[17] for how to use it. To make Charm easy-to-use, we provide automated installers for various platforms such as Windows, Mac OS X and Linux. Additionally, to support embedded environments, we have ported the framework to mobile platforms such as Android. Our end goal is to enable Charm on as many platforms as possible.

9 Conclusion

This paper describes Charm, a framework for rapidly prototyping cryptographic systems. We believe that the approaches outlined in this paper, together with the Python implementation, constitute a solution to many of the theoretical and practical shortcomings of existing cryptographic prototyping tools. We believe that the framework is easy enough to not deter implementers yet powerful enough to capture many of the recent developments in cryptography. It is our aim to encourage others to develop schemes and to contribute them to the framework.

An open area is to develop automated compilers for performing various operations on cryptographic schemes. One such example is the translation of schemes between various settings, e.g., composite-order to prime-order bilinear groups. Both David Freeman [44] and Alison Lewko [59] have recently proposed tools for this type of translation; however, all of these tools currently require human intervention. We believe that Charm provides an excellent platform for implementing techniques that *automatically* translate such schemes (represented in a domain-specific language) to working implementations.

On the engineering side, there are a number of issues related to improving Charm for applications that require extremely high performance. For example, the current Python threading model is not ideal for applications that would benefit from substantial parallel processing (e.g., lattice-based fully-homomorphic encryption schemes [45]). One of our major open problems is to find ways to take full advantage of multi-core systems. Finally, we understand that there may be instances where development requirements cannot support a high-level interpreted language such as

[17] http://charm-crypto.com/Documentation.html.

Python. To address this, we plan to examine the possibility of compiling Charm code directly to languages such as Haskell and C, using tools such as Shedskin [41].

Appendix

See Figs. 8, 9, 10, 11, 12, 13.

Using BSW07 Scheme in C

```
# variable declarations
Charm_t *group,*cpabe,*hyabe,*keyTupl,*recmsg;
Charm_t *pkDict,*mskDict,*skDict,*ctDict,*ctBlob;
char *msg,*policy,*attrlist;

# setup Charm environment
InitializeCharm();
# initialize group with super singular curve
# and 512-bits for base field.
group = InitPairingGroup(module, "SS512");
 # initialize the scheme
cpabe = InitScheme("abenc_bsw07",
    "CPabe_BSW07",group);
# call to initialize adapters
hyabe  = InitAdapter("abenc_adapt_hybrid",
    "HybridABEnc",cpabe,group);
# no arguments to setup
keyTupl = CallMethod(hyabe, "setup", "");
# extract master public & private keys
pkDict = GetIndex(keyTupl, 0);
mskDict = GetIndex(keyTupl, 1);
# call keygen
attrlist = "[SALES, IT]";
skDict = CallMethod(hyabe,"keygen","%O%O%A",
    pkDict,mskDict,attrlist);
# call encrypt
msg = "this is a test message";
policy = "(CORPORATE and (SALES or IT))";
ctDict = CallMethod(hyabe,"encrypt","%O%b%s",
    pkDict,msg,policy);
# serialize object into base-64 string
ctBlob = objectToBytes(ctDict, group);
# call decrypt
recmsg = CallMethod(hyabe,"decrypt","%O%O%O",
    pkDict,skDict,ctDict);
. . . free Charm_t variables . . .
# tear down the Charm environment
CleanupCharm();
```

Fig. 8 A working example of how the API is utilized in a C application to embed a hybrid encryption adapter (see Fig. 9b) for any CP-ABE scheme such as the BSW07 [14] scheme shown in Figs. 11 and 12. We provide several high-level functions that simplify using Charm schemes. In particular, the CallMethod() encapsulates several types of arguments to Python such as: %O for Charm objects, %s for ASCII strings, %A to convert into a Python list, and %b to a binary object

IBE-to-Sig Adapter

```
def __init__(self, scheme, groupObj):
    PKSig.__init__(self)
    global ibe, group
    condition = [('secDef',IND_ID_CPA),('scheme','IBenc'),
                 ('messageSpace',GT)]
    if PKSig.checkProperty(self, scheme, condition):
       # inherit properties of scheme & update definitions
       PKSig.updateProperty(self, scheme, secDef=EU_CMA,
          id=str, secModel=ROM)
       ibe = scheme; group = groupObj
```

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

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

def verify(self, pk, m, sig):
    if hasattr(ibe, 'verify'):
       result = ibe.verify(pk, m sig)
       if result == False: return False
    new_m = group.random(GT)
    C = ibe.encrypt(pk, sig['IDstr'], new_m)
    if ibe.decrypt(sig, C) == new_m:
       return True
    else:
       return False
```

(a) IBE-to-Sig Adapter

Hybrid-Enc-ABE Adapter

```
class HybridABEnc(ABEnc):
    def __init__(self, scheme, groupObj):
        ABEnc.__init__(self)
        global abenc, group
        # ... verify scheme properties ...
        abenc = scheme
        group = groupObj

    def setup(self):
        return abenc.setup()

    def keygen(self, pk, mk, object):
        return abenc.keygen(pk, mk, object)
```

```
    def encrypt(self, pk, M, object):
        key = group.random(GT)
        c1 = abenc.encrypt(pk, key, object)
        # init a symmetric enc scheme from this key
        cipher = AuthCryptoAbstraction(sha1(key))
        c2 = cipher.encrypt(M)
        return { 'c1':c1, 'c2':c2 }

    def decrypt(self, pk, sk, ct):
        c1, c2 = ct['c1'], ct['c2']
        key = abenc.decrypt(pk, sk, c1)
        cipher = AuthCryptoAbstraction(sha1(key))
        return cipher.decrypt(c2)
```

(b) Hybrid Enc Adapter

Fig. 9 **a** The entire IBE to signature adapter scheme [19]. **b** A hybrid encryptor for ABE schemes in Charm

CS98 Keygen Description

Key Generation. The key generation algorithm runs as follows. Random elements $g_1, g_2 \in G$ are chosen, and random elements $x_1, x_2, y_1, y_2, z \in \mathbf{Z}_q$ are also chosen. Next, the group elements

$$c = g_1^{x_1} g_2^{x_2},\ d = g_1^{y_1} g_2^{y_2},\ h = g_1^{z}$$

are computed. Next, a hash function H is chosen from the family of universal one-way hash functions. The public key is (g_1, g_2, c, d, h, H), and the private key is (x_1, x_2, y_1, y_2, z).

Charm Implementation

```
def keygen(self, secparam):
    # code for checking group setting
    g1, g2 = group.random(G, 2)
    x1, x2, y1, y2, z = group.random(ZR, 5)
    c = (g1 ** x1) * (g2 ** x2)
    d = (g1 ** y1) * (g2 ** y2)
    h = (g1 ** z)
    pk = { 'g1':g1, 'g2':g2, 'c':c, 'd':d, 'h':h }
    sk = { 'x1':x1, 'x2':x2, 'y1':y1, 'y2':y2, 'z':z }
    return (pk, sk)
```

Fig. 10 Keygen in the Cramer–Shoup scheme [38]. We exclude group parameter generation

BSW07 Scheme Description

Setup. The setup algorithm will choose a bilinear group $\mathbb{G}_0$ of prime order p with generator g. Next it will choose two random exponents $\alpha, \beta \in \mathbb{Z}_p$. The public key is published as:

$$\mathrm{PK} = \mathbb{G}_0, g, h = g^{\beta}, f = g^{1/\beta}, e(g,g)^{\alpha}$$

and the master key MK is (β, g^{α}). (Note that f is used only for delegation.)

Charm Implementation

```
def setup(self):
    g, gp = group.random(G1), group.random(G2)
    alpha, beta = group.random(ZR), group.random(ZR)
    g_alpha = gp**alpha

    pk = { 'g': g, 'g2': gp, 'h': g**beta,
           'f': g**~beta,
           'egg_alpha': pair(g, g_alpha) }
    mk = { 'beta':beta, 'g2_alpha': g_alpha }

    return (pk, mk)
```

KeyGen(MK, S). The key generation algorithm will take as input a set of attributes S and output a key that identifies with that set. The algorithm first chooses a random $r \in \mathbb{Z}_p$, and then random $r_j \in \mathbb{Z}_p$ for each attribute $j \in S$. Then it computes the key as

$$\mathrm{SK} = \big(D = g^{(\alpha+r)/\beta},\ \forall j \in S:\quad D_j = g^{r} \cdot H(j)^{r_j}, D'_j = g^{r_j}\big).$$

```
def keygen(self, pk, mk, S):
    r = group.random(ZR); g_r = (pk['g2'] ** r)
    D = (mk['g2_alpha'] * g_r) ** (1 / mk['beta'])
    D_j, D_j_pr = {}, {}
    for j in S:
        r_j = group.random(ZR)
        D_j[j] = g_r * (group.hash(j, G2) ** r_j)
        D_j_pr[j] = pk['g'] ** r_j

    return { 'D':D, 'Dj':D_j, 'Djp':D_j_pr, 'S':S }
```

Fig. 11 Setup and Keygen in the Bethencourt, Sahai, and Waters scheme [14]. We exclude group parameter generation

BSW07 Scheme Description

Encrypt(PK, $M, \mathcal{T}$). The encryption algorithm encrypts a message M under the tree access structure $\mathcal{T}$.

. . .

Let, Y be the set of leaf nodes in $\mathcal{T}$. The ciphertext is then constructed by giving the tree access structure $\mathcal{T}$ and computing

$$\mathrm{CT} = \big(\mathcal{T},\ \tilde{C} = Me(g,g)^{\alpha s},\ C = h^{s},\ \forall y \in Y:\quad C_y = g^{q_y(0)}, C'_y = H(\mathsf{att}(y))^{q_y(0)}\big).$$

Charm Implementation

```
def encrypt(self, pk, M, policy_str):
   policy = util.createPolicy(policy_str)
   Y = util.getAttributeList(policy)
   s = group.random(ZR)
   share = util.calculateSharesDict(s, policy)
   C_y, C_yp = {}, {}
   for i in Y.keys():
     j = util.strip_index(i)
     C_y[i] = pk['g'] ** share[i]
     C_yp[i] = group.hash(j, G2) ** share[i]
   return { 'C_tilde': (pk['e_gg_alpha'] ** s) * M,
            'C': pk['h'] ** s,
            'Cy': C_y, 'Cyp':C_yp,
            'policy':policy,'attributes':Y }
```

Decrypt(CT, SK).

Direct computation of DecryptNode (optimization):

$$z_\ell = \prod_{\substack{x \in \rho(\ell) \\ x \neq r}} \Delta_{i,S}(0) \quad \text{where } S = \{\, \mathsf{index}(y) \mid y \in \mathsf{sibs}(x) \,\}^{\,i = \mathsf{index}(x)}$$

$$\mathrm{DecryptNode}(\mathrm{CT}, \mathrm{SK}, r) = \prod_{\substack{\ell \in L \\ i = \mathsf{att}(\ell)}} \left(\frac{e(D_i, C_\ell)}{e(D'_i, C'_\ell)}\right)^{z_\ell}$$

$$A = \mathrm{DecryptNode}(\mathrm{CT}, \mathrm{SK}, R)$$

$$\tilde{C}/(e(C,D)/A) = \tilde{C}/\left(e\left(h^{s}, g^{(\alpha+r)/\beta}\right)/e(g,g)^{rs}\right) = M$$

```
def decrypt(self, pk, sk, ct):
   policy = util.createPolicy(ct['policy'])
   pruned_list = util.prune(policy, sk['S'])
   if pruned_list == False: return False
   z = util.getCoefficients(ct['policy'])
   A = 1
   for i in pruned_list:
     # j has index in case of duplicate attributes
     j = i.getAttributeAndIndex(); k = i.getAttribute()
     A *= ( pair(ct['Cy'][j],sk['Dj'][k])
            / pair(sk['Djp'][k],ct['Cyp'][j]) ** z[j]

   return ct['C_tilde']/((pair(ct['C'], sk['D']) / A)
```

Fig. 12 Encryption and decryption in the Bethencourt, Sahai, and Waters ABE scheme [14]. The Charm toolbox provides several utility routines that are shared by different ABE schemes

Fig. 13 CL signatures [30] are a useful building block for anonymous credential systems. We provide a full scheme description and Charm code, but exclude group parameter generation

CL04 Scheme Description

Key generation. The key generation algorithm runs the *Setup* algorithm in order to generate $(q, G, \mathsf{G}, g, \mathsf{g}, e)$ It then chooses $x \leftarrow \mathbb{Z}_q$ and $y \leftarrow \mathbb{Z}_q$, and sets $sk = (x, y)$, $pk = (q, G, \mathsf{G}, g, \mathsf{g}, e, X, Y)$.

Signature. On input message m, secret key $sk = (x, y)$, and public key $pk = (q, G, \mathsf{G}, g, \mathsf{g}, e, X, Y)$, choose a random $a \in G$, and output the signature $\sigma = (a, a^y, a^{x+mxy})$.

Verification. On input $pk = (q, G, \mathsf{G}, g, \mathsf{g}, e, X, Y)$, message m, and purported signature $\sigma = (a, b, c)$, check that

$$e(a, Y) = e(g, b) \qquad \text{and} \qquad e(X, a) \cdot e(X, b)^m = e(g, c) \tag{1}$$

holds.

```
def keygen(self):
    g = group.random(G1)
    x, y = group.random(ZR, 2)
    sk = { 'x':x, 'y':y }
    pk = { 'X':g ** x, 'Y':g ** y, 'g':g }
    return (pk, sk)

def sign(self, sk, M):
    (x, y) = sk['x'], sk['y']
    a = group.random(G2)
    m = group.hash(M, ZR)
    sig = { 'a':a, 'b':a ** y, 'c':a ** (x + (m * x * y)) }
    return sig

def verify(self, pk, M, sig):
    (a, b, c) = sig['a'], sig['b'], sig['c']
    m = group.hash(M, ZR)
    if pair(pk['Y'],a) == pair(pk['g'],b) and
        (pair(pk['X'],a) * (pair(pk['X'],b) ** m)) == pair(pk['g'],c):
       return True
    return False
```

References

1. The Advanced Crypto Software Collection. http://acsc.cs.utexas.edu/
2. Acar, T., Belenkiy, M., Bellare, M., Cash, D.: Cryptographic agility and its relation to circular encryption. In: EUROCRYPT (2010)
3. Acar, T., Fournet, C., Shumow, D.: Design and verication of a crypto-agile distributed key manager (2011)
4. Akinyele, J.A., Green, M., Rubin, A.: Charm-crypto framework. http://eprint.iacr.org/2011/617
5. Almeida, J.B., Bangerter, E., Barbosa, M., Krenn, S., Sadeghi, A.-R., 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, ESORICS, pp. 151–167. Springer, Berlin (2010)
6. Aranha, D.F., Gouvêa, C.P.L.: RELIC is an efficient library for cryptography. http://code.google.com/p/relic-toolkit/
7. Ateniese, G., de Medeiros, B.: On the key exposure problem in chameleon hashes. In: SCN. LNCS vol. 3352, pp. 165–179. Springer, Berlin (2004)
8. Bangerter, E., Barzan, S., Sadeghi, A., Schneider, T., Tsay, J.: Bringing zero-knowledge proofs of knowledge to practice. In: 17th International Workshop on Security Protocols (2009)
9. Bangerter, E., Camenisch, J., Krenn, S., Sadeghi, A.-R., Schneider, T.: Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471 (2008). http://eprint.iacr.org/
10. Bellare, M., Rogaway, P.: Optimal asymmetric encryption padding—how to encrypt with rsa. In: EUROCRYPT, pp. 92–111 (1994)
11. Bellare, M., Rogaway, P.: The exact security of digital signatures: how to sign with RSA and Rabin. In: Maurer, U (ed.) EUROCRYPT. LNCS, vol. 1070. Springer, Berlin (1996)
12. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) Progress in cryptology—LATINCRYPT. Lecture Notes in Computer Science. Springer, Berlin (2012, to appear). Document ID: 5f6fc69cc5a319aecba43760c56fab04, http://cryptojedi.org/papers/
13. Bethencourt, J.: Libpaillier (2006)
14. Bethencourt, J., Sahai, A., Waters, B.: Ciphertext-policy attribute-based encryption. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy, pp. 321–334. IEEE Computer Society, New York (2007)
15. Bethencourt, J., Song, D., Waters, B.: Analysis-resistant malware. In: NDSS (2008)
16. Blakley, G., Chaum, D., ElGamal, T.: A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms, vol. 196, pp. 10–18. Springer, Berlin (1985)
17. Boneh, D., Boyen, X.: Efficient selective-ID secure identity-based encryption without random oracles. In: EUROCRYPT. LNCS, vol. 3027, pp. 223–238 (2004)
18. Boneh, D., Boyen, X., Shacham, H.: Short group signatures. In: CRYPTO. LNCS, vol. 3152, pp. 45–55 (2004)
19. Boneh, D., Franklin, M.K.: Identity-based encryption from the Weil Pairing. In: CRYPTO. LNCS, vol. 2139, pp. 213–229 (2001)
20. Boneh, D., Katz, J.: Improved efficiency for cca-secure cryptosystems built using identity based encryption. In: CT-RSA. LNCS, vol. 3376. Springer, Berlin (2005)
21. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil Pairing. In: ASIACRYPT. LNCS, vol. 2248, pp. 514–532 (2001)

22. Boyen, X.: Mesh signatures: how to leak a secret with unwitting and unwilling participants. In: EUROCRYPT. LNCS, vol. 4515, pp. 210–227. Springer, Berlin (2007)
23. Brassard, G., Schnorr, C.: Efficient Identification and Signatures for Smart Cards, vol. 435, pp. 239–252. Springer, Berlin (1990)
24. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS, pp. 132–145. ACM, New York (2004)
25. Camenisch, J., Groth, J.: Group signatures: better efficiency and new theoretical aspects. In: Blundo, C., Cimato, S. (eds.) Security in Communication Networks. Lecture Notes in Computer Science, vol. 3352, pp. 120–133. Springer, Berlin (2005)
26. Camenisch, J., Hohenberger, S., Stergaard Pedersen, M.: Batch verification of short signatures. In: EUROCRYPT. LNCS, vol. 4515. Springer, Berlin, pp. 246–263 (2007)
27. Camenisch, J., Kohlweiss, M., Rial, A., Sheedy, C.: Blind and anonymous identity-based encryption and authorised private searches on public key encrypted data. In: PKC, Irvine, pp. 196–214. Springer, Berlin (2009)
28. Camenisch, J., Lysyanskaya, A.: An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In: EUROCRYPT. LNCS, vol. 2045, pp. 93–118. Springer, Berlin (2001)
29. Camenisch, J., Lysyanskaya, A.: A signature scheme with efficient protocols. In: Proceedings of the 3rd International Conference on Security in Communication Networks, SCN, pp. 268–289. Springer, Berlin (2003)
30. Camenisch, J., Lysyanskaya, A.: Signature Schemes and Anonymous Credentials from Bilinear Maps, pp. 56–72. Springer, Berlin (2004)
31. Camenisch, J., Neven, G., Shelat, A.: Simulatable adaptive oblivious transfer. In: EUROCRYPT. LNCS, vol. 4515, pp. 573–590 (2007)
32. Camenisch, J., Stadler, M.: Efficient group signature schemes for large groups. In: CRYPTO. LNCS, vol. 1296, pp. 410–424 (1997)
33. Camenisch, J., Van Herreweghen, E.: Design and implementation of the idemix anonymous credential system. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS, pp. 21–30. ACM, New York (2002)
34. Canetti, R., Halevi, S., Katz, J.: Chosen-ciphertext security from identity based encryption. In: EUROCRYPT. LNCS, vol. 3027, pp. 207–222 (2004)
35. Cha, J.C., Cheon, J.H.: An identity-based signature from gap Diffie-Hellman groups. In: PKC. LNCS, vol. 2139, pp. 18–30. Springer, Berlin (2003)
36. Chow, S.S.M., Yiu, S.M., Hui, L.C.K.: Efficient identity based ring signature. In: Applied Crypto and Network Security—ACNS. LNCS, vol. 3531, pp. 499–512. Springer, Berlin (2005)
37. Condra, G.: pypbc. http://www.gitorious.org/pypbc
38. Cramer, R., Shoup, V.: A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In: CRYPTO, pp. 13–25. Springer, London (1998)
39. Denis, T.S.: LibTomCrypt Project. http://libtom.org
40. Dolev, D., Dwork, C., Naor, M.: Non-malleable cryptography. SIAM J. Comput. 542–552 (2000)
41. Dufour, M.: Shedskin (2009). http://code.google.com/p/shedskin
42. El Gamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Proceedings of Crypto, pp. 10–18 (1984)
43. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: CRYPTO. LNCS, vol. 263, pp. 186–194 (1986)
44. Freeman, D.: Converting pairing-based cryptosystems from composite-order groups to prime-order groups. In: EUROCRYPT, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 44–61 (2010)
45. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC, pp. 169–178. ACM, New York (2009)
46. GNU. The GNU Multiple Precision Arithmetic Library. http://www.gmplib.org
47. Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. J. ACM **38**(3), 690–728 (1991)
48. Groth, J., Sahai, A.: Efficient non-interactive proof systems for bilinear groups. In: EUROCRYPT. LNCS, vol. 4965, pp. 415–432. Springer, Berlin (2008)
49. Henecka, W., Kögl, S., Sadeghi, A.-R., Schneider, T., Wehrenberg, I.: Tasty: tool for automating secure two-party computations. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS, pp. 451–462. ACM, New York (2010)
50. Hess, F.: Efficient identity based signature schemes based on pairings. In: SAC, LNCS 2595, pp. 310–324. Springer, Berlin (2002)
51. Hohenberger, S., Waters, B.: Realizing hash-and-sign signatures under standard assumptions. In: Advances in Cryptology—EUROCRYPT (2009)
52. Hohenberger, S., Waters, B.: Constructing verifiable random functions with large input spaces. In: EUROCRYPT, 29th Annual International Conference on the Theory and Applications of Cryptographic, Techniques, pp. 656–672 (2010)
53. Iovino, V., Persiano, G.: Hidden-vector encryption with groups of prime order. In: Proceedings of the 2nd International Conference on Pairing-Based Cryptography, Pairing '08, pp. 75–88. Springer, Berlin (2008)
54. Lacy, J.B.: CryptoLib: Cryptography in software. USENIX Security Conference IV, pp. 1–18 (1993)
55. Laurie, B., Clifford, B.: The Stupid programming language. Source code available at http://code.google.com/p/stupid-crypto/
56. Lewis, J.R., Martin, B.: CRYPTOL: High Assurance, Retargetable Crypto Development and Validation (2003). http://www.galois.com/files/Cryptol_Whitepaper.pdf
57. Lewko, A., Sahai, A., Waters, B.: Revocation systems with very small private keys. In: Proceedings of the IEEE Symposium on Security and Privacy, SP, pp. 273–285. IEEE Computer Society, Washington, DC (2010)
58. Lewko, A., Waters, B.: Decentralizing attribute-based encryption. In: Patterson, K.G. (ed.) EUROCRYPT. LNCS, vol. 6632, pp. 568–588. Springer, Berlin. http://eprint.iacr.org/
59. Lewko, A.B.: Tools for simulating features of composite order bilinear groups in the prime order setting. IACR Cryptol. ePrint Archive **2011**, 490 (2011)
60. Litzenberger, D.C.: PyCrypto—The Python Cryptography Toolkit. http://www.dlitz.net/software/pycrypto/
61. Lynn, B.: The Stanford Pairing Based Crypto Library. http://crypto.stanford.edu/pbc
62. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay—a secure two-party computation system. In: Proceedings of the 13th USENIX Security Symposium, pp. 287–302. USENIX Association, Berkeley (2004)
63. Meiklejohn, S., Erway, C.C., Küpçü, A., Hinkle, T., Lysyanskaya, A.: ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In: Proceedings of the 19th USENIX Conference on Security, USENIX Security, pp. 13–13. USENIX Association, Berkeley (2010)
64. Meiklejohn, S., Mowery, K., Checkoway, S., Shacham, H.: The phantom tollbooth: privacy-preserving electronic toll collection in the presence of driver collusion. In: Proceedings of the 20th USENIX conference on Security, SEC, pp. 32–32. USENIX Association, Berkeley (2011)

65. NIST.: Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186 (1994)
66. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO. LNCS, vol. 576, pp. 129–140 (1992)
67. Regev, O.: Lattice-based cryptography. In: Dwork, C. (ed.) Advances in Cryptology—CRYPTO 2006. Lecture Notes in Computer Science, vol. 4117, pp. 131–141 Springer Berlin Heidelberg (2006).
68. Rouselakis, Y., Waters, B.: New constructions and proof methods for large universe attribute-based encryption. Cryptology ePrint Archive Report 2012/583 (2012) http://eprint.iacr.org/
69. Sahai, A., Waters, B.: Fuzzy identity-based encryption. In: EUROCRYPT, pp. 457–473 (2005)
70. Scott, M.: MIRACL library. Indigo Software. http://indigo.ie/mscott/download
71. Stein, W., et al.: Sage Mathematics Software (Version 5.0.1). The Sage Development Team. http://www.sagemath.org
72. Stern, J., Paillier, P.: Public-Key Cryptosystems Based on Composite Degree Residuosity Classes, vol. 1592, pp. 223–238. Springer, Berlin (1999)
73. The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS (2010). http://www.openssl.org
74. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full sha-1. In: Proceedings of Crypto, pp. 17–36. Springer, Berlin (2005)
75. Wang, X., Yu, H.: How to break md5 and other hash functions. In: EUROCRYPT. Springer, Berlin (2005)
76. Waters, B.: Efficient identity-based encryption without random oracles. In: EUROCRYPT. LNCS, vol. 3494, pp. 114–127 (2005)
77. 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/
78. Waters, B.: Functional encryption for regular languages. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 218–235. Springer, Berlin (2012)
79. Wustrow, E., Wolchok, S., Goldberg, I., Halderman, J.A.: Telex: Anticensorship in the network infrastructure. In: Proceedings of the 20th USENIX Security Symposium (2011)