SaTC: CORE: Small:
API-Centric Cryptography

Principal Investigator: Thomas Shrimpton (University of Florida)

Solicitation: `http://www.nsf.gov/pubs/2017/nsf17576/nsf17576.htm`

Introduction

For years, the cryptography research community has been imploring developers to "please stop rolling your own crypto!" An expanded version of this might be: cryptography is hard, so let the crypto experts handle the crypto; you (just) worry about implementing what we provide. There is good sense in this message. Cryptography *is* hard. The security theorems that we prove can depend crucially, and subtly, on parameter choices, padding schemes, protocol logic, etc. Our research community has developed a collective wisdom that helps guide us to good solutions (or at least to avoid pitfalls).

To their credit, most software developers do seem to be heeding our plea. But there is a rejoinder that we academic cryptographers have largely missed, or disregarded. To paraphrase Paul Kocher's invited lecture at Crypto'16, and various speakers (e.g., David McGrew (Cisco), Daniel Kahn-Gilmore (ACLU), Thai Duong (Google)) at Real World Crypto: cryptographers must understand that the design of cryptographic libraries, specifically the APIs they export, *is hard too*. Moreover, fielded APIs have long-lifetimes, because changing them wreaks havoc upstream. So if a theoretical primitive it is hard to realize with existing libraries and their APIs, then either it won't be implemented, or (worse) it will be prone to being implemented incorrectly.

Motivated by this reality, our position is that *it is incumbent upon academic cryptographers to accept practical implementation and deployment needs as a guide for developing theory.* Specifically, the research directions that we propose here are founded on our beliefs that:

- Theory is considerably more useful to real-world security when it is mindful of its translation to practice.
- Ease-of-correct-implementation should be treated as a first-class security consideration.
- Cryptographic primitives should aim to be forgiving of misuse.

In support of the first, we suggest an *API-centric* approach to the development of cryptographic theory. By this we mean several things. First, that existing theory — in particular the formal syntax that we have developed to describe cryptographic primitives, and our theoretical realizations of them — should be carefully reexamined with respect to the actual APIs that libraries export. As prior works by PI Shrimpton [NRS14, BPS17] and others (e.g. [FGMP15]) show, mismatches may be common and lead to theory that is less useful than we might think, or even damaging. Second, that new realizations of existing primitives should take into consideration how easy they will be to implement with (at least) broadly adopted libraries. Third, that the formalization of new primitives should include syntax that is clear about the functionalities that will need to be implemented, i.e. syntax that is "API-like". Relatedly, security notions should consider these functionalities, so as to have a more practice-grounded view of the security of primitives. All of the research that we propose here adheres to an agenda of *API-centric cryptography*.

To the second point, ease of correct implementation is not something that is formally considered as part of the provable-security paradigm that underpins modern cryptography. We think this is unfortunate, as security vulnerabilities that result from incorrect implementation are far more common than vulnerabilities from bad theory. In the best case, we would like to see ease-of-correct-implementation quantified and incorporated into security theorems. This is something that we will work towards as part of our research agenda, but in this proposal we will focus on a slightly different tact. In lieu of a direct quantification, we look to a little-known work by Rogaway and Stegers [RS09]. Loosely speaking, they model cryptographic protocols as being comprised of a partially specified protocol (PSP) and protocol details (PD). The PSP is the cryptographic core of the protocol, and it is assumed that this portion of the actual real-world scheme is specified in sufficient detail to be implemented correctly. The PD is the rest of the scheme, i.e., all of the details that are not attended to by the theory.[1] In their modeling, the PD is *adversarially* controlled. Thus, a proof of security provides evidence of robustness to errors in the implementation of the protocol. Rogaway and Stegers apply this interesting approach to entity authentication protocols, but we believe it can much more broadly applied to cryptographic protocols and primitives. Some of the concrete tasks that we propose reflect this.

The third point, that cryptographic primitives should aim to be forgiving of misuse in practice, has recently gained some traction in the symmetric-key crypto community. Intuitively, the idea is to preemptively mitigate the damage resulting from common misuses of cryptographic primitives, by building the misuse into the adversarial model. Concretely misuse-resistance was introduced by Rogaway and PI Shrimpton [RS06] in order to build nonce-based authenticated encryption schemes that do not forfeit all provable-security guarantees when a nonce is repeated. Since then, nonce-misuse-resistance has appeared in a number of papers (e.g., [ABL$^+$14, HRRV15, HKR15]), and as one of

[1] Rogaway and Stegers take aim at protocol standards, but one can just as easily consider academic papers.

the desiderata in the CAESAR competition [CAE]. Recent work by PI Shrimpton [BPS17] looks at hedged public-key encryption, which can be seen as a kind of misuse-resistance where the encryption scheme is used with bad randomness. (This was done from an API-centric perspective, too, as we will discuss.) By interacting with industrial developers, we will seek out other forms of misuse, and push to protect a broad range of cryptographic primitives — and practitioners who wish to use them — against real-world usage errors.

Public artifacts and broader impacts

In addition to publications at venues targeted at expert cryptographers and security researchers, we will produce higher-level and survey articles appropriate for a more general computer science audience. Whenever possible, we will implement the theory that we develop, and make the software open-source. Likewise, any software tools that we develop in support of the tasks outlined here will be made open source. We specifically propose to work closely with industrial contacts in order to develop a better understanding of real-world APIs, real-world patterns of misuse, etc., and we will provide feedback and assistance to industry in return. We specifically propose to examine standards, and we will provide feedback and support to the standards committees and/or their organizing bodies as appropriate. The PI has a track record of not only releasing public, open-source research systems (e.g. LibFTE [FTE], Marionette [Mar]), but also going the extra mile to help incorporate such implementations into production systems, such as Tor and Google's uProxy. (See Section 5 for more information about our track record in this regard.) We will target similar impact for the proposed work

We will also make data sets publicly available by default, the exception being cases in which we have privacy or confidentiality obligations. See the Data Management Plan for more details regarding our handling of data.

PI Qualifications

PI Shrimpton has a proven track record of successful research in security, cryptography, and system development. We have already mentioned some of the relevant work that prepares us to make progress on the the proposed research. For more than a decade, PI Shrimpton has pursued an agenda of making cryptography that is relevant to practice and forgiving of misuse (e.g., [RS06] and the related RFC [Har08], [BRSS10, PRS11, LDJ+14, ST15]), of examining standards [NRS14, SSW16, CNS+17], and more recently of explicitly considering cryptography with respect to real libraries and APIs [BPS17]. PI Shrimpton also has a mature network of colleagues that may benefit this work. Several (e.g., Phillip Rogaway, Kenny Paterson, Thomas Ristenpart, Trevor Perrin) have already expressed interest in collaborating on the research proposed here.

Proposal Organization and Tasks

In the following sections, we will provide background and motivation for the research threads will pursue. Throughout, we identify the concrete tasks we will undertake with the visual call-out **Task**. Our list of tasks serves as a compact outline of what we will deliver, at a minimum. We follow the technical material with a discussion of the broader impact of our research agenda.

API-centric Cryptography

The security community has published extensively on the *usability* of APIs; recent developments in this area include [ABF+17, HHL+17, IKND16]. Intuitively, if an API is easy to use correctly (and hard to use *incorrectly*), developers are more likely to build secure applications on top of it. Our approach is complementary, focusing on the relationships between functionalities that APIs present and the primitives that crypto theory provides. In practice, API design is primarily driven by non-security requirements (e.g. functionality, adoption, deployment). As such, we suggest to take an *API-centric* view of the primitives and protocols that we formalize.

We are not the first to recognize the benefits of this viewpoint. Recent works by Fischlin et al. [FGMP15] and Huang et al. [HRRV15], for example, make clear efforts to mimic actual APIs in their formalizations. Quoting [HRRV15], "To accurately model the underlying goal... we adopt an *API-based view*, where the segmentation of a plaintext is determined by the caller." (Our emphasis.) What we propose is to fully embrace this viewpoint, and to apply it broadly to cryptography.

In the remainder of this section, we discuss some specific areas to which we will apply this API-centric perspective.

Understanding the Landscape. It is important to recognize that there are at least two types of "customers" to consider when taking an API-centric viewpoint on cryptography. The first we have already mentioned, namely *application developers*. Application developers are primarily concerned about functionality, and typically have little to know real

expertise in cryptography. (Nor should they, frankly.) For them, the interface to cryptographic functionalities should be simple and have intuitive semantics, e.g.,

$$\begin{aligned}\text{SecureSocketHandle} &\leftarrow \textbf{SecureOpen}(\text{ReceiverID}),\\ \text{Status} &\leftarrow \textbf{SecureSend}(\text{SecureSocketHandle, Msg}),\\ (\text{Msg}, \text{Status}) &\leftarrow \textbf{SecureReceive}(\text{SecureSocketHandle}),\\ \text{Status} &\leftarrow \textbf{SecureClose}(\text{SecureSocketHandle}).\end{aligned}$$

as an API for securely exchanging messages with an specified partner. The API would also specify the structure of the inputs and outputs. For example, the Msg object may include associated data or attributes, in addition to the actual plaintext.

The above API neatly hides an enormous amount of plumbing implemented by the *security engineer*. There are a number of components to put together. The high-level tasks are to authenticate the peer, exchange a key, use the key to transmit the message, and securely tear down the session. Designing and implementing such a protocol gets into the weeds quite quickly. This is an area where having good APIs for things like digital signatures, message authentication codes, and encryption is extremely helpful; even so, it is rarely apparent how these are to be securely composed. This difficulty is compounded by the fact that the protocol needs to be as efficient as possible, as application developers (and users of the applications in turn) are generally unwilling to pay a heavy price for security. To illustrate these issues, consider the following API for unilaterally[2] authenticated key exchange:

$$\begin{aligned}\text{SocketHandle} &\leftarrow \textbf{Open}(\text{ReceiverAddr}),\\ (\text{Key}, \text{Status}) &\leftarrow \textbf{NegotiateKey}(\text{SocketHandle}, \text{ReciverCert}, \text{CertAuth})\end{aligned}$$

where $(\text{ReceiverAddr}, \text{ReceiverCert})$ comprise ReceiverID (above). First, note that the root-of-trust, CertAuth, is implicit in the first API. Unless we are to trust the developer to manage certificates (as many existing APIs do), then the security engineer needs to make this transparent to upper levels. This has inherent risks, which may not be obvious to the security engineer [BFS12]. Second, there are numerous ways to realize **NegotiateKey**, each with varying degrees of security and efficiency. Third, modern protocols, such as TLS, are designed so that application data may be piggy-backed on the handshake. (This is called 0-RTT, or zero round trip time, data.) As such, providing an opaque interface is likely not desirable. In general, the "right" level of exposure to the application developer or security engineer is highly task-dependent.

Task 1: *We will carry out a detailed survey of existing APIs exposed by cryptographic libraries. The goals will be to understand exactly what are the effective primitives being exported, to build a framework for understanding design patterns common across libraries, to set metrics for comparing the complexity of the primitives exposed, and so on.*

The above will support multiple subsequent tasks. We note that the *effective* cryptographic primitives that are presented to the application developer may be considerably different that those considered cryptographers.

Task 2: *Beginning with application developer APIs for secure exchange of messages and secure read/write from storage, we will formalize the cryptographic primitives they imply, and develop security notions that capture the security properties that developers assume are provided. (See also the related Task 3.)*

Secure Channels. Perhaps the most practically important cryptographic functionality is the secure channel, the task of which is to provide privacy and authenticity for message flows between peers. Over the last decade or so, *authenticated encryption with associated data* (AEAD) has emerged as an important ingredient for constructing secure channels. Its syntax is easy to reason about from a theoretical point of view, and yet is robust enough to encompass most tasks that practitioners encounter. Moreover, notions like *nonce-misuse resistant* [RS06] and *robust* [HKR15] authenticated encryption make AEAD schemes difficult to (inadvertently) use incorrectly. These properties make AEAD an attractive tool for designing protocols.

It is tempting to think that AEAD, on its own, provides a secure channel. But as pointed out by Badertscher et al. [BMM+15], classical security notions, such as those for AEAD "do not capture in which contexts a scheme satisfying them can securely be used." Indeed, bugs in the *protocol logic* have been exploited to circumvent the security provided by the underlying crypto [Vau02, BKN04, APW09].[3] This gap has spurred the study of *stateful* authenticate

[2] By unilaterally, we mean that the party opening the channel authenticates the peer, but not the other way around.

[3] See `https://www.mitls.org/pages/attacks#protocol` for a list of protocol-level attacks on TLS.

encryption schemes that model secure channels more closely, such as the highly influential SSH paper [BKN04] and the analysis of the TLS 1.2 record layer by PI Shrimpton and coauthors [PRS11]. Boldyreva et al. [BDPS12] later extended [BKN04] to encompass the protocol-level attack of [APW09] that exploits fragmented delivery of the ciphertext stream.

Fischlin et al. [FGMP15] observe that this line of work still misses an important point about the APIs (i.e. the effective cryptographic primitives) that are presented in practice. Namely, the prior work treated channels as providing secure transport of atomic messages between two parties. But TCP is ubiquitous for the delivery of web content, and virtually all programming languages export an API for *streaming data* over TCP. Likewise, implementations of the TLS record layer also export a streaming data API. Fischlin et al. [FGMP15] were the first to give a cryptographic treatment of secure channels as data streams. (Prior work [BDPS12, BFK+13] treated plaintexts and/or ciphertexts as atomic.) Their syntax directly captures a streaming API, and their security notions (privacy and authenticity) model an adversary who transmits ciphertext "fragments" over the channel, and may drop, inject, and reorder fragments at will. This allows them to model streaming protocols (such as TLS) that divide the message stream into discrete "records" and encrypt each individually before transmitting. The peer may receive only a piece of a record, or multiple records simultaneously.

We regard [FGMP15] as an important step in an API-centric treatment of secure channels, but a number of important practical issues remain. In order to apply their methodology to a real protocol, one would need to precisely specify it as a streaming secure channel (according to their formalism) and provide a proof of security. This on its own is a complex undertaking (cf. [DLFK+17]), but it also places a heavy burden on security engineers. It is incumbent on them to implement this protocol *precisely*, and if the protocol needs to be changed (say, to address some engineering constraint), then the cryptographer needs to provide a fresh proof.

A largely-overlooked work by Rogaway and Stegers [RS09] affords an opportunity to bridge this gap.[4] Their observation is that when cryptographers design protocols, they provide an abstraction that omits many details relevant to implementing them on real systems. This is not without reason, of course; the needs of systems are complex and always changing. TLS is a prime example. The record layer is the piece of the protocol encompassing all messages between client and server. The specification document for the upcoming version 1.3 [Res17] is comprised of a rich set of rules for how the stream of messages are mapped to records: some types of messages may be coalesced into a single record, others must be the only message in a record, and so on. These rules may seem, at first glance, irrelevant from a security perspective, but it quickly becomes clear that the analysis must address them. Rogaway and Stegers provide a framework for analyzing *partially-specified protocols*, where the "cryptographic core" of the protocol is separated from the "protocol details" relevant to engineers. In the analysis, one assumes the latter is utterly controlled by the adversary. Thus, a proof of security under this pessimistic assumption affords the protocol designer a high degree of flexibility, while providing a certain amount of robustness to bugs in implementations.

Task 3: *We propose to apply Rogaway-Stegers methodology to streaming secure channels towards providing guidance in the design of the TLS 1.3 record layer.*

Authenticated Key Exchange. The key agreement protocol used to establish the channel is also amendable to an API-centric treatment. Since the seminal work of Bellare And Rogaway [BR93], the vast majority of the literature on authenticated key exchange has focused on the case of *mutual* AKE (MAKE), where both client and server wish to authenticate the other. In fact, virtually all TLS handshakes on the Internet are only *unilaterally* authenticated (UAKE). (The client wishes to verify the server's identity.) We alluded to another issue above. An important consideration in the upcoming TLS 1.3 specification [Res17] is the transmission of 0-RTT data. This feature is crucial from an engineering perspective, but as is well-known [Res17, Section 2.2], we cannot provide the same level of security for the message on which we are piggy-backing the exchange as is possible for subsequent messages. In particular, *forward security* is out of reach.

These issues point to a gap between how programmers expect they should be able to use cryptography, and how cryptography is to be used securely. This presents an opportunity to distill from design patterns in existing KE protocols (e.g., the TLS handshake) an abstraction (and corresponding API) that cleanly captures the needs of software engineers. A promising direction is to begin with Dodis and Fiore [DF17], who show that UAKE is much simpler to model than MAKE, which requires a more general protocol framework, such as Bellare-Rogaway. Dodis and Fiore define a natural primitive they call *confirmed encryption* for KEM-based UAKE, which has an API that, we feel, is intuitive for security engineers. Briefly, the sender encrypts its message with the public key of the recipient essentially

[4] Degabreile et al. [DPW11] also note this: "this is an intriguing approach that we expect to see further developed."

as one would with a conventional encryption scheme. The receiver decrypts the ciphertext, then transmits a "confirmation" proving to the sender that the message was received. This primitive immediately yields UAKE. Using an approach by Krawczyk [Kra16], it can also be used as a building block for a MAKE protocol.

Distilling a complex problem in this way is invaluable to practice. A limitation of [DF17], however, is that confirmed encryption cannot provide forward security on its own. As this has become a first-class concern, we feel it deserves a first-class primitive.

Task 4: *Extending the work of [DF17], we will study UAKE protocols that provide* forward security *with the aim of providing an API that is intuitive for security engineers.*

Having such an API would simplify the task of designing real-world KE protocols, but it is only a component of the design process.

Going in a different direction, Rogaway and Stegers [RS09] suggest their methodology can be extended from entity authentication to AKE. Perhaps due to the limited attention this paper has received in the cryptographic community, this natural extension as not been explored.

Task 5: *We will apply the partially-specified-protocol methodology of [RS09] to authenticated key-exchange.*

Standardized APIs. In [SSW16], PI Shrimpton and coauthors gave a provable-security treatment to the design and analysis of so-called "cryptographic APIs", standardized in the PKCS#11 document.[5] That standard details an interface by which one may interact with cryptographic tokens (e.g. smart cards, USB devices, enterprise-grade HSMs) that are trusted to perform key-management duties, enforcement of policies for the use of stored keys, and a limited number of asymmetric- and symmetric-key crypto operations. These tokens support a variety of applications, including entity authentication, certification authorization, SSL/TLS acceleration, and interbank communication.

Our approach in this paper was API-centric, in that we took the PKCS#11 standard as the reference point, and defined from it a new cryptographic-API primitive. We then formalized some of the intuitive notions of security for a cryptographic API. This was a complex undertaking, as one of the core key-management functionalities is to allow the exporting and importing of keys via "key (un)wrapping" operations. (This supports, among other things, the transportation of cryptographically protected keys between tokens.) These keys may have a variety of attributes associated to them that proscribe their usage, e.g. this key may/may not be used to encrypt data (via a given API call), that key may or may not be wrapped for future export, etc. Thus, the internal state of the key-management functionality changes as an attacker carries out permitted API calls. (We note that works such as [KS13] exploit this changing of internal state, sometimes in concert with underspecification in standards and device documentation, to cause catastrophic security breaks in HSMs.) Moreover, a real adversary may be in possession of one or more tokens — keep in mind that these may be physically vulnerable devices, such as USB sticks or smart cards — and these are subject to corruption. As a result, a victim token is asked to import adverasarially-known keys. This can cause many other keys under management by the victim to become (effectively) corrupted, e.g. if the attributes of the imported key allow it to be used for wrapping.

While [SSW16] makes important progress in the analysis of cryptographic APIs (in particular PKCS#11), they do leave open some interesting avenues. For example, their handling of attributes is quite abstract, only indicating their influence on tokens as part of the execution model. But what one really wants are formal mechanisms and security notions for reasoning about the enforcement of specific policies, e.g. that certain keys may be used only by certain entities and only for a given set of cryptographic operations. Also, [SSW16] does not consider the public-key primitives that are typically exposed by tokens. Finally, it would be useful to understand the extent to which real tokens respect PKCS#11 and, if we find the model of [SSW16] to be significantly disconnected from these tokens (say, because tokens often take liberties with PKCS#11, or because other standards are gaining traction) then we should revisit the models.

Task 6: *We will explore these interesting and important extensions to [SSW16]: (1) formal mechanisms and security notions for reasoning about the enforcement of specific policies, (2) incorporating the public-key primitives that are typically exposed by tokens, (3) establishing how closely tokens adhere to PKCS#11 and revisiting the formal models based on what we find.*

[5] See `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11` for the latest spec.

Misuse-Forgiving Primitives

Towards the goal of building cryptography that is forgiving of misuse, prior work by PI Shrimpton defined the notion of nonce-misuse-resistant authenticated encryption [RS06], and showed how to achieve it using standard symmetric-key tools. Rogaway had previously argued [Rog04] that the classical viewpoint that symmetric encryption schemes are randomized primitives was misaligned with practice, and what we should be delivering encryption schemes that are deterministic, surfacing an explicit IV input. (This is an early effort to close the gap between a theoretical primitive and its real-world presentation.) Moreover, to make schemes easier to use correctly, we should target security when the IV is a non-repeating value (i.e., a nonce) rather than demanding the IV be random. In [RS06], we sought to make these nonce-based encryption schemes even easier to use, by designing them so that their security guarantee degrades gracefully when the nonce IV repeats. Nonce-misuse resistance has been recognized as an important goal in practice, and was one of the desiderata of the CAESAR competition [CAE].

We are interested to learn other ways in which cryptographic primitives are prone to being misused in practices.

Task 7: *We will leverage institutional and personal connections to industry (Intel, Comcast, Honeywell, Harris, Cisco, Agilent, BMW, IBM, Juniper, Mentor Graphics, Qualcomm, to name just a few[6]) to learn what things developers don't understand about the primitives that theory provides them (in particular when implementing or using them), and how these primitives are commonly abused (knowingly or unknowingly). We will use the lessons learned from this to formalize new primitives and new notions of misuse-resistant security.*

Scrutinizing standards. Later work by PI Shrimpton [NRS14] reconsidered the traditional wisdom about building authenticated encryption (AE) via generic composition of an encryption scheme and a MAC. The seminal work by Bellare and Namprempre [BN00] showed that of the three classical compositions — encrypt-and-mac, mac-then-encrypt, encrypt-then-mac — only the last is secure given any secure encryption scheme and MAC. This wisdom was heeded by an ISO standard, which would have been a good thing, except that the ISO standard was mandating a nonce-based AE scheme, whereas the Bellare-Namprempre results were about randomized AE. As a result, the ISO scheme was actually broken, despite the standard's (applaudible) efforts to do what it seemed the crypto community told them. Here again, the mismatch between theoretical primitives and their real-world realization was problematic. In [NRS14], we readdressed generic composition from the nonce-based perspective, finding an interesting (albeit less simple) picture of what compositions are (and are not) generically secure. (Curiously, one of the secure compositions that our work uncovered was SIV-mode, which appeared in [RS06] as the first nonce-misuse-resistant AE scheme.)

Task 8: *Borrowing tools (and expert colleagues) from NLP and data mining, we will examine standards — the full body of IEEE and ISO standards, minimally — to surface those that are likely to contain cryptographic errors.*[7] *We will write a survey/systemization-of-knowledge paper based on our findings, and work with standards bodies to address the mistakes we find.*

Going in a different direction, the traditional syntax for encryption assumes that the plaintext data is a bitstring. For many practical applications this suffices, as structured data can be serialized (or "flattened") prior to application of encryption. But there are important applications in which this won't work. For example, database encryption. Here the data (i.e. the database) is highly structured, and in order to support efficient queries on portions of the (encrypted) data, one cannot simply flatten the data into a single string and apply traditional IND-CPA-secure encryption. Various works have proposed to use combinations of deterministic encryption [BBO07], searchable encryption [CJJ+13], and order-preserving/revealing encryption [LW16]. All such property-revealing encryption schemes leak some information in order to support computations, and typicallly this leakage is formally captured with the attendant security notion. But as recent attacks show [CGPR15, NKW15, GSB+17], capturing what is leaked does not necessarily say anything about how damaging that leakage is, in practice.

We see this as an example of seemingly good theory unknowingly "setting up" developers to misuse it. What is needed is a proper cryptographic primitive for handling highly structured data — not just databases, but XML and JSON documents, sequences of plaintext fragments and their associated headers/metadata, filesystem encryption, and the like — and a framework for reasoning about security. Such a framework will need to capture not only what is leaked by the scheme, but also what types of malleability the scheme allows. For the latter, we mean ciphertexts that the adversary *should* be able to create given a set of observed ciphertexts. (Think: ciphertexts that decrypt to a

[6] See `http://fics.institute.ufl.edu/sponsors/` for a more complete list.

[7] Very recent work by PI Shrimpton and coauthors finds many errors in an important IEEE standard for protecting electronic intellectual property; this will appear at ACM CCS '17.

reordering of rows or columns in a table, ciphertexts of trees that are missing a subtree. For traditional encryption, the adversary should only be able to "create" replays.) Having explicit malleability models will allow us to reason about active attacks, which have often plagued efforts to develop database encryption schemes [GRS17].

Task 9: *We will develop a framework of security notions for primitives that encrypt structured plaintexts. These notions will capture both what is leaked by ciphertexts, and also what explicit malleability models. We will establish new primitives for encrypting structured data, with support for structural metadata and plaintext associated-data.*

Hedged Cryptography. Recent work by PI Shrimpton and coauthors [BPS17] revisits the theory of hedged public-key encryption (PKE) [BBN+09] from an API-centric perspective. The motivation for hedged PKE is quite practical. Traditional PKE schemes are proved secure under the assumption that they have access to a source of uniform random bits; but in practice, PKE schemes are implemented on systems that have faulty RNGs, or where entropy is difficult to harvest. In these cases, the security guarantees proved in theory are, at best, voided; at worst, failure of the RNG can lead to recovery of the plaintext. Hedged PKE is designed to satisfy traditional notions of security when provided uniform randomness, and still deliver useful security properties as the quality of randomness degrades. The most elegant construction of hedged PKE works like this [BBN+09, BH15]: synthesize fresh random bits by hashing all of the encryption inputs (the public key, the message, the provided randomness), and then use these bits as the randomness for an underlying PKE scheme. In practice, implementing this simple construction is surprisingly difficult, as the high- and mid-level APIs presented by the most commonly used crypto libraries (e.g. OpenSSL and significant forks thereof) *do not* permit one to specify the per-encryption randomness.[8] In [BPS17], the theory of hedged PKE is reconsidered from the perspective of what can be easily constructed given what real APIs expose, and what provable security guarantees these can achieve.

Still, [BPS17] leaves open some important matters. First of all, the randomness sources in their models are stateless. Most crypto libraries export *pseudorandom number generators with inputs* [BH05, DPR+13, ST15] for providing random bits to library code and applications. These allow the programmer to add entropy into the PRNG's internal state. Additionally, [BPS17] only considers PKE.

Task 10: *We will push hedged PKE closer to practice by exploring the consequences of sources that are PRNGs with input. Minimally, this will require a reconsideration of security notions and new constructions. We will implement the best schemes to evaluate performance (e.g. overhead relative to plain PKE and the schemes from [BPS17]) and interact with library maintainers to facilitate deployment of our constructions.*

Task 11: *We will work to hedge other primitives against faulty randomness, using appropriate models of the randomness sources, all from the viewpoint of what real-world APIs expose.*

Implementation-considerate Formalisms

The modern "provable security" approach to cryptography loosely follows a four-step recipe: define a precise syntax for the primitive under study, define a formal notion of security for that primitive, realize the primitive, and prove it meets the security notion. The first step, defining the syntax, essentially defines the component objects that collectively make up the primitive, e.g. "An encryption scheme Π is a triple of algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ ..." The syntax typically describes the number and type of the inputs and outputs of these components, too, e.g. "The decryption algorithm provides a mapping $\mathcal{D}\colon \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$ ...", as well as requirements on the behavior of these components in any correct realization of them.

The syntax for a new primitive provides an API for researchers. For those wishing to *realize* the primitive (in theory), it makes clear what are the functionalities that must be instantiated, and what are the assertions that must hold for any realization; for those wishing to *use* the primitive, it makes clear what calls will be available to them, and what are the expected inputs and outputs of those calls. Like a real software API, syntax is considered good if it serves the needs of its "customers"; in this case, if it supports clear security notions, facilitates the writing (and verification) of proofs, and is easily built upon.[9]

[8] Other constructions of hedged PKE that do not explicit require the ability to manipulate the encryption coins are, likewise, difficult or impossible to implement via existing high- and mid-level APIs. In their cases, it is because the primitives that are needed do not exist in common libraries.

[9] As an aside, PI Shrimpton has found the analogy of *syntax-as-API* as a useful pedagogical tool. Computer science students often seem far more familiar and comfortable with programming abstractions than with mathematical ones.

DECRYPT(**var** ChannelContext, Msg):

(SentContext, Ciphertext) ← **Deserialize**(Msg)

(ExternalVals, InternalVals) ← **Decrypt**(**var** ChannelContext, SentContext, Ciphertext)

Verdict ← **IsValid**(InternalVals)

ExternalVals ← **ErrHandler**(**var** ChannelContext, Verdict, ExternalVals)

Return ExternalVals

Figure 1: Describing symmetric-key decryption of received message Msg in terms of explicit functionalities that must be realized. All variables are passed by value except those preceded by the keyword **var**. The ChannelContext is receiver-maintained information about the channel, and may contain values shared with the sender (e.g., the key, channel ID), along with local state. The SentContext is context data associated to this particular ciphertext. The ExternalVals include the message and are intended to be released to the external caller, and the InternalVals are intended for use only within the decryption process boundary. The types of all input and output values are implicit.

Of course, there is a considerable gap between the formal syntax that papers provide and a *real* API, i.e., the functionalities that need to be implemented. For example, the well-established syntax for symmetric-key decryption with associated data is a deterministic algorithm $\mathcal{D}\colon \mathcal{K} \times \mathcal{H} \times \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$, where $\mathcal{K}$ and $\mathcal{H}$ are sets, the first finite and the second possibly infinite. We cryptographers would typically write something like $M \leftarrow \mathcal{D}_K^H(C)$ for the decryption of ciphertext $C \in \{0,1\}^*$, with associated data $H \in \mathcal{H}$, under key $K \in \mathcal{K}$. But this quietly assumes a number of implicit functionalities that must be attended to in practice.

First of all, the formal syntax assumes that H and C are simply presented as inputs. In reality the decryption process may need to parse (aka deserialize, unmarshall) what is received from the channel into the ciphertext C and any context information (aka associated data) that was sent with C. (For example, the TLS ciphertext stream encodes the length of each encrypted record.) Moreover, the overall decryption process may depend on both the context that is associated to this ciphertext transmission *and* on context that the receiver maintains for the channel. (Note that the syntax is silent as to the source of the associated data; it may have been transmitted with C or made available locally.) In practice the receiver may maintain local message counters, lists of nonces already observed on the channel, information about the overall status and health of the channel (e.g. is it still considered active), and so on. The decryption key K is also part of the context that the receiver maintains.

Next, the formal syntax says that decryption returns a string or a distinguished symbol $\perp$, the latter being traditionally understood by cryptographers to mean "invalid". In practice, this means that some logic must be implemented to decide (in)validity, and that when the decision is "invalid" some error-handling mechanism ensures that the decryption process returns nothing but a distinguished error message.

In Figure 1, we give an example of a more API-like presentation of decryption. In fact, what this gives a framework for specifying a wide variety of schemes. We will specify two schemes within this framework shortly.

Making these implied functionalities more explicit has several benefits. The most obvious is helping developers to see what they need to implement, rather than assuming they will be able to correctly tease out what's needed. Conversely, it prompts the theoretician to think in terms of abstractions that appear in practice. For both developer and theoretician, this style of presentation can surface a clearer picture of what security demands of these functionalities. For example, specifying error handling as an explicit functionality[10]

ExternalVals ← **ErrHandler**(**var** ChannelContext, Verdict, ExternalVals)

more naturally prompts one to consider questions, like, what ExternalVals should be returned for a given Verdict, and what security-relevant information might be leaked by them? How should the ChannelContext be updated for a given Verdict?

In Figure 2 we give two potential realizations of the functions needed for decryption. In particular, they specify the components of Figure 1 used to realize **DECRYPT**.

[10]Following Rogaway and Stegers [RS09], variables are passed by value except where annotated by **var**. In this case they are passed by reference so that their value may be modified by the caller.

On the left side of the figure, we give a realization of conventional encrypt-then-MAC (EtM) AEAD, following what we theoreticians consider as best practice. Namely, if the tag check fails, then one should label the ciphertext as invalid, and suppress all output other than the single error message "Invalid". Additionally, when "Invalid" is returned, the channel status field of ChannelContext is changed to "closed".

On the right side, we give a more complex (and arguably realistic) EtM-style realization that includes in the ChannelContext a list of valid tokens that are used within the decryption process. It also allows for a stream ID (sid) as part of the transmitted AD (i.e. the SentContext). In the absence of any decryption errors, the decryption process will return (as the output of **ErrHandler**) the stream ID and the plaintext. If the tag check fails, then the Verdict that is returned by **IsValid** is either "Invalid", or $\langle$"Invalid","BadToken"$\rangle$, depending on whether or not the token ID (tid) recovered within **Decrypt** is in the token list. In either case, the decryption process returns "Invalid" and the channel status field of ChannelContext is changed to "closed", as on the left. If the tag check *succeeds* but the recovered tid is invalid, then the decryption process still returns "Invalid", but the ChannelContext is not changed. One can imagine that this might be a real-world behavior, where a ciphertext-validity failure is fatal to the channel, but a bad token is considered less of an infraction. Say, if the cost to tear down/set up a new channel is considered unreasonable for the given application. In the latter case, perhaps the application requests retransmission, or sends an *application layer* error message back through the channel. We note that by making the specification of the functionalities explicit, we clearly surface at least two possible attack models. If the adversary sees and can access the channel only, then only the ExternalVals are observed; whereas if the adversary sees things at the boundary of the decryption process, both the ExternalVals and portions of the ChannelContext may be visible.

Task 12: *We will reconsider the traditional formalisms for cryptographic primitive from an implementation considerate perspective, and establish security notions with respect to the new syntax.*

There are, of course, other ways to handle decryption errors than described in Figure 2; the "right way" is highly dependent on the setting in which the scheme is used. However, its API (Figure 1) offers a feature that is useful in all real-world applications: it explicitly separates the *message* and *error* in the output. Indeed, virtually all APIs that we are aware of separate these. Errors have varying degrees of severity: some are *fatal*, requiring the connection be torn down; some are simply *warnings* that leave to the application the decision to process the message or to tear down the connection; and some might be completely benign, indicating, e.g., that more data is available for reading, or that the end-of-file has been reached.

We are aware of no work that explicitly separates the message and error in this manner. Some works consider a richer semantics for the error [BDPS14, FGMP15], but even these couple the message and error so that one of these is output, but never both. This suggest another secure-channels task.

Task 13: *In our study of secure channels (Task 3) we will explore the implications of separating the message and error in the output of decryption.*

Given ChannelContext=
  $\langle$ModeKey K, PRFKey L, ChannelStatus "open"$\rangle$

Alg. **Deserialize**(Msg):
  $V \leftarrow$ Msg.V; $H \leftarrow$ Msg.H; $C \leftarrow$ Msg.C
  Return $((V,H),C)$

Alg. **Decrypt**(**var** ChannelContext,(V,H), C):
  Parse C into tag T and remainder Z
  if $F_L(V,A,Z) \neq T$ then InternalVals $\leftarrow$ "TagFail"
  ExternalVals $\leftarrow \mathcal{D}_K^V(Z)$
  Return (ExternalVals,InternalVals)

Alg. **IsValid**(InternalVals):
  Verdict $\leftarrow$ "Valid"
  if InternalVals = "TagFail" then Verdict $\leftarrow$ "Invalid"
  Return Verdict

Alg. **ErrHandler**(**var** ChannelContext,Verdict,ExternalVals):
  if Verdict = "Invalid" then
    ExternalVals $\leftarrow$ "Invalid"
    ChannelContext.ChannelStatus $\leftarrow \langle$Status "closed"$\rangle$
  Return ExternalVals

Given ChannelContext=
  $\langle$ModeKey K, PRFKey L, ChannelStatus "open",
    TokenList $TL\rangle$

Alg. **Deserialize**(Msg):
  $V \leftarrow$ Msg.V; $H \leftarrow$ Msg.H; $C \leftarrow$ Msg.C
  sid $\leftarrow H$.StreamID
  Return $((V,H,\text{sid}),C)$

Alg. **Decrypt**(**var** ChannelContext,(V,H,sid), C):
  Parse C into tag T and remainder Z
  if $F_L(V,H,Z) \neq T$ then InternalVals $\leftarrow$ "TagFail"
  tid $\|\, M \leftarrow \mathcal{D}_K^V(Z)$
  if tid $\notin TL$ then
    InternalVals $\leftarrow \langle$InternalVals, "BadToken", tid$\rangle$
  ExternalVals $\leftarrow \langle$sid, $M\rangle$
  Return (ExternalVals,InternalVals)

Alg. **IsValid**(InternalVals):
  Verdict $\leftarrow$ "Valid"
  if "TagFail" $\in$ InternalVals then
    Verdict $\leftarrow$ "Invalid"
  if "BadToken" $\in$ InternalVals then
    Verdict $\leftarrow \langle$Verdict, "BadToken"$\rangle$
  Return Verdict

Alg. **ErrHandler**(**var** ChannelContext,Verdict,ExternalVals):
  if "Invalid" $\in$ Verdict then
    ExternalVals $\leftarrow$ "Invalid"
    ChannelContext.ChannelStatus $\leftarrow \langle$Status "closed"$\rangle$
  else if "BadToken" $\in$ Verdict then
    ExternalVals $\leftarrow$ "Invalid"
  Return ExternalVals

Figure 2: Specifying the functionalities surfaced by our decryption API. **Left:** Encrypt-then-MAC decryption over IV-based symmetric-key decryption $\mathcal{D}$ and vector-input PRF F. A tag-check failure results in the suppression of all plaintext/errors output by $\mathcal{D}_K^V(Z)$, and causes a single error message to be exposed. Also, in the event of a tag-check failure, the error-handling algorithm updates the ChannelStatus to "closed". Note that this specification carries out decryption $\mathcal{D}_K^V(Z)$ even if there is a tag-check failure, thereby hiding a potential timing side-channel. **Right:** EtM-like decryption, with support for valid-token checking and multiple error-handling behaviors.

Curriculum Development Activities, Outreach, and Result Dissemination

Curriculum development. A key aspect of our work will be in improving curriculum to ensure students have the right skill sets to tackle real-world problems using techniques from applied cryptography.

PI Shrimpton will develop a course in cryptography suitable for undergraduates. This course will focus less on proofs and formalisms than his graduate course, and more on applications and developing a good sense of cryptographic hygiene. Development of this course is partially motivated by local industry needs. He will also update his graduate cryptography course to explicitly consider modern crypto libraries and APIs, and to compare the primitives they expose to the existing formal syntax that is traditionally developed. New assignments and projects will be developed that force students to interact with real-world crypto artifacts and standards.

Outreach and diversity. We believe that scientific communities are most productive when they include researchers from a wide variety of background, since science disproportionately benefits from a diversity of viewpoints. Towards this end, we will make an effort when attracting students to especially target women and underrepresented minorities. We have already had success in this regard. Shrimpton has mentored Mrs. Tashell Kelly (undergraduate), Ms. Morgan Miller (MS 2010), Ms. Erin Chapman (MS 2012); and served as a committee member for Mrs. Nichole Schimanski (PhD, 2014). He continues actively recruit female graduate students, and plans that at least one of the two budgeted positions will go to a woman.

PI Shrimpton is working with local academic and industry colleagues to reach potential CS students earlier in the educational pipeline. His previous institution (Portland State) hosted cybersecurity summer camps, part of a program sponsored by the Department of Homeland Security. At the University of Florida, the security institute runs day long visits for local high-school students. Shrimpton and colleagues are currently developing plans for an outreach center that fosters relationships with local K-12 educators.

Shrimpton's current graduate students are Christopher Patton (PhD, expected 2020) and Animesh Chhotaray (PhD, expected 2020). His previous graduates are working Google and Qualcomm Research. The Florida Institute for Cybersecurity, to which he belongs, has recently placed graduates at UIUC, NC State, University of Alabama, and a number of prominent companies.

Interaction with industry. Our research, if successful, will improve security in a number of critical contexts. To facilitate that, we maintain an active network of industry contacts, which, among other things, helps us see and navigate around potential deployment roadblocks. PI Shrimpton has relationships with researchers and developers within Intel, Qualcomm, Google, HP, and Cisco. He has been developing relationships within Harris and Honeywell, and has access to a broad array of industrial partners and governmental bodies through the Florida Institute for Cybersecurity.

Technological impact and software dissemination. This project will have its success measured in large part by the degree to which the research impacts real-world artifacts. PI Shrimpton has a track record in this respect: FTE now ships as part of the Tor browser bundle and is integrated with Google Idea's uProxy product. His work on tweakable ciphers is now part of products sold by Voltage Security.

To facilitate technology transfer and impact, we will make public and open-source the software prototypes that result from the proposed research. We are strong believers in the idea that making available software produced in the course of publicly-funded research accelerates scientific advancement, technology transfer, and education. We have a track record of doing this as well: our software implementations for FTE are open source (see `http://fteproxy.org` and `https://libfte.org`).

Developing the scientific community. An important part of our longer term work is development of the applied cryptography research community, which requires integrating better disparate disciplines within computer science. We particularly target expanding the interaction between those building and deploying systems and the cryptographic theory research communities. PI Shrimpton is on the steering committee of the Real World Cryptography workshop, now in its fifth year; we intend to continue over the lifetime of this grant. This workshop brings together practitioners and academics to hear about the latest applied cryptography research as well as industry problems, and is already popular venue (with some 400 attendees and both practitioners and academics). We believe it is strengthening the sometimes fractious community of cryptography researchers who (want to) do more applied work.

Results of Prior NSF Support

Shrimpton is the PI of NSF grant #1319061 "Tweakable-blockcipher-based Cryptography", for $433,000 and with period 06/2013–07/2016. He is also a co-PI of NSF grant #1564444 "Distribution Sensitive Cryptography", for $399,833 and with period 09/2015–08/2019. There is no substantive overlap in technical content of these proposals and the current proposal.

Intellectual Merit. These awards have resulted in a number of results across security and cryptography, including setting theoretical foundations for hashing, and understanding how to build and analyze random number generators (RNGs). The grants have funded work that resulted in many top-tier publications, including [DRS09, BCS09, ÖSS10, BRSS10, FLR+10, RSS11, PRS11, DCRS12, LST12, ST13, DCRS13, LDJ+14, LSRJ14], and two best-paper awards.

Broader Impact. Prior work by PI Shrimpton has impacted a number of diverse areas. His line of work on hash functions influenced the designs of several of the NIST SHA-3 entrants [DRS09, BCS09, BRSS10, RSS11]. Recent work uncovering significant cryptographic flaws in the IEEE P1735 standard [CNS+17] resulted in a US-CERT vulnerability advisory. Earlier work [NRS14] uncovered a security critical error in the ISO 19772 standard. PI Shrimpton's work on format-transforming encryption FTE [DCRS13, LDJ+14, LSRJ14] now ships with the Tor browser bundle and Google's uProxy. His work on tweakable ciphers [LST12, ST13] has been incorporated into Voltage Security's (now part of HP) product line.