Modular Code-Based Cryptographic Verification

Cédric Fournet, Markulf Kohlweiss, Pierre-Yves Strub

Modular Code-Based Cryptographic Verification

Cédric Fournet
Microsoft Research
fournet@microsoft.com

Markulf Kohlweiss
Microsoft Research
markulf@microsoft.com

Pierre-Yves Strub
MSR-INRIA Joint Centre
pierre-yves@strub.nu

ABSTRACT

Type systems are effective tools for verifying the security of cryptographic programs. They provide automation, modularity and scalability, and have been applied to large security protocols. However, they traditionally rely on abstract assumptions on the underlying cryptographic primitives, expressed in symbolic models. Cryptographers usually reason on security assumptions using lower level, computational models that precisely account for the complexity and success probability of attacks. These models are more realistic, but they are harder to formalize and automate.

We present the first modular automated program verification method based on standard cryptographic assumptions. We show how to verify ideal functionalities and protocols written in ML by typing them against new cryptographic interfaces using F7, a refinement type checker coupled with an SMT-solver. We develop a probabilistic core calculus for F7 and formalize its type safety in COQ.

We build typed module and interfaces for MACs, signatures, and encryptions, and establish their authenticity and secrecy properties. We relate their ideal functionalities and concrete implementations, using game-based program transformations behind typed interfaces. We illustrate our method on a series of protocol implementations.

Categories and Subject Descriptors

K.6.m [**Security and Protection**]: Security; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification techniques.

General Terms

Security, Verification, Languages.

Keywords

Cryptography, refinement types, security protocols.

1. TYPECHECKING CRYPTOGRAPHY

One long-standing challenge for reliably building secure software is to integrate cryptographic expertise with general-purpose program verification.

Several research tools provide automated security analysis of code that uses cryptography; they can deal with sample protocol implementations written in C [32, 23, 28] or in ML [17, 16, 18, 19]. These tools can verify complex properties and reveal logical flaws in protocol designs and implementations. On the other hand, they rely on strong, symbolic assumptions on the underlying cryptographic primitives, treated as black boxes, as initially proposed by Dolev and Yao [27]. They formally restrict the power of the attacker, and are often at odds with weaker guarantees considered by the cryptographers who actually design and implement these primitives. In symbolic models, for instance, each primitive constructs syntactically-distinct symbolic terms, each subject to distinct rewrite rules, which makes it difficult to reflect attacks where the same bitstring may be successfully decrypted *and* parsed using different keys and message formats. Similarly, brute-force attacks on concrete bitstrings used as keys or passwords, or side-channel attacks on binary formats are difficult to reflect symbolically.

Formal Computational Cryptography The semantics of computational models are more complex, as they must account for adversaries with bounded computational capabilities and for small probabilities of failure, even with correct primitives and protocols. More fundamentally, computational models reason about *partial specifications* of cryptographic primitives. Instead of giving, for instance, rewrite rules that completely define their behavior, cryptographers provide minimal positive *functional guarantees*: what the protocol requires to run as intended, e.g. decryption of a correctly encrypted plaintext with the correct key will succeed; and minimal negative *security assumptions*: what a computationally-bounded adversary cannot achieve, except with a negligible probability, e.g. distinguish between encryptions of two different plaintexts. Thus, computational proofs must apply parametrically, for any concrete primitives that meet those functional and security hypotheses.

Direct vs Indirect Approaches The relation between symbolic and computational models for cryptography has been an active subject of research for the last ten years [see e.g. 39, 3, 5]. Despite impressive results, computational soundness holds only for a fraction of the primitives and patterns supported by symbolic verification tools; it also restricts their usage (for instance demanding tags to disambiguate parsing) and requires non-standard assumptions [e.g. type-*n* security, 3]. Thus, to tackle the details of existing protocol designs and implementations, with their own choice of cryptographic primitives and properties, one usually cannot first verify them symbolically then apply a generic computational soundness theorem. As advocated by Halevi [33], Datta et al. [25], Blanchet [20], Barthe et al. [9, 10], we follow a more direct approach: we design and adapt automated verification techniques for computational cryptography, rather than hypothetical symbolic abstractions.

Typing Cryptographic Primitives Types naturally capture partial cryptographic specifications. In the context of communications security, protocol code can be typechecked against abstract interfaces for their primitives, then linked to any implementation that meets this interface, with guaranteed security for the resulting executable. For example, Laud [40, 41] adapt type systems for symbolic security and establish their computational soundness with relatively minor changes to the typing rules.

Symbolic Typechecking with F7 (Review) Bengtson et al. [15] develop a general-purpose refinement type system and a prototype typechecker (F7) for verifying the code of protocols and applications written in F#, a dialect of ML for the .NET platform. Bhargavan et al. [19] extend their method by designing symbolic libraries for a wide range of cryptographic primitives and patterns and evaluating the performance of typechecking versus global verification by ProVerif, a leading protocol verifier [2]. Their approach relies on dual implementations for cryptography: a *concrete implementation* links to systems libraries and is used for running the protocol; another *symbolic implementation* defines cryptography in terms of programming-language abstractions such as seals [42] and algebraic data types. Formally, security holds only within the symbolic model. Following Dolev and Yao [27], concrete security also depends on a careful assessment of the presumed capabilities of adversaries versus those enabled by the symbolic libraries.

Cryptographic Verification with F7 In this paper, we introduce a modular method for programming and verifying cryptographic systems. Using F# and F7, we build computationally sound libraries for sample functionalities, and we use them to program and verify sample protocols. Our method is as follows: for each cryptographic functionality,

- we write a new refinement-typed *ideal interface* that captures its security guarantees;
- we program and typecheck an *ideal implementation* for this functionality, thereby verifying its security;
- we show how to substitute concrete cryptographic implementations for the ideal one, while preserving its security for all well-typed programs that use the functionality, under standard assumptions coded as games in ML.

We then automatically verify programs and protocols that use these functionalities by typechecking against their ideal interfaces. Although these interfaces differ from those used for symbolic verification, the F7 verification framework still applies; we can in particular re-typecheck existing code against our new interfaces, and thus verify their security in the computational model without modifying their specification. Our contributions are as follows:

(1) We adopt the "code-based game-playing" approach advocated by Bellare and Rogaway [13] for writing precise cryptographic proofs: we reason on cryptography using ML programs (rather than Turing machines) and conduct our security proofs by programming, typechecking, and reasoning on ML code (§2 to §6).

(2) We develop a probabilistic variant of RCF, the core typed calculus for F7. We have formalized our calculus and most of its metatheory in Coq [46], including all the key typing lemmas such as type safety and parametricity (§2).

(3) We program typed ideal functionalities for MACs and signatures, and establish their soundness for authenticity, assuming CMA security (§3).

(4) We define a notion of perfect secrecy and show how to automatically verify it by typing using parametricity (§4).

(5) We program parametrically typed ideal functionalities for CPA and CCA2 secure encryption, and establish that they are indistinguishable from concrete encryption (§5).

(6) We evaluate our method on cryptographic constructions, such as hybrid encryption and encrypt-then-MAC, and security protocols, including large implementations previously verified using symbolic models. To our knowledge, these are the largest protocols verified in the standard model (§6).

Our approach is modular, since the existence of ideal interfaces and implementations can be established separately for each cryptographic functionality, and since their composition preserves security. (Symbolic verification lacks this form of modularity, as it relies on a single definition of symbolic terms.) Our ideal implementations correspond to ideal functionalities in the universal composability model [22]; as required, concrete and ideal implementations are indistinguishable for all well-typed environments. Similarly, our use of refinement typed interfaces for ideal functionalities may be seen as a form of conditional simulatability [6].

The code fragments and interfaces listed in this paper are part of typed, executable programs. The source files, the formal Coq development, and a companion paper with additional details and proofs are available online.

Related Work We discuss only closely related work on cryptographic type systems, computational soundness for symbolic cryptography, and verification tools for computational cryptography. As regards perfect secrecy, our definitions and proof techniques rely on parametricity, a classic approach for symbolic security [see e.g. 42, 1, 21].

We adapt an existing type system and its implementation, but do not attempt to design a new one for reasoning about probability distributions or polynomial-time complexity. Many type systems have been developed specifically for computational cryptography, notably in the context of information flow security [e.g. 40, 41]. Beyond type systems, other verification techniques also rely on logical properties of protocols, using domain-specific axioms [29, 26].

CertiCrypt [9] and EasyCrypt [10] provide a framework and a tool to derive Coq proofs for cryptographic constructions. CryptoVerif [20] can verify the security of protocols under asymptotic assumptions, both expressed in a domain-specific pi calculus. It searches for a sequence of games, until it reaches a 'final game' such that the target security properties can be checked. Bhargavan et al. [16] build a model extractor from ML to CryptoVerif, and apply it to verify the cryptographic core of their TLS implementation. Their tools rely on global program transformations that do not scale well to large programs. Our approach is scalable and modular, but does require type annotations.

Our work can be seen as a typed variant of simulation-based security [22, 5, 37]. Using types to express constraints on environments, we achieve fine-grained modularity, giving for example an ideal functionality for CPA encryption.

Backes et al. [8] establish the asymptotic soundness of symbolic trace properties in RCF, the core calculus of F7, by translation to CoSP [7]. They do not rely on types; instead, they relate the seal-based symbolic cryptography of Bengtson et al. [15] to an algebraic Dolev-Yao model, then apply a general computational-soundness theorem. Our method seems more direct and compositional, and also applies to observational equivalences, such as indistinguishability.

2. COMPUTATIONAL RCF

We define a variant of RCF, with mutable references and probabilistic sampling, but without non-determinism: we need sampling to model cryptographic algorithms and probabilistic adversaries; we exclude non-determinism because it would yield too much power to computationally bounded adversaries, essentially giving them access to an NP oracle. Instead, as usual with concrete cryptography, we let the adversary schedule the computation. The resulting calculus has the same expressive power and complexity as probabilistic Turing machines, while retaining the high-level programming syntax and tools of ML. We stay as close as possible to the calculus of Bengtson et al. [15]. We adapt their type system and use F7, their typechecker, for all our experiments.

Core Syntax We give below our syntax for RCF, a formal core of F# with support for sampling and references. This syntax is a simplification of the concrete ML syntax supported by F# and F7; the technical report provides encodings and syntactic sugar showing how to support the ML code presented in the next sections.

Syntax of Values and Expressions

a, b, c	label
x, y, z	variable
ϕ	first-order logic formulas
$h ::=$	value constructor
inl	left constructor of sum type
inr	right constructor of sum type
fold	constructor of recursive type
$M, N ::=$	value
x	variable
$()$	unit
fun $x \to A$	function (scope of x is A)
(M, N)	pair
$h\ M$	construction
$\mathbf{read}_a$	reference reader
$\mathbf{write}_a$	reference writer
$A, B ::=$	expression
M	value
$M\ N$	application
let $x = A$ **in** B	let (scope of x is B)
let $(x, y) = M$ **in** A	pair split (scope of x, y is A)
match M **with**	constructor match
$h\ x \to A$ **else** B	(scope of x is A)
assume ϕ	assumption of formula ϕ
assert ϕ	assertion of formula ϕ
sample	fair coin toss
ref M	reference creation
true $\stackrel{\triangle}{=}$ *inl* $()$ **false** $\stackrel{\triangle}{=}$ *inr* $()$	

The syntax includes standard functional constructors, values, and expressions. It also has two expressions for specification purposes: **assume** and **assert** both take a formula ϕ as parameter. Informally, **assume** ϕ records ϕ as a new hypothesis, whereas **assert** ϕ records ϕ as a proof obligation, which should logically follow from the hypotheses recorded so far. Operationally, these expressions do not affect the rest of the computation. The formulas ϕ range over first-order logic with predicates on F# values; we omit their standard syntax.

Our first addition to RCF is the expression **sample**, which reduces to either **true** or **false** with probability $\frac{1}{2}$. This expression models a perfect source of random bits.

Instead of communications and concurrency, our calculus features references à la ML, via the presence of references creators, readers and writers. We use these imperative references for programming schedulers, communications, and stateful oracles. Formally, references are pairs of functions that read and write on a memory location. Each evaluation of **ref** allocates a fresh label a, and returns a pair $(\mathbf{read}_a, \mathbf{write}_a)$ for resp. reading and writing at location a. Following the syntax of ML, we write $!M$ and $M{:=}N$ for reference reading and writing, defined as

$$\begin{aligned} \mathit{get}\ c &\stackrel{\triangle}{=} \mathbf{let}\ (x_1, x_2) = c\ \mathbf{in}\ x_1\ () \\ \mathit{set}\ c\ v &\stackrel{\triangle}{=} \mathbf{let}\ (x_1, x_2) = c\ \mathbf{in}\ x_2\ v \end{aligned}$$

Conversely, our calculus does not include primitive comparison on arbitrary values ($M = N$), present in RCF. This restriction matters for modelling secrecy in §4, where we need to exclude comparisons on values with abstract types. (Comparisons on constructed values such as Booleans, pairs, and bitstrings are still enabled using **match**, so this restriction does not affect the expressive power of protocols and adversaries that exchange only first-order values.)

Probabilistic semantics We define a probabilistic reduction semantics. We write $[X, L, A] \to_p [X', L', B]$ when expression A reduces in one step to B, from store X to store X', under logical assumptions (i.e. sequence of formulas) L to L', with probability p ($0 < p \leq 1$); this relation is defined in the full paper. Stores are finite maps from labels to values and keep track of references creations and assignments in the usual way. We write $[X, L, A] \to_p^n [X', L', B]$ when $[X, L, A] \to_{p_1} \cdots \to_{p_n} [X', L', B]$ in n steps and p is the product $p_1 \cdots p_n$, and write $[X, L, A] \to_p^* [X', L', B]$ when $[X, L, A] \to_p^n [X', L', B]$ for some $n \geq 0$. Our reduction rules coincide with those of RCF, except for the presence of probability indices, the rules for references and assumption logging:

$$\begin{aligned} [X, L, \mathbf{ref}\ M] &\to_1 [X \uplus \{a \mapsto M\}, L, (\mathbf{read}_a, \mathbf{write}_a)] \\ [X \uplus \{a \mapsto M\}, L, \mathbf{read}_a\ ()] &\to_1 [X \uplus \{a \mapsto M\}, L, M] \\ [X \uplus \{a \mapsto M\}, L, \mathbf{write}_a\ N] &\to_1 [X \uplus \{a \mapsto N\}, L, ()] \\ [X, L, \mathbf{assume}\ \ \phi] &\to_1 [X, L \cup \{\phi\}, ()] \end{aligned}$$

the sampling rules $\mathbf{sample} \to_{\frac{1}{2}} \mathbf{true}$, $\mathbf{sample} \to_{\frac{1}{2}} \mathbf{false}$ and the stuttering rule on values $[X, L, M] \to_1 [X, L, M]$ which guarantees that every closed configuration performs a reduction step with total probability 1. For example, **let** x_0 = **sample in** ... **let** x_{n-1} = **sample in** $(x_0, \ldots, x_{n-1})$ reduces in $2n$ steps to each binary n-word with probability $\frac{1}{2^n}$ and may model a uniform random bitstring generator.

Runtime Safety A close expression A is *safe* if and only if, in all evaluations of A, all assertions succeed, that is, each time an **assert** is evaluated, its formula logically follows from the previously assumed formulas. More generally, we define a probabilistic notion of safety. Let P be a predicate over monotonic trace properties. The probability that $P(A)$ holds in $n \geq 0$ steps is defined as $q_n \stackrel{\triangle}{=} \sum_{A \to_{p_1} \mathbf{A}_1 \ldots \to_{p_n} \mathbf{A}_n | P}(p_1 \ldots p_n)$ where the sum ranges over distinct configurations $\mathbf{A}_1, \ldots, \mathbf{A}_n$ up to relabeling. ($\mathbf{A}_i$ represent configurations with main expression A_i, and an expression A is treated as the configuration with an empty store and an empty log.) The series $(q_n)_{n \geq 0}$ is positive, increasing, and bounded by 1, so it has a limit $q \in [0, 1] = \lim_{n \to \infty} q_n$. We let $\mathsf{Pr}[A \Downarrow M]$ be q when M is a closed value and P is "$\mathbf{A}_n$ has expression M". We similarly define the probability that A *terminates*, letting P be "$\mathbf{A}_n$'s expression is a value", and that A *fails*, letting P be "some $\mathbf{A}_i$ is a failing assert" (i.e. $\mathbf{A}_i = [X_i, L_i, E[\mathbf{assert}\ \ \phi]]$ for some evaluation context E and formula ϕ that does not follow from L_i). Hence, A is safe if and only if it fails with probability 0.

Type Safety We type expressions using dependent types, with the syntax below. The main addition to F# types are *refinement types*: an expression has type $x{:}T\{\phi\}$ when its values M (if any) have type T and, moreover, are such that formula $\phi\{M/x\}$ follows from prior assumptions. Dependent functions and dependent pairs bind a value variable x in the formulas within the types of the function result and the second element of the pair, respectively.

The main typing judgment is of the form $I \vdash A : T$, stating that expression A has type T in environment I. Following Bhargavan et al. [19], we do not rely on kinding (formally, we do not have the subtyping rule (Public Tainted)) and we use an auxiliary typing judgment $I \vdash B \rightsquigarrow I'$ stating that B is a module that is well-typed in environment I and that exports the typed interface I'. Formally, if we have $I \vdash B \rightsquigarrow I'$ and $I' \vdash A : T$, then also $I \vdash B \cdot A : T$ where $B \cdot A$ represents the composition of the F# modules B and A. We refer to the full paper for auxiliary definitions and the RCF typing rules.

Syntax of Types

$T,U,V ::=$	type	
	α	type variable
	unit	unit type
	$x{:}T \to U$	dependent function type (scope of x is U)
	$x{:}T * U$	dependent pair type (scope of x is U)
	$T+U$	disjoint sum type
	$\mu\alpha.T$	recursive type (scope of α is T)
	$x{:}T\{\phi\}$	refinement type (scope of x is ϕ)
bool $\stackrel{\triangle}{=}$ unit + unit	**ref** $T \stackrel{\triangle}{=} (() \to T) * (T \to ())$	

We obtain a safety theorem for probabilistic RCF:

THEOREM 1 (SAFETY). If $\varnothing \vdash A : T$ then A is safe.

Concretely, to establish $I \vdash B \rightsquigarrow I'$, the F7 typechecker takes as input a series of interface definition files (with .fs7 suffixes) instead of typed environments I and I' plus an F# implementation file (with an .fs suffix) instead of the expression B. After typechecking with F7, all formulas can be erased to yield ordinary F# interfaces and implementations.

Security By Typing We use safety to model the security of protocols and their implementations, as follows. The protocol and its libraries are written as F# modules (say C_{PR}), using **assume** and **assert** to specify their security properties, whereas the adversary is their main programs (say A). The capabilities of the adversary are specified as a typed adversary interface I_{PR}: we let A range over arbitrary programs well-typed against this interface ($I_{\mathsf{PR}} \vdash A : T$) that do not use **assume** and **assert**.

Since the **assume** and **assert** specify what is actually verified by typing, one must carefully review them before interpreting formal safety as a meaningful security property. In this paper, we suppose in particular that, for all runs of our systems, the assumptions recorded in the global log are logically consistent, in the sense that we cannot derive false from them. Bhargavan et al. [19] rely on a similar logical-consistency hypothesis and provide simple syntactic conditions on assumed formulas to guarantee consistency. In comparison, our code uses only basic assumptions (e.g. to record runtime events) and their consistency is straightforward.

For protocols, we write the adversary interface I_{PR} so that all its functions operate on plain, unrefined types such as concrete bytes. Thus, our condition $I_{\mathsf{PR}} \vdash A : T$ expresses that the adversary can interact with the protocol only by calling these functions, using any concrete bytes that it can compute, and its formal typability does not exclude any such attack. For instance, I_{PR} may export functions to control the network, generate keys, and trigger runs of the protocol with various parameters, but keep some of the protocol keys private. Thus, by quantifying over all probabilistic runs of all resulting systems $C_{\mathsf{PR}} \cdot A$, we account for the potential behavior of C_{PR} when controlled by any active network attackers. To automatically verify the protocol, we let F7 check that it is a well-typed implementation of the interface I_{PR}, that is, $\varnothing \vdash C_{\mathsf{PR}} \rightsquigarrow I_{\mathsf{PR}}$. Then, by composing the two typing hypotheses on C_{PR} and A, we have $\varnothing \vdash C_{\mathsf{PR}} \cdot A : T$, and by Theorem 1, we conclude that $C_{\mathsf{PR}} \cdot A$ is safe for all active network attackers.

For cryptographic functionalities, the main focus in this paper, we write more precise typed interfaces, using refinement types to express (and typecheck) obligations that protocol designers need to comply with when using these functionalities. Our typed cryptographic interfaces enable us to treat protocols and adversaries as a single program when reasoning on cryptographic libraries. These interfaces may not directly correspond to a class of untyped adversaries. As illustrated for the RPC protocol at the end of §3, they complement more traditional bytes-only adversary interface: given a system composed of cryptographic libraries and protocol code, one can automatically verify by typing that the system exports this particular adversary interface. (See also the full paper for an extended discussion of adversary typability.)

Computational Complexity We finally define notions of termination and complexity. *Asymptotic security* is expressed for algorithms, protocols, and adversaries parameterized by a global *security parameter* $\eta \geq 0$, intuitively the length of the cryptographic materials in use. In this paper, we treat η as a symbolic integer constant in expressions and we often keep its instantiation implicit, writing for instance A instead of $(A_\eta)_{\eta\geq 0}$ for the series of expressions obtained by instantiating η with $0,1,\ldots$.

- A series of probabilities $(q_\eta)_{\eta\geq 0}$ is *negligible* when, for any polynomial p, we have $\lim_{\eta\to\infty}(q_\eta * p(\eta)) = 0$; it is *overwhelming* when $1 - q_\eta$ is negligible.
- A closed expression A is asymptotically safe when its series of probabilities of failing is negligible.
- Closed expressions A^0 and A^1 are asymptotically indistinguishable, written $A^0 \approx_\varepsilon A^1$, when $|\Pr[A^0 \Downarrow M] - \Pr[A^1 \Downarrow M]|$ is negligible for all closed values M.
- A closed expression A has probabilistic polynomial-time complexity, or is p.p.t. for short, when there exists a polynomial p such that, for all $\eta \geq 0$, A_η terminates with probability 1 in at most $p(\eta)$ steps.

We extend our notions of polynomial-time complexity to typed, modular systems with first-order interfaces, that is, interfaces declaring values of base type and n-ary functions on such values. (In our F# programs, we treat curried functions as n-ary functions.)

(1) A closed first-order functional value is p.p.t. when its runtime is bounded by a polynomial in the size of its parameters. In the definition below, we let B range over modules that just bind such values—this ensures that B does not perform any computation on its own, or use any shared state.

(2) An open expression A such that $I \vdash A : T$ is p.p.t. when, for every $\vdash B \rightsquigarrow I$, the expression $B \cdot A$ is p.p.t.

(3) A module F such that $I \vdash F \rightsquigarrow I_F$ is p.p.t. when, for every p.p.t. expression A such that $I_F \vdash A$, the open expression $F \cdot A$ is p.p.t.

The third step is needed to support stateful modules, such as ideal functionalities that allocate a constant-size entry in a log each time they are called. For a given p.p.t. expression A, the size of the log is polynomial, which usually suffices to show that F is polynomial.

For functions on bitstrings, our definitions coincide with those used for Turing machines. Our definition for modules is similar to inexhaustible Turing machines [37] and reactive polynomial-time [34]. We leave as future work a more general treatment of complexity for higher-order modules; see for instance Kapron and Cook [35] for second-order functions, or Danielsson [24] for type-based verification of complexity.

3. AUTHENTICITY USING MACS

Message authentication codes (MACs) provide integrity protection based on keyed cryptographic hashes. We consider the properties of an F# module that implements this functionality. We begin with its plain F# programming interface:

type *key*	**val** *GEN*: unit $\to$ *key*
type *text* = bytes	**val** *MAC*: *key* $\to$ *text* $\to$ *mac*
type *mac* = bytes	**val** *VERIFY*: *key* $\to$ *text* $\to$ *mac* $\to$ bool
val *macsize*: int	**val** *LEAK*: *key* $\to$ bytes
val *keysize*: int	

The interface declares types for keys, texts, and MACs. The type for keys is abstract, whereas those for texts and MACs are just type aliases for 'bytes', the type of concrete byte arrays, used for clarity in other declarations. The interface also declares symbolic constants for the sizes (in bytes) of MACs and keys, and four functions: to generate a fresh key; to produce a MAC given a key and a text; to verify whether a MAC is valid given a key and a text; and to serialize a key into bytes. This function is named *LEAK* since it will be used to model key compromise. (The full paper also discusses a converse function *COERCE* to turn bytes into keys.)

Concrete implementation (*C*) We give below a sample implementation of that interface based on the .NET cryptographic libraries, which we use for running our protocols.

```
open System.Security.Cryptography
let keysize = 16 (∗ 128 bits ∗)
let macsize = 20 (∗ 160 bits ∗)
type key = bytes
let GEN () = randomBytes keysize
let MAC k (t:text) = (new HMACSHA1(k)).ComputeHash t
let VERIFY k t sv = (MAC k t = sv)
let LEAK (k:key) = k
```

(For brevity, we often omit repeated declarations in code excerpts, such as **type** *text* = bytes; the complete source files are available online.) This F# code sets sizes for keys and MACs, uses concrete random bytes as keys, and just performs system calls to standard algorithms [36]. As with all practically deployed symmetric primitives, there is no formal security guarantee and the choice of algorithms is expected to evolve over time.

Ideal Interface (*I*) To capture the intended properties of MACs, we rely on another, refined *ideal interface* for the same module, written *I*, as follows:

```
type mac = b:bytes{Length(b) = macsize}
predicate val Msg: key ∗ text → bool
val GEN: unit → key
val MAC: k:key → t:text{Msg(k,t)} → mac
val VERIFY: k:key → t:text → m:mac → v:bool{v=true ⇒ Msg(k,t)}
val LEAK: k:key{!t. Msg(k,t)} → b:bytes {Length(b) = keysize}
```

This refined interface is designed for protocol verification and is similar to those used for typing symbolic cryptography [19]. It declares the type of keys as abstract, thus preventing accidental key leakage in data sent on a public network or passed to *MAC*.

To support authentication properties, the interface introduces a logical predicate on keys and texts, *Msg*(*k*,*t*), to indicate that *t* is an authentic message MACed with key *k*. This predicate occurs in the types of *MAC* and *VERIFY*, as a pre-condition for MACing and as a post-condition of successful MAC verification. The interpretation of *Msg*(*k*,*t*) is protocol-dependent: as illustrated at the end of this section, each protocol defines *Msg* according to the properties it wishes to authenticate using MACs, possibly giving a different interpretations to each key. In order to be safe for any logical definition of *Msg*, calls to *VERIFY* may succeed at most for texts previously passed as arguments to *MAC* with matching keys (until the key is leaked), thereby excluding the possibility of forging a MAC for any other text.

The pre-condition of *LEAK* accounts for dynamic key compromise: the protocol may call *LEAK* to access the actual key bytes, while other parts of the protocol still rely on the post-condition of *VERIFY*. To preserve safety, the precondition !*t*, *Msg*(*k*,*t*) demands that, before leaking the key, *Msg*(*k*,*t*) holds for all texts. (In F7 syntax, !*t* is logical universal quantification on *t*.) The function *LEAK* can be used to model the potential corruption of principals by leaking their keys to the adversary; see Bhargavan et al. [18, 19] and §6 for protocols verified by typing despite partial key compromise. (The function *COERCE* would have a similar pre-condition, so that any key supplied by the adversary is treated as compromised. Independently, the adversary may include its own copy of the concrete implementation *C*, and thus use any bytes as keys.)

In contrast with symbolic models, our texts and MACs are just concrete byte arrays. Taking advantage of refinements, the interface also enforces the consistency of sizes for MACs and key bytes. (In F7, *Length* is a logical library function.) Thus, for instance, typechecking would catch parsing errors leading to a call to *VERIFY* with a truncated MAC, as its argument would not match the refined type *mac*.

Intuitively, the ideal interface is too good to be true: a simple information-theoretic argument shows that any implementation such that (1) MACs are shorter than texts and (2) *MAC* and *VERIFY* communicate only through MACs must be unsafe. In particular, we do *not* have $\vdash C \rightsquigarrow I$. Next, we show how we can establish its weaker, asymptotic safety under cryptographic assumptions.

Security (*CMA*) Common assumptions on MACs are functional correctness and security. The correctness assumption, as well as other expected functional properties of our concrete implementation, are expressed using another *refined concrete interface* that declares

```
type mac = b:bytes{Length(b) = macsize}
predicate val GENerated: key → bool
predicate val MACed: key ∗ text ∗ mac → bool
val GEN: unit → k:key {GENerated(k)}
val MAC: k:key → t:text → m:mac {MACed(k,t,m)}
val VERIFY: k:key → t:text → m:mac →
  v:bool{ GENerated(k) ∧ MACed(k,t,m) ⇒ v = true }
val LEAK: k:key → b:bytes{Length(b) = keysize}
assume !k,t,m0,m1.
  GENerated(k) ∧ MACed(k,t,m0) ∧ MACed(k,t,m1) ⇒ m0 = m1
```

In this interface, written I^C in the following, two auxiliary predicates *GENerated* and *MACed* keep track of the results returned by *GEN* and *MAC*; the final assumption states that *MAC* is a deterministic function. These properties are easily met by any implementation that just recomputes the MAC for verification, as our concrete implementation does: to typecheck against I^C, it suffices to add events at the end of *GEN* and *MAC* to record their result.

The security assumption is expressed as a game. We adopt a standard notion, *resistance against existential Chosen Message forgery Attacks* (CMA), introduced by Goldwasser et al. [31] for signatures; we follow the presentation of Bellare et al. [14]. In this game, an adversary attempts to forge a valid MAC; to this end, it can adaptively call two *oracles* for computing and verifying MACs on a single, hidden key; the adversary wins if it can produce a text and a MAC such that verification succeeds but the text has not been passed to the MAC oracle. We set up this game using an F# module, written *CMA* and defined as follows:

```
let k = GEN()
let log = ref []
let mac t = log := t::!log; MAC k t
let verify t m =
  let v = VERIFY k t m in assert(not v || List.mem t !log); v
```

This code generates a key, implements the two oracles as functions using that key, and maintains a log of all texts passed as arguments to the *mac* function. We intend to run this code with an adversary given access to *mac* and *verify*, but not *k* or *log*. The verification oracle also tests whether any verified pair wins the game: the **assert** claims that this does not happen: either *v* is false, or *t* is in the log. (In the asserted formula, || is logical disjunction.)

DEFINITION 1. *C is CMA-secure when, for all p.p.t. expressions A with no* **assume** *or* **assert** *such that mac: text → mac, verify: text → mac → bool ⊢ A: unit, the expression C · CMA · A is asymptotically safe.*

We establish asymptotic safety for programs well-typed against the *ideal* interface of a CMA-secure MAC.

THEOREM 2 (ASYMPTOTIC SAFETY FOR MAC). *Let C be p.p.t. CMA-secure and such that $\vdash C \rightsquigarrow I^C$. Let A be p.p.t. such that $I \vdash A : unit$. The expression $C \cdot A$ is asymptotically safe.*

As illustrated at the end of this section, Theorem 2 usefully applies to expressions A composed of a fixed protocol and any adversaries typed against its protocol interface.

To prove Theorem 2, our main tool is an intermediate, well-typed functionality, written F and defined below, such that $C \cdot F$ implements the ideal interface I. We call $C \cdot F$ an ideal MAC implementation. Our next theorem states that the concrete and ideal implementations are indistinguishable; it can be used in compositional proofs relying on F. For example, §6 in the full paper uses it to show that encrypt-then-MAC implements authenticated encryption. (In terms of universal composability, C emulates F, with C as a simulator.)

THEOREM 3 (IDEAL FUNCTIONALITY FOR MAC). *Let C be p.p.t. CMA-secure such that $\vdash C \rightsquigarrow I^C$. Let A be p.p.t. such that $I \vdash A$. We have $C \cdot A \approx_\varepsilon C \cdot F \cdot A$.*

Ideal Functionality (F) Our ideal functionality re-defines the type for keys, now represented as integer indexes, and re-defines the functions *GEN*, *MAC*, *VERIFY*, and *LEAK* using those provided by any implementation of the refined concrete interface I^C. It maintains a global, private association table *ks* that maps each key index to some internal state, consisting of a concrete MAC key (*kv*), a list of previously-MACed texts (*log*), and a Boolean flag (*leaked*) indicating whether the key has been leaked or not. In the code below, qualified names such as **Mac**.*GEN* refer to entries from the concrete interface I^C, while names such as *GEN* refer to their re-definition. Our ideal functionality F is:

```
let ks = ref [] (* state for all keys generated so far *)
let GEN () =
  let k = list_length !ks
  let kv = Mac.GEN() in
  ks := (k,(kv,empty_log k,false_flag k))::!ks; k
let MAC k t =
  let (kv,log,leaked) = assoc k !ks in
  log := t:: !log; Mac.MAC kv t
let VERIFY k t m =
  let (kv,log,leaked) = assoc k !ks in
  Mac.VERIFY kv t m && (mem k t !log || !leaked)
let LEAK k =
  let (kv,log,leaked) = assoc k !ks in
  leaked := true; Mac.LEAK kv
```

where the functions *list_length*, *assoc*, and *mem* are the standard ML functions *List.length*, *List.assoc*, and *List.mem* with more precise refinement types, and where the functions *empty_log* and *false_flag* allocate mutable references that initially contain an empty list and **false**, respectively.

GEN allocates a key with index k (the number of keys allocated so far), generates a concrete key, and records its initial state, with an empty log and a flag set to **false**. When called with a key, the three other functions first perform a table lookup to retrieve its state (*kv*,*log*,*leaked*). *MAC* computes the MAC using the concrete key *kv* and adds the MACed text to the log. *VERIFY* performs the concrete MAC verification, then it corrects its result from **true** to **false** when (i) the concrete verification succeeds; but (ii) the text has not been MACed, as recorded in *log*; and (iii) the key has not been leaked, as recorded by *leaked*. *LEAK* returns the concrete key bytes, and sets the *leaked* flag to **true**.

The module $C \cdot F$ obtained by composing the concrete MAC implementation with F is a well-typed implementation of the ideal interface, as F corrects any unsafe verification of C, but it is unrealistic, as F relies on global shared state.

LEMMA 1 (TYPING). *$I^C \vdash F \rightsquigarrow I$.*

This lemma is proved by letting F7 typecheck *F*. Intuitively, it holds because the result of the verification function is always consistent with the log and the flag, whose refinement types record enough facts *Msg*(k,t) to establish the post-condition of *VERIFY*.

We provide a brief outline of the proof for Theorems 3 and Theorem 2, but refer to the full paper for the details. Classically, we rely on an "up to bad" lemma of [13], stating that $C \cdot F$ and C have exactly the same behaviour until F corrects a verification; when this happens, a related adversary wins against CMA. We use two main reductions, to account for dynamic key leakage, then for the use of multiple keys in F. To complete the proof of Theorem 2, we use Lemma 1 and the typing hypotheses of C and A to conclude from Theorem 1 that $C \cdot F \cdot A$ is perfectly safe.

Sample Protocol: An Authenticated RPC To illustrate the use of our MAC library for verification, we include the code of a sample protocol for remote procedure calls (RPC) adapted from Bhargavan et al. [19]. In the protocol, a client and a server exchange requests s and responses t coded as strings. To authenticate and correlate these strings, they use MACs computed from a shared key k:

```
let k = GEN()
let client s =
  assume (Request(s));
  send (concat (utf8 s) (MAC k (request s)));
  recv (fun msg →
    let (v,m′) = split macsize msg in let t = iutf8 v in
    if VERIFY k (response s t) m′ then assert(Response(s,t));())
let server () =
  recv (fun msg →
    let (v,m) = split macsize msg in let s = iutf8 v in
    if VERIFY k (request s) m then
     (assert (Request(s));
      let t = "22" in assume (Response(s,t));
      send (concat (utf8 t) (MAC k (response s t)))))
```

We omit the formatting functions, *request* and *response*, that compute the bytes actually MACed as tagged concatenations of their arguments. For simplicity we only consider the single key setting. Fournet et al. [30] show how to verify a variant of the protocol with multiple keys shared between pairs of principals using our MAC functionality; the details are available online.

Security is modelled as safety when running with an active adversary that calls the functions *client* and *server* (to trigger parallel sessions) and controls the network. To this end, these two functions include matching **assume** and **assert** for each message sent and accepted by the protocol, respectively. With this specification, type-safety entails two correspondence properties between these events, of the form "whenever the server accepts a request s, the client must have sent that request before" and similarly for responses. The protocol uses a single key, so its security depends on MACing unambiguous texts (formatted by *request* and *response*).

Let *Net* range over modules implementing the network and the scheduler, informally controlled by the adversary, with the following interface for the protocol:

$$I_{Net} \triangleq send\text{: bytes} \rightarrow \text{unit}, recv\text{: (bytes} \rightarrow \text{unit)} \rightarrow \text{unit}$$

In addition, *Net* may export an arbitrary interface I'_{NET} for the adversary. Let C_{RPC} be the sample protocol code above, after inlining its other (trusted) library functions, with the following interface for the adversary: $I_{\mathsf{RPC}} \triangleq client\text{: string} \rightarrow \text{unit}, server\text{: unit} \rightarrow \text{unit}$

We arrive at the following computational safety theorem:

THEOREM 4 (RPC SAFETY). Let C be a CMA-secure p.p.t. module with $\vdash C \rightsquigarrow I^C$. Let *Net* be a module with $\vdash Net \rightsquigarrow I_{\mathsf{NET}}$, $\vdash Net \rightsquigarrow I'_{\mathsf{NET}}$, and $Net \cdot C_{\mathsf{RPC}}$ is p.p.t. Let A be a p.p.t. expression with $I'_{\mathsf{NET}}, I_{\mathsf{RPC}} \vdash A : unit$.

The expression $C \cdot Net \cdot C_{\mathsf{RPC}} \cdot A$ is asymptotically safe.

(Technically, we state our p.p.t. assumption on *Net* composed with C_{RPC} because *Net* has a second-order interface.) For each concrete definition of *Net*, we obtain a computational safety theorem against adversaries A with access to the network and the protocol, guaranteeing that the two correspondence properties discussed above hold with overwhelming probability.

The proof of Theorem 4 roughly consists of letting F7 typecheck $I_{\mathsf{NET}}, I \vdash C_{\mathsf{RPC}} \rightsquigarrow I_{\mathsf{RPC}}$. As a sanity check, we outline a broken (non-typable) variant $C^\bullet_{\mathsf{RPC}}$ and a sample adversary A that breaks response authentication (with probability 1). Suppose that the formatting functions *request* and *response* simply concatenate their arguments—without tagging. Any valid MAC for a request will also be accepted as a valid MAC for the empty response to that request. The adversary A below meets the conditions of Theorem 4, and tries this simple reflection attack by triggering a request, intercepting it, and sending back a fake response with the same MAC:

```
client "hello";
let msg = recv'() in
let (v,mac) = split macsize msg in
send' (concat (utf8 "") mac)
```

where *send'* and *recv'* are part of the attacker interface I'_{NET}. The expression $C \cdot Net \cdot C^\bullet_{\mathsf{RPC}} \cdot A$ reduces to a configuration of the form $[X, \{Request(\texttt{"hello"})\}, E[\textbf{assert}(Response(\texttt{"hello"},\texttt{""}))]]$, with a failing assertion as the the client accepts the fake response. As can be expected, the protocol $C^\bullet_{\mathsf{RPC}}$ does not typecheck, since the precondition for calling *MAC* would have to imply $Response(s,\texttt{""})$.

Bhargavan et al. [19] verify similar code by typing against symbolic cryptographic interfaces. Although we now rely on computational assumptions, our protocol code still typechecks essentially for the same reasons. In comparison with their symbolic theorem, ours is also simpler—we do not need to give our adversary access to the cryptographic library, since the adversary may include its own implementation of MAC algorithms.

Experimentally, we also test that our protocol code runs correctly, at least when compiled with a *Net* library and an adversary that implements a reliable network and a simple test. The rest of the code is available online, including a sample (harmless) adversary that correctly schedules clients and servers for testing, and another adversary that tries the attack described above.

Public-Key Signatures We have also applied our verification method to public-key signature schemes, under similar assumptions, using RSA as sample implementation. The main differences are that we have distinct types for public and private keys, and that we give the protocol (and the adversary) unconditional access to the bytes of public keys. See the full paper for the resulting ideal functionality.

4. PERFECT SECRECY BY TYPING

In this section and the next, we focus on a simple notion of perfect secrecy and on *indistinguishability*, its cryptographic counterpart. We let $\approx$ represent probabilistic equivalence between closed terminating expressions: two expressions are equivalent when they return values with identical probabilities. Our notion of secrecy concerns values given an abstract type, written α in this section. With encryptions, α will be the type of plaintexts. Intuitively, protocols given values of type α cannot directly operate on them [44]; we use this parametricity property to establish equivalences between implementations of α. (Roy et al. [45] similarly restrict protocols, using a notion of secretive traces for protecting nonces.)

DEFINITION 2. *For a fixed type variable α, let* secret types *be of the form $T_\alpha ::= \alpha \mid T \to T_\alpha$ where T ranges over base types. Let* secret interfaces *be of the form $I_\alpha = \alpha, x_1 : T_{\alpha,1}, \ldots, x_n : T_{\alpha,n}$ for some $n \geq 0$. Let P_α range over modules that define* **let** *$x_i = v_i$ for some pure total values v_i for $i = 1..n$, such that $\vdash P_\alpha \rightsquigarrow I_\alpha$.*

THEOREM 5 (SECRECY BY TYPING). Let A such that $I_\alpha \vdash A : bool$. For all P^0_α and P^1_α, we have $P^0_\alpha \cdot A \approx P^1_\alpha \cdot A$.

The proof relies on a variant of the subject reduction lemma, which we verified in COQ. Experimentally, we use F7 to automatically check the typing condition of Theorem 5.

For example, consider a protocol C that operates on a secret string s (formally a free variable of C) using operations defined in library P (including a definition of s), and an adversary A that tries to gain information on s by interacting with the protocol using interface I_C. We are thus interested in systems of the form $P \cdot C \cdot A$ with $\vdash P \cdot C \rightsquigarrow I_C$ and $I_C \vdash A : bool$. To prove that $P \cdot C$ does not leak any information on s, it suffices to verify that P defines only pure values, to write a secret interface of the form $I_\alpha = \alpha, s : \alpha, \ldots$ for P, and to typecheck that $\vdash P \rightsquigarrow I_\alpha$ and $I_\alpha \vdash C \rightsquigarrow I_C$. Theorem 5 then applies to $C \cdot A$ and any two variants P^0_α and P^1_α of P that differ only in their definition of s, and thus the probability $\Pr[P \cdot C \cdot A \Downarrow b]$ does not depend on the value of s.

A Module for Plaintexts In preparation to encryption in §5, we introduce a sample module for plaintexts, written P, with two interfaces: a *secret interface* I_{PLAIN} with abstract type α=*plain*; and another, *concrete interface* I^C_{PLAIN} that reveals the representation of *plain*. We begin with the latter:

```
val plainsize: int
type repr = b:bytes {Length(b) = plainsize}
type plain = repr
val plain: repr → plain
val respond: plain → plain
```

The constant *plainsize* and the type *repr* define a fixed-length, concrete representation of *plain* values. (We need to bound the length of plaintexts, as otherwise encryption would necessarily leak some information about plaintexts.)

We let I_{PLAIN} be I^C_{PLAIN} with the abstract type declaration **type** *plain* substituted for the type alias **type** *plain = repr*. The interface I_{PLAIN} is secret, and such that $\vdash P \rightsquigarrow I^C_{\mathsf{PLAIN}}$ implies $\vdash P \rightsquigarrow I_{\mathsf{PLAIN}}$. (It does not strictly match Definition 2 because of the integer constant *plainsize*; however Theorem 5 applies after inlining *plainsize* in P.) In both interfaces, the functions *plain* and *respond* stand for any code that operates concretely on secrets, for instance *respond* may compute responses to secret requests in an RPC protocol; this code may be hoisted to P in order to apply Theorem 5. The function *plain* (implemented as the identity function) maps representations to plaintexts; it is useful, for instance, to let the adversary provide chosen values for plaintexts. Conversely, a function *repr* from plaintexts to concrete representations, necessary for instance to marshal plaintexts before their concrete encryption, would break secrecy. As explained in the next section, however, the secret interface I_{PLAIN} still enables ideal encryption.

5. SECRECY USING ENCRYPTION

We consider the secrecy guarantees provided by encryption, We begin with public-key encryption, then briefly discuss symmetric encryption. As for MACs in §3, we have a basic F# interface and a more precise concrete F7 interface (written I^C_{ENC}) that expresses functional correctness and enforces consistent bytes lengths for encryption. We let C_{ENC} range over p.p.t. implementations of I^C_{ENC}. The full paper gives these interfaces and a sample implementation based on RSA-OAEP.

Our ideal interface for public-key encryption (I_{ENC}), parameterized by the plaintext type of I_{PLAIN}, is as follows:

```
predicate val PKey: bytes → bool
type pkey = pk:bytes{Length(pk)=pksize ∧ PKey(pk)}
type cipher = b:bytes {Length(b)=ciphersize}
type skey
val GEN: unit → pkey ∗ skey
val ENC: pkey → plain → cipher
val DEC: skey → cipher → plain
```

The interface sets sizes for public keys and ciphertexts, and declares three algorithms: *GEN* generates keys, *ENC* encrypts using the public key, and *DEC* decrypts using the secret key. Secret keys *skey* are abstract and, in contrast with §3, the interface does not support their dynamic compromise, as this would lead to a well-known commitment problem [see e.g. 4]. Public keys *pkey* are concrete, fixed-length bytes; these keys may be passed to the adversary, who can then encrypt using its own implementation of *ENC*, or to other parties, who can authenticate them then encrypt their secrets. The refinement *PKey*(*pk*) tracks the public keys generated by *GEN* and prevents the use of untrusted keys.

Security of Encryption (CCA2) We use a standard notion of security, named indistinguishability under chosen-ciphertext attacks (CCA2) [43]. We follow the presentation of Bellare and Rogaway [12], allowing the adversary to perform multiple oracle encryptions. We formalize this notion as a game in F#, using two modules CCA^b for $b = 0,1$ obtained from the code below by replacing *select* $x_0\ x_1$ with x_0 and x_1, respectively.

```
let pk, sk = GEN ()
let log = empty_log pk
let enc x0 x1=
    let x = select x0 x1 in
    let v = ENC pk x in
    log := cons pk (v,x) !log; v
let dec v =
    match assoc pk v !log with
    | Some(x) → zero
    | None → DEC sk v
```

These modules generate a keypair then define oracles for encryption and decryption: the adversary is given access to *enc*, *dec* and *pk*, but not to the private decryption key *sk*. This is enforced by the interface

$I_{cca} \triangleq pk\colon pkey,\ enc\colon plain \to cipher,\ dec\colon cipher \to plain$

using the types of I^C_{PLAIN} and I^C_{ENC}. The oracle-encryption function *enc*, also known as a left-or-right oracle, encrypts either x_0 or x_1, depending on b; it also logs the pair (v,x) where v is the resulting ciphertext. The oracle-decryption function *dec* first checks whether its argument v is the result of an oracle encryption, and then returns a constant (*zero*), otherwise it performs the decryption. In the end, the adversary wins if it can guess b.

DEFINITION 3. *C_{ENC} is CCA2 secure when, for all p.p.t. modules P with $\vdash P \rightsquigarrow I^C_{\mathsf{PLAIN}}$ and A with $I^C_{\mathsf{PLAIN}}, I_{cca} \vdash A : bool$, we have $P \cdot C_{\mathsf{ENC}} \cdot CCA^0 \cdot A \approx_\varepsilon P \cdot C_{\mathsf{ENC}} \cdot CCA^1 \cdot A$.*

A first encryption theorem yields secrecy by typing for protocols whose plaintexts can be given a secret interface.

THEOREM 6 (ASYMPTOTIC SECRECY). *Let C_{ENC} be a p.p.t. CCA2-secure module with $I^C_{\mathsf{PLAIN}} \vdash C_{\mathsf{ENC}} \rightsquigarrow I^C_{\mathsf{ENC}}$. Let A be a p.p.t. expression with $I_{\mathsf{PLAIN}}, I_{\mathsf{ENC}} \vdash A : bool$.*

For any two pure p.p.t. implementations P^b of I_{PLAIN}, we have $P^0 \cdot C_{\mathsf{ENC}} \cdot A \approx_\varepsilon P^1 \cdot C_{\mathsf{ENC}} \cdot A$.

A second, more general theorem states the soundness of a well-typed ideal functionality for encryption F_{ENC}, defined below.

THEOREM 7 (IDEAL FUNCTIONALITY). *Let C_{ENC} be a p.p.t. CCA2-secure module with $I^C_{\mathsf{PLAIN}} \vdash C_{\mathsf{ENC}} \rightsquigarrow I^C_{\mathsf{ENC}}$, P a p.p.t. module with $\vdash P \rightsquigarrow I^C_{\mathsf{PLAIN}}$, and A a p.p.t. expression with $I^C_{\mathsf{PLAIN}}, I_{\mathsf{ENC}} \vdash A$. We have $P \cdot C_{\mathsf{ENC}} \cdot A \approx_\varepsilon P \cdot C_{\mathsf{ENC}} \cdot F_{\mathsf{ENC}} \cdot A$.*

In contrast with Theorem 6, Theorem 7 does not require that A be well-typed using secret plaintexts; instead, it says that, irrespective of I^C_{PLAIN}'s implementation, the use of encryption does not leak additional information about plaintexts. As illustrated in the rest of the paper, Theorem 7 is convenient for composing cryptographic modules, for instance with plaintexts carrying cryptographic materials such as short term keys.

Ideal Functionality (F_{ENC}) We let F_{ENC} be the module:

```
let ks = ref [] (∗ state for all keypairs generated so far ∗)
let zero = zeroCreate plainsize
let GEN () =
    let pk, sk = PKEnc.GEN ()
    ks := (pk,(sk, empty_log pk))::!ks; (pk,SK(pk))
let ENC pk x =
    let (sk,log) = assoc pk !ks in
    let c = PKEnc.ENC pk zero in log := cons pk (c,x) !log; c
let DEC (SK(pk)) c =
    let (sk,log) = assoc pk !ks in
    match assocc pk c !log with
    | Some(x) → x | None → plain (PKEnc.DEC sk c)
```

Our ideal functionality (for abstract plaintexts) calls a concrete public-key encryption module **PKEnc** (for plaintexts of type *repr*). It stores all concrete secret keys in a private association table *ks* indexed by public keys, together with a *log* of all ciphertext-plaintexts pairs for this key. It implements the type of secret keys as *SK* **of** *pkey*, also used to index the table. The refinement *PKey*(*pk*) of I_{ENC} ensures that all table lookups in *ks* succeed. To ideally encrypt x, F_{ENC} concretely encrypts *zero* (some constant *plainsize* bytes), and adds (c,x) to the log of the public key. To ideally decrypt c, F_{ENC} first searches for some (c,x) in the log; otherwise it concretely decrypts c and uses the function *plain* to turn the resulting *repr* into an abstract *plain*. Thus, F_{ENC} treats plaintexts parametrically, and can be typed using the secret interface I_{PLAIN}: we have $I_{\mathsf{PLAIN}}, I^C_{\mathsf{ENC}} \vdash F_{\mathsf{ENC}} \rightsquigarrow I_{\mathsf{ENC}}$ and this enables us to prove Theorem 6 from Theorems 5 and 7.

CPA Encryption and Authenticated Encryption We outline variants of our ideal interface for two other notions of security: resistance to chosen plaintext attacks (CPA) and authenticated encryption. We refer to the full paper and the code for the corresponding security definitions and variants of Theorems 6 and 7. We focus on symmetric encryptions, for simplicity and because authenticated encryption does not have a direct public-key analogue. (See also Küsters and Tuengerthal [38] for ideal functionalities for symmetric encryption with support for corruption.)

CPA Secure Encryption is an assumption weaker than CCA2, obtained by removing the decryption oracle, and meant to be used with authenticated ciphertexts.

Accordingly, our ideal CPA interface I^{cpa}_{SENC} declares

```
predicate val ENCrypted: key * plain * cipher → bool
val ENC: k:key → p:plain → c:cipher {ENCrypted(k,p,c)}
val DEC: k:key → c:cipher {∃p. ENCrypted(k,p,c)} → p:plain
```

where the predicate *ENCrypted(k,p,c)* in the refinements ensures that a well-typed program never attempts to decrypt a ciphertext not produced by *ENC*. The interface also includes a logical assumption, stating that a ciphertext is the correct encryption of at most one plaintext for a given key.

Authenticated Encryption is a stronger notion that provides both secrecy and authenticity. Accordingly, our ideal interface I^{ae}_{SENC} declares

```
predicate val Msg: key * plain → bool
val ENC: k:key → p:plain {Msg(k,p)} → c:cipher
val DEC: k:key → c:cipher → (p:plain {Msg(k,p)}) option
```

where the predicate *Msg* plays the same role as in §3. The precondition on *DEC* has disappeared, and *DEC* now returns an option: either some authentic plaintext, or *None* to report decryption error.

Sample Construction Hybrid encryption [12] supports public-key encryption for large plaintexts by encrypting them under a fresh symmetric key itself encrypted under the public key. To illustrate our method, we program hybrid encryption given a pair of public-key and symmetric-key encryptions and we show that, if both these encryptions are CPA, then hybrid encryption is also CPA. Our code defines:

```
let ENC pk t =
  let k = SEnc.GEN() in concat (PKEnc.ENC pk k) (SEnc.ENC k t)
let DEC sk c =
  let c0,c1 = split SEnc.ciphersize c in SEnc.DEC(PKEnc.DEC sk c0) c1
```

The code (C_{Hyb}) uses two encryption modules named **SEnc** and **PKEnc** (written C_{SENC} and C_{ENC} below), each parameterized by its own plaintext module (P and P_k): large, fixed-sized bytes for symmetric encryption and the resulting hybrid encryption, and symmetric keys for public-key encryption. Using F7, we verify that C_{Hyb} implements an interface ($I^{cpa,\mathsf{hyb}}_{\mathsf{ENC}}$, listed below) that includes a public key ideal interface for CPA and defines its *ENCrypted* predicate as a logical specification of hybrid encryption:

```
definition !pk,p,c. ENCrypted(pk,p,c) ⇔
  ∃c0,c1,k. (c = c0|c1 ∧Length(k)=SEnc.keysize
  ∧PKEnc.ENCrypted(pk,k,c0) ∧Length(c0)=PKEnc.ciphersize
  ∧SEnc.ENCrypted(k,p,c1) ∧Length(c1)=SEnc.ciphersize)
val GEN: unit →pk:pkey * sk:skey {pk=PK(sk)}
val ENC: pk:pkey →p:plain →c:cipher {ENCrypted(pk,p,c)}
val DEC: sk:skey →c:cipher {∃p. ENCrypted(PK(sk),p,c)} →
  p:plain{ENCrypted(PK(sk),p,c)}
```

We outline a proof of our two encryption theorems for hybrid encryption. To prove the ideal-functionality theorem, we apply ideal-functionality theorems first to public-key encryption, then to symmetric-key encryption. Let P and A be the plaintext module and main expression (as in Theorem 7). Let P_k define symmetric keys as public-key encryptions plaintexts. Starting with our concrete implementation $C \triangleq C_{\mathsf{SENC}} \cdot P_k \cdot C_{\mathsf{ENC}} \cdot C_{\mathsf{Hyb}}$, we have

$$\begin{aligned} P \cdot C \cdot A &\approx_\varepsilon P \cdot C_{\mathsf{SENC}} \cdot P_k \cdot C_{\mathsf{ENC}} \cdot F_{\mathsf{ENC}} \cdot C_{\mathsf{Hyb}} \cdot A \\ &\approx_\varepsilon P \cdot C_{\mathsf{SENC}} \cdot F_{\mathsf{SENC}} \cdot P_k \cdot C_{\mathsf{ENC}} \cdot F_{\mathsf{ENC}} \cdot C_{\mathsf{Hyb}} \cdot A. \end{aligned}$$

To apply the ideal public-key and symmetric-key theorems, we let F7 verify their two typing hypotheses:

$$I^{C,\mathsf{ENC}}_{\mathsf{PLAIN}}, I_{\mathsf{ENC}} \vdash P \cdot C_{\mathsf{SENC}} \cdot C_{\mathsf{Hyb}} \rightsquigarrow I^{cpa,\mathsf{hyb}}_{\mathsf{ENC}}$$
$$I^{C,\mathsf{SENC}}_{\mathsf{PLAIN}}, I_{\mathsf{SENC}} \vdash P_k \cdot C_{\mathsf{ENC}} \cdot F_{\mathsf{ENC}} \cdot C_{\mathsf{Hyb}} \rightsquigarrow I^{cpa,\mathsf{hyb}}_{\mathsf{ENC}}.$$

and we conclude with ideal functionality $F \triangleq C_{\mathsf{SENC}} \cdot F_{\mathsf{SENC}} \cdot P_k \cdot C_{\mathsf{ENC}} \cdot F_{\mathsf{ENC}} \cdot C_{\mathsf{Hyb}}$. To prove the asymptotic secrecy theorem (with hypotheses similar to those of Theorem 6), we use our new ideal-functionality theorem twice and Theorem 5:

$$P_0 \cdot C \cdot A \approx_\varepsilon P_0 \cdot F \cdot A \overset{\text{Thm 5}}{\approx_\varepsilon} P_1 \cdot F \cdot A \approx_\varepsilon P_1 \cdot C \cdot A$$

To apply Theorem 5, we let F7 verify $I_{\mathsf{PLAIN}} \vdash F \rightsquigarrow I^{cpa,\mathsf{hyb}}_{\mathsf{ENC}}$.

The full paper also describes a broken, non-typable variant of hybrid encryption that "forgets" to encrypt its symmetric keys, a sample protocol that encrypts a secret, and a sample passive adversary that breaks asymptotic secrecy using its own concrete implementation of the decryption algorithm.

6. APPLICATIONS

We show how to compose the typed interfaces and functionalities given in §3 and §5 to verify larger programs.

Multiple Instances of Cryptographic Libraries Formally, a single protocol can use several copies of the same functionality, with distinct copies of their ideal interface and distinct abstract types to protect their keys. Asymptotic security then follows from repeatedly applying our authenticity and secrecy theorems. This enables us, for instance, to verify protocols that use both MACs and signatures, or protocols that encrypt values with different concrete types. The full paper illustrates this approach, showing for instance how to encrypt the keys of another functionality, and how to verify variants of our RPC protocol with authenticated encryption.

Sample Construction: Encrypt-then-MAC We realize authenticated encryption [11] using Encrypt-then-MAC. Given a CMA-secure MAC C_{MAC} and a CPA-secure encryption C_{SENC}, we build a module C_{EtM} such that $C \triangleq C_{\mathsf{SENC}} \cdot C_{\mathsf{MAC}} \cdot C_{\mathsf{EtM}}$ implements authenticated encryption, with ideal functionality $F \triangleq C_{\mathsf{SENC}} \cdot F^{cpa}_{\mathsf{SENC}} \cdot C_{\mathsf{MAC}} \cdot F_{\mathsf{MAC}} \cdot C_{\mathsf{EtM}}$. Using the theorems of §3 and §5, we show that, for any plaintext module P such that $\vdash P \rightsquigarrow I^{C}_{\mathsf{PLAIN}}$ and every p.p.t. A such that $I^{C}_{\mathsf{PLAIN}}, I^{ae}_{\mathsf{SENC}} \vdash A$, we have $P \cdot C \cdot A \approx_\varepsilon P \cdot F \cdot A$.

We also use F7 to verify that F implements the ideal authenticated encryption interface of §5, that is, $I_{\mathsf{PLAIN}} \vdash F \rightsquigarrow I^{ae}_{\mathsf{SENC}}$.

Sample Implementation: Secure Multiparty Sessions We outline the application of our method to larger existing protocols, and consider the cryptographic implementations of secure multiparty sessions of Bhargavan et al. [18], excluding their fixed key distribution sub-protocol.

Multiparty sessions are global, structured communication protocols. Sessions are specified using a declarative language that indicates who can send which messages, when, and what those messages can contain. From this high-level specification, one can infer generalized correspondence properties, named *session integrity*, that account for parallel runs of sessions and the compromise of some of their principals, and then generate custom cryptographic protocols that realize all these properties. To this end, Bhargavan et al. developed a verifying protocol compiler that generates both the protocol code (in F#) and detailed type annotations (in F7) embedding the multiparty session logic and the refinements for each of the MAC keys of the protocol—essentially a large definition of the predicate *Msg* presented in §3. The compiled code is independently verified by F7. Some of their sample protocols involve up to six roles and hundreds of different cryptographic messages formats, handled by more than 5 000 lines of ML protocol code and 4 000 lines of F7 declarations.

Nonetheless, the refined interface for MACs used for typechecking their protocols is almost identical to the ideal interface I_{MAC} of §3, thanks to its support for dynamic key generation and key compromise. Thus, we directly benefit from their symbolic verification effort: for each protocol implementation that they generate and typecheck against their cryptographic library interfaces, Theorem 2 applies and, assuming that their MAC algorithm is CMA-secure, the generated protocol is also asymptotically safe, and their notion of session integrity holds with overwhelming probability.

7. CONCLUSIONS AND FUTURE WORK

We have shown how to express and verify the computational security of sample cryptographic functionalities (once and for all) using ideal refinement-typed interfaces.

(1) The cryptographic proofs of these functionalities are independent and relatively simple; they mostly rely on programming and typechecking.

(2) The security proofs for protocols using these cryptographic functionalities entirely rely on automated typechecking, much as in prior work on symbolic verification.

(3) Their verified security properties are directly expressed as (asymptotic variants of) safety and perfect secrecy on the protocol code.

This yields an effective, modular method for specifying and verifying protocol code under standard assumptions.

Although we conducted our experiments using F# and F7, we would like to carry over our approach to protocols coded in lower-level languages such as C and C++. Refinement types are a convenient way to implement point (2) above, but any general-purpose static verification tool that can prove that a program conforms with our ideal interfaces will do.

Acknowledgments Many thanks to Martín Abadi, Aslan Askarov, Karthik Bhargavan, Avik Chaudhuri, Andy Gordon, Gurvan le Guernic, James Margetson, Tamara Rezk, Nikhil Swamy, Dimitrios Vytiniotis and the anonymous reviewers for their help.

References

[1] M. Abadi. Secrecy by typing in security protocols. *JACM*, 46(5), 1999.

[2] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *JACM*, 52(1), 2005.

[3] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Crypt.*, 15(2), 2002.

[4] M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication in a simulatable Dolev-Yao-style cryptographic library. *Int. J. Inf. Sec.*, 4(3), 2005.

[5] M. Backes, B. Pfitzmann, and M. Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. *Inf. Comput.*, 205(12), 2007.

[6] M. Backes, M. Dürmuth, D. Hofheinz, and R. Küsters. Conditional reactive simulatability. *Int. J. Inf. Sec.*, 7(2), 2008.

[7] M. Backes, D. Hofheiz, and D. Unruh. CoSP: A general framework for computational soundness proofs. In *ACM CCS 09*, 2009.

[8] M. Backes, M. Maffei, and D. Unruh. Computational sound verification of source code. In *ACM CCS 09*, 2010.

[9] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Formal certification of ElGamal encryption. In *FAST 2008*, 2008.

[10] G. Barthe, B. Gregoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO 2011*, 2011.

[11] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *J. Crypt.*, 21, 2008.

[12] M. Bellare and P. Rogaway. Introduction to modern cryptography. In *UCSD CSE 207 Course Notes*, 2005.

[13] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT 2006*, 2006.

[14] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO'96*, 1996.

[15] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. Technical Report MSR–TR–2008–118–SP1, Microsoft Research, 2008.

[16] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for TLS. In *ACM CCS 09*, 2008.

[17] K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy. Verified implementations of the Information Card federated identity-management protocol. In *ASIACCS'08*, 2008.

[18] K. Bhargavan, R. Corin, P.-M. Dénielou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF 2009*, 2009.

[19] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL 2010*, 2010.

[20] B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, 2006.

[21] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *JLAP*, 75(1), 2008.

[22] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[23] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *CSF 2009*, 2009.

[24] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *POPL 2008*, 2008.

[25] A. Datta, A. Derek, J. C. Mitchell, V. Shmatikov, and M. Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *ICALP 2005*, 2005.

[26] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (PCL). *Electr. Notes Theor. Comput. Sci.*, 172, 2007.

[27] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2), 1983.

[28] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *CSF 2011*, 2011.

[29] N. A. Durgin, J. C. Mitchell, and D. Pavlovic. A compositional logic for proving security properties of protocols. *JCS*, 11(4), 2003.

[30] C. Fournet, K. Bhargavan, and A. Gordon. Cryptographic verification of protocol implementations by typechecking. In *FOSAD*, 2011.

[31] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attack. *SICOMP*, 17(2), 1988.

[32] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI'05*, 2005.

[33] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005.

[34] D. Hofheinz, D. Unruh, and J. Müller-Quade. Polynomial runtime and composability. Cryptology ePrint Archive, Report 2009/023, 2009.

[35] B. M. Kapron and S. A. Cook. A new characterization of type 2 feasibility. *SICOMP*, 25, 1996.

[36] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication, 1997. RFC 2104.

[37] R. Küsters. Simulation-based security with inexhaustible interactive Turing machines. In *CSFW 2006*, 2006.

[38] R. Küsters and M. Tuengerthal. Universally composable symmetric encryption. In *CSF*, 2009.

[39] P. Laud. Semantics and program analysis of computationally secure information flow. In *ESOP 2001*, volume 2028, Apr. 2001.

[40] P. Laud. Secrecy types for a simulatable cryptographic library. In *ACM CCS 09*, 2005.

[41] P. Laud. On the computational soundness of cryptographically-masked flows. In *POPL 2008*, 2008.

[42] J. H. Morris, Jr. Protection in programming languages. *CACM*, 16(1), 1973.

[43] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO'91*, 1991.

[44] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 1983.

[45] A. Roy, A. Datta, A. Derek, and J. C. Mitchell. Inductive trace properties for computational security. *JCS*, 18(6), 2010.

[46] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2011.