So far, we have discussed several cryptography primitives, from hash functions to encryption schemes, and explored many applications of those primitives to systems security. These primitives have provided security, but in a very all-or-nothing sense: in order to provide broadly applicable security, our definitions required that a certain party (with the key) could either completely decrypt the message, learning the message contents, or cannot do anything with the message at all.

In some cases, however, it may be useful to loosen these restrictions, adding features to our encryption schemes.

1 Functional Encryption

One feature that turns out to be useful is the ability for some party to compute a certain function of a message without learning the message itself.

Example: Email Encryption. Email is a common form of communication, and it seems like it would be desirable for it to be end-to-end encrypted so that only the sender and recipient can learn the message contents. However, email is not end-to-end encrypted. This is because end-to-end encryption prevents many kind of important functionality—for example, if we applied ElGamal encryption to our emails, our email server would not be able to inspect incoming emails to filter spam—all emails would look totally random to the server! In order to hide our messages from the server while still allowing the server to classify messages as spam, we need a new type of encryption scheme called functional encryption.

Definition 1.1 (Functional Encryption). A functional encryption scheme is one that allows, for all functions $f$, generating $sk_f$ such that it is possible to efficiently determine $f(m)$ given $Enc(m)$ and $sk_f$, but nothing else.

It turns out that problem of functional encryption is equivalent to the problem of program obfuscation.

1.1 Program Obfuscation

Imagine we are providing bug fixes, including security fixes, to a piece of proprietary software. We would ideally like to avoid dis-
closing the security bugs in the old version of the software, but if we distribute our program anyone can inspect the changes and see what they are fixing. Ideally, we would have a way of distributing software that does something without revealing what the software does. We would like some obfuscator $O$ such that for a given program $C$, $O(C)$ is “unintelligible” but provides identical input-output behavior.

Current approaches to this problem are heuristic-based, involving things like adding useless code that were intended to make it difficult to reverse-engineer the changes. More recently, research has defined a very weak form of obfuscation called indistinguishability obfuscation and shown that this notion of indistinguishability obfuscation is equivalent to functional encryption. Indistinguishability Obfuscation does not provide obfuscation itself, but provides that for any two programs $C_1$ and $C_2$ with the same input-output behavior, the obfuscator $O$ provides that $O(C_1) \cong O(C_2)$—that it is not possible to distinguish between the obfuscations of two functions with the same input-output behavior.

It turns out that we could generate many of the primitives that we have discussed so far just from this notion of indistinguishability obfuscation.

2 Zero-Knowledge Proofs

We have said broadly that our goal with encryption and other primitives is that the adversary “learns nothing” about the statement. However, this has clearly not been the case so far: the adversary learns the encryption of the message or the hash of the message.

For a zero-knowledge proof, we would like to have a way of proving that a statement is true without revealing any information beyond the fact that the statement is true.

As a real-world example, consider that Alice has two balls and would like to prove to Bob that the two balls are different without revealing how they are different. To do this, Alice can give the balls to Bob and tell Bob to put them behind his back and decide whether to switch the two balls or not switch the two balls. Bob should then give the balls to Alice, and Alice will tell Bob whether the balls were switched or not. If Alice is right many times, Bob can be quite sure that the balls are indeed different, since Alice can reliably distinguish them.

This does not sound at all like a typical proof that you might write down on paper. Indeed, zero-knowledge proofs are impossible using “classical proofs”. However, zero-knowledge proofs are possible using interactive proofs.
**Definition 2.1** (Interactive Proof). An interactive proof consists of two parties: a “prover” $P$ and a “verifier” $V$. If $P$ knows the “proof” of the statement, then $P$ is accurate with probability 1. If the statement is false, then $V$ accepts the proof with probability less than $\frac{1}{2}$.

Perhaps surprisingly, we can show that any proof at all can be converted into a zero-knowledge proof.

As another example, consider proving that a graph is 3-colorable. The input is a graph $G$, and $P$ aims to prove to $V$ without revealing any information that it is possible to assign colors 1, 2, or 3 to each vertex such that no two adjacent vertices have the same color. This is easy to prove in the non-zero-knowledge case: $P$ could simply provide a coloring of $G$, which $V$ can easily verify. However, this reveals information about the coloring.

To prove that the graph is 3-colorable without revealing information about $G$, the verifier will first permute the colors $\{1, 2, 3\}$. $P$ will then provide $n$ commitments to $V$—statements that commit $P$ to a given color for each vertex without revealing those colors to $V$.

$V$ then picks an edge $(i, j)$ from the set of edges in $G$ and sends that pair to $P$. $P$ then verifies their commitments of the colors of $i$ and $j$ to $V$ by revealing the colors. $V$ accepts the response if the two colors are distinct and the corresponding commitments are valid. If $G$ is not three-colorable, this will fail with probability at least $\frac{1}{|E|}$. Thus, this can be repeated to achieve the desired probability.

The statement of 3-colorability is NP-complete, meaning that any NP statement can be converted to 3-colorability. That is, it is possible to write a reduction from any statement with a polynomially-sized witness to 3-colorability such that the existence of a 3-coloring of the corresponding graph proves that the original statement is true. Therefore, this shows that we can generate a zero-knowledge proof for any statement in NP!

In fact, interactive proofs can be used to efficiently prove anything in PSPACE, which is believed to be much larger than NP.