Review: Indistinguishability Obfuscation from Functional Encryption

Sakura Lim, Will Hawkins
CS 6501, Spring 2015

1 Introduction

In *Indistinguishability Obfuscation from Functional Encryption*, Bitansky and Vaikuntanathan propose an indistinguishability obfuscation (IO) scheme based on functional encryption (FE) primitives. Their result represents an advance in the state-of-the-art to cryptography researchers who have searched for IO constructions built on standard cryptographic assumptions. To date, IO constructions rely on the relatively untested multilinear maps primitive [GGH13, BV15].

At a high level, program obfuscation schemes "hide" an input program in a modified version. At runtime, the modified version of the program behaves exactly as the original but anyone who looks at the variant cannot learn anything about the original (beyond how it operates, obviously). The realization of a scheme for program obfuscation, of which IO is a type, would give software developers a powerful practical tool. Developers could use it to confidently distribute their programs knowing that proprietary implementation details and algorithms are hidden from prying eyes. Video game producers could use it to prevent circumvention their DRM scheme and defeat piracy. Software maintainers could use it to release patches knowing that hackers would be unable to reverse engineer the update to infer the security vulnerability.[1]

Program obfuscation would also be a useful academic tool, beneficial to high-level cryptographic operations as a primitive. Researchers have shown that it offers the requisite power to implement public key cryptography [BB01]! Even a weaker notion of program obfuscation like IO is a useful cryptographic primitive, "powerful enough to give rise to almost any known cryptographic object" [BV15].

Like IO, functional encryption is both a useful theoretical and practical tool. It can be used by other cryptographic constructions and also by users who want to selectively release access to their data.

Bitansky and Vaikuntanathan are not the first researchers to study the connection between FE and IO. Previous research has proven that IO can be used to create FE. By describing an IO construction from FE, Bitansky and Vaikuntanathan show that FE implies IO and, therefore, establish their equivalence ("up to some sub-exponential loss" [BV15]).

1.1 Functional Encryption

FE is a system that allows a user to define precise control over how much of the underlying plaintext to

1 Credit for these examples due to [BB01].

reveal from a given ciphertext. In the work that first precisely defined the concepts of FE, Boneh et al described it as a step forward in cryptography akin to the jump from private- to public-key encryption schemes.

As they explain it, traditional public and private key encryption schemes allow an encryptor to distribute a key that gives all-or-nothing access to the ciphertext's plaintext. For a public-key encryption scheme (Setup, Keygen, Enc, Dec), the decision is encoded by *f*():

$$f\left(SK, Enc\left(PK, m\right)\right)=\begin{cases} m \\ - \end{cases}$$

where

$$\underset{(PK, SK)\leftarrow Keygen}{Pr}\left[f\left(SK, Enc\left(PK, m\right)\right)=m\right]=1-\epsilon$$

for some negligible ε [KL07]. In FE, the decision to reveal ciphertext's plaintext does not have to be binary. User-defined functions encoded by the encryptor and distributed to others can reveal some parts of the ciphertext and not others. For example, an encryptor creates the ciphertext of their favorite number and distributes this as *c*. Then, the encryptor gives out a key that reveals to the evaluator whether or not their favorite number is even or odd but nothing more. The decision encoded by that key is *f*():

$$f\left(k, Enc\left(n\right)\right)=\begin{cases} true & if\ 2\div n=0 \\ false & otherwise \end{cases}$$

Boneh et al showed that this concept of functional encryption is a revolutionary advance in cryptography by demonstrating that it captures other evolutionary advances from the previous two decades. In their work they showed that the identity-based encryption (IBE) and attribute-based encryption (ABE) could be restated in terms of FE and even formulated PKE in terms of FE.

What happens if an encryptor wants to protect information about the function itself and the data? As defined by Boneh et al, a FE system keeps the encryptor's data secret but does not make any guarantees about the function. In other words, users with access to the keys for the even-odd function may be able to learn the function itself. An obvious extension to FE is function private FE (FPFE). In FPFE the possessor of a functional key *fk* for *f* and a ciphertext *c* for *m* cannot determine anything about *f* from *fk* and *c* besides the value of *f* when applied to *m*.

There are two variants of FPFEs: public- and private-key. In the private-key variant, only the master authority can issue valid ciphertexts for a function-hiding functional key. In addition to fetching function-hiding function key from the master authority, the user must interact with the master authority again to get a ciphertext of an input to the function in order to evaluate. In the private-key variant, the user can independently generate ciphertexts and evaluate the function for those hidden inputs using the function-hiding function key.

See Section 3.2 for a complete definition of FPFEs.

As Brakerski et al report in their paper, this has an immediate application for the user of cloud services. Consider a cloud service that hosts encrypted versions of user's data. The cloud service allows users to upload queries to their servers and retrieve matching data. Using FE, the user could query the server without the provider ever knowing anything more about the data than whether it matched the query. Unfortunately, the provider is able to learn the contents of the query itself. This is not an ideal situation for a user whose query contains sensitive terms like social security number or password. The situation is entirely different when the user employs FPFE. The user's query and the data are both kept from the cloud provider at all times.

1.2 Obfuscation

The question is, does it match the capabilities assumed to exist under the academic definition of program obfuscation? As mentioned above, there are several different versions of program obfuscation. Barak et al in 2001 were the first to systematically define these different versions.

According to Barak et al, the strongest form of program obfuscation is Virtual Black Box (VBB) program obfuscation. To understand VBB program obfuscation, think about this example. There is a program *P* that adds a user-provided input number to *S*, a secret number preprogrammed into *P*'s code code. A user Alice has "remote" (or oracle) access to the program. In other words, she can supply an input *i* and the oracle will respond with *P*'s output, *i* + *P*. A user Brenda, on the other hand, has complete, unlimited access to an obfuscated version *O(P)* of *P*. She can run *O(P)* (recall that an obfuscated program operates the same as the original at runtime) and the program will respond with *i* + *S*.

Alice and Brenda, given enough time, can both determine the program's output for every single input. Although this is a contrived example, the point is Alice will never be able to figure out *S* because she cannot look at the program.

Now assume that, no matter how hard she looks at *O(P)*, Brenda cannot figure out *S* either. In essence, although she controls a version of *P* and can manipulate it as she pleases, she knows exactly the same information about the program as Alice. In this case, *O(P)* is a VBB obfuscation of *P*.

On the other hand, assume that, because she has access to inspect *O(P)*, Brenda is able to uncover the secret number *S*. With knowledge of *S*, Brenda in some sense "knows" more about *P* than Alice – Brenda can anticipate program's output without having to execute it like Alice. In this case, *O(P)* is not a VBB obfuscation of *P*.

Immediately upon defining VBB in [BB01], the authors found a counterexample to prove that, in the general case, VBB is impossible to achieve. Almost as an afterthought, they proposed several weaker definitions of program obfuscation – IO is one of those. IO is more intuitive than VBB.

Assume that there are a set of programs {*P*} that all perform the same task and a hacker picks two programs, P_1 and P_2, from that set. The hacker gives those programs to a challenger and the challenger randomly chooses one to obfuscate and return to the hacker. The challenger asks the hacker to guess whether he/she has $O(P_1)$ or $O(P_2)$. In possession of $O(P_?)$, the hacker uses all of his/her skills to

decipher the program. In the end, however, the hacker cannot make an educated guess about which program the challenger obfuscated and decides to flip a coin to respond to the challenger. In this case, the obfuscation is an indistinguishable obfuscation.

On the other hand, if the hacker can correctly determine (not necessarily perfectly, but with some reasonable probability) that he/she has $O(P_1)$ and not $O(P_2)$ (or vice versa), then the obfuscation is not an indistinguishable obfuscation. See below for the formal definition of IO.

Recently Garg et al published the first candidate IO. He and his colleagues built their construction on multilinear maps applied to programs that had been transformed into matrixes according to Barrington's Theorem.[2] While a reasonable cryptographic assumption, multilinear maps are not as well-studied as its less general cousin the bilinear map. Cryptographers wary of such untested assumptions have been trying to find another candidate IO construction that uses less exotic assumptions. Bitansky and Vaikuntanathan have done just that by building an IO scheme from FE.

1.3 Roadmap

The rest of this review paper is organized as follows. Section 2 describes the goal of Bitansky and Vaikuntanathan's construction. Section 3 provides the context for the method of their construction: Section 3.1 describes the objects (a FE scheme, a symmetric private-key encryption scheme and a puncturable pseudorandom function) and Section 3.2 outlines a related construction presented by Brakerski and Segev upon which Bitansky and Vaikuntanathan model their construction and proof. Section 4 puts the pieces together: Section 4.1 describes Bitansky and Vaikuntanathan's actual construction and Section 4.2 gives a high-level description of the proof. The paper concludes with a discussion of the outstanding open questions in this subfield of cryptography and possibilities for future work.

2 Result

2.1 Results

In their own words, Bitansky and Vaikuntanathan “construct indistinguishability obfuscation from any public-key functional encryption scheme with succinct ciphertexts and sub-exponential security” [BV15]. This is a very dense statement that needs to be unpacked.

Think of their construction as a type of computer program. The program takes three inputs and generates a single output. The inputs are a public-key FE scheme, a symmetric private-key encryption (SKE) scheme and a punctural pseudorandom function (PPRF). The output is an IO.

Precise definitions for each of these primitives will be given in Section 3.1.

2.2 Outline of Construction

Figure 1 gives an overview of Bitansky and Vaikuntanathan's construction. Throughout this review paper, we will discuss each of the steps in detail but it is useful to get the overall picture before going

2 Dr. Mahmoody described Barrington's Theorem in his lecture Algebraic Computation on April 1, 2015.

further. Let *f* be the functionality that the construction is attempting to obfuscate.

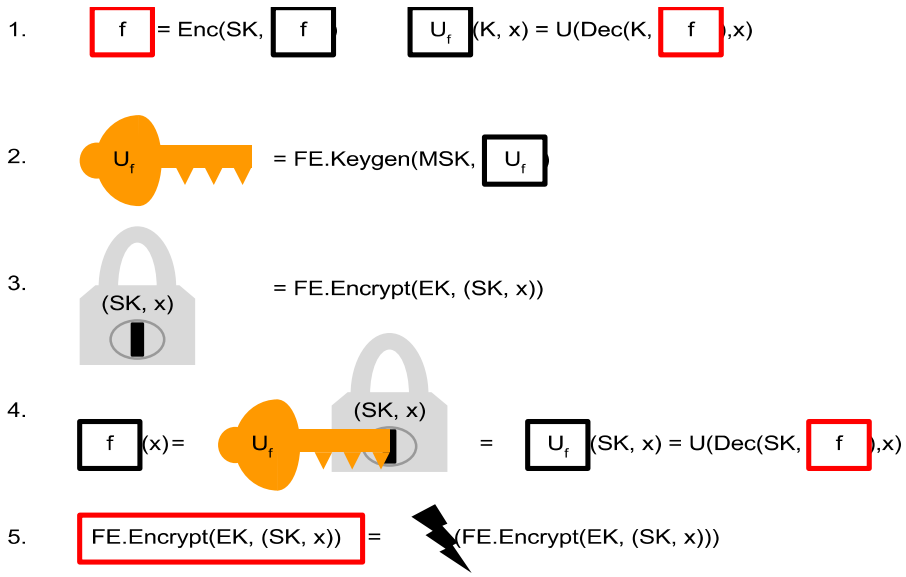

Figure 1: Overview of Bitansky and Vaikuntanathan's construction of IO from FE.

In a sense, under FPFE there is already some program obfuscation at work [BV15]. Informally, a public-key FPFE meets the specification for a IO: The user cannot learn anything about the function besides its outputs and the user can run the program independently. The problem is that constructions of public-key FPFEs do not exist and, worse, are not even well-defined [BS15].

The authors look for help in a private-key FPFE. They follow the construction of such a primitive provided by [BS15]. Steps 1 – 4 parallel that construction. In Step 1, *f* is encrypted with a symmetric private key to generate *f'*. Function U_f is a function that will decrypt *f'* with *K* and invoke it on the input *x*. In Step 2, an FE is used to generate a function key for U_f. Notice that because U_f contains only *f'*, the user cannot learn anything about *f* from the function key without the corresponding SK.

Step 3 generates the ciphertext for the pair (*SK*, *x*). A user in possession of such a ciphertext could use the function key of U_f to evaluate *f*(*x*) (Step 4).

Because the function key of U_f is distributed to the user, the first criteria for an IO is met: a user in possession of the obfuscated program cannot learn anything besides its output on given inputs. However, to meet the second criteria of IO, the user would need access to *SK* to complete Step 3 independently of the master authority.

This limitation would not cause a problem if VBB program obfuscation existed: The user could have access to a VBB obfuscation of the private-key FPFE's ciphertext generator (Step 5). This would complete the construction. To overcome the fact that VBB obfuscation does not exist, Bitansky and Vaikuntanathan create a recursive obfuscator (See Section 4.1) that is capable of hiding functions like the **Encrypt** operation of an FE.

2.3 Limitations

The recursive obfuscator is called once for each bit of the representation of the circuit that implements the **Encrypt** operation. At a minimum, this circuit must be built with hardwired access to the FE's encryption key and a copy of the symmetric key. Because the length of those keys is related to the number of invocations of the recursive obfuscator, the underlying public-key FE must have sub-exponential security with compact ciphertexts. See Section 4.1 and Section 4.2 for details.

Moreover, all existing FE constructions rely on IO. Using Bitansky and Vaikuntanathan's construction to get IO would create a loop in the assumptions and, therefore, does not currently offer much power. That said, when researchers create FE from other primitives, their construction will be an important tool.

3 Underlying Assumptions

3.1 Objects

Functional Encryption: An FE is a set of four algorithms (Setup, Keygen, Encrypt and Decrypt):

- **Setup**: The Setup algorithm generates an encryption key EK and a Master Secret Key (MSK) from the security parameter λ: *EK*, *MSK* = Setup(λ).
- **Keygen**: The Keygen algorithm generates a function key FK_f for the function *f* from the *MSK*: FK_f = Keygen(*MSK*, *f*).
- **Encrypt**: The Encrypt algorithm allows the holder of *EK* to encrypt an input *x* in such a way that it can be later used in the evaluation of *f*: *m* = Encrypt(*EK*, *x*).
- **Decrypt**: The Decrypt algorithm undoes the Encrypt operation: *x'* = Decrypt(FK_f, *m*).

To be syntactically well-defined, *f*(*x*) = Decrypt(FK_f, Encrypt(*EK*, x)).

According to the informal definition from above, the goal of an FE is to prevent an adversary from being able to learn anything from FK_f = Keygen(*MSK*, *f*) and *m* = Encrypt(*EK*, *x*) besides *f*(*x*). It is possible to capture this notion in terms of a guessing game G_{fe} between an adversary *A* and a challenger *C*.[3] The challenger instantiates an instance of the FE. Adversary *A* chooses any function *f* and *C* provides the appropriate FK_f. Adversary *A* may use EK to encrypt as many *x*'s as he/she wants as long as they are not the inputs that it ultimately submits to *C*. Eventually *A* submits an x_0 and x_1 to *C*. *C* selects $b \leftarrow \{0,1\}$ uniformly at random and returns Encrypt(*EK*, x_b) to *A*. Adversary *A* wins the guessing game if it can detect whether *C* encrypted x_0 or x_1. To be fair to *C*, all of *A*'s queries must meet the following condition:

$$f(x_0) = f(x_1).$$

If the adversary did not abide by the restriction, it could easily tell the difference between m_0 =

3 Boneh et al contend that it is not possible to codify the details of the security of an FE in an guessing game. They find that a simulation definition is required.

Encrypt(EK,x_0) and m_1 = Encrypt(EK,x_1). Define an experiment that uses G_{fe} and accepts the parameter, b, that determines which input to encrypt and results in a single bit *b'* that represents the challenger's guess:

$$Expt\left(G_{fe}\left(A,C\right),b\right)=b'$$

If $b = b'$, then the adversary wins; otherwise, the adversary loses. If A cannot win G_{fe} with more than

$$\frac{1}{2}+\epsilon$$

probability (where ε defines the security of the scheme, usually in terms of the security parameter), then the scheme is considered to be *selective message secure*. In other words if,

$$\left|Pr\left[Expt\left(G_{fe}\left(A,C\right),0\right)=1\right]-Pr\left[Expt\left(G_{fe}\left(A,C\right),1\right)=1\right]\right|\leq\epsilon.$$

then the scheme is selective message secure.

Private-Key Functional Encryption: When a single master authority holds both *EK* and *MSK*, it is referred to as a *private-key FE scheme*. The guessing game must be modified in this case. Instead of being able to encrypt inputs itself, the adversary *A* must rely on an encryption oracle provided by the challenger.

Public-Key Functional Encryption: When the master authority holds only *MSK* but distributes *EK*, it is known as a *public-key FE scheme*. The syntactic definition of the scheme must be modified in this case. The Encrypt operation must explicitly include randomness into its algorithm:

1. **public-key Encrypt**: The Encrypt algorithm allows the holder of *EK* to encrypt an input *x* using randomness *r* in such a way that it can be later used in the evaluation of *f*: *m* = Encrypt(*EK*, *x*, *r*). Bitansky and Vaikuntanathan note that many researchers will omit the explicit *r* when it can be inferred from the context. However, in their final construction, they explicitly construct *r*.

As is the case for public-key message encryption schemes, unless the encryption function uses randomness, repeated encryptions of the same input would be identical and easily detected by the adversary in a guessing game. [KL07] outlines the importance of randomness for public-key encryption schemes.

Symmetric Private-key Encryption: Symmetric private-key encryption schemes are standard cryptographic constructions. See [KL07] for a complete syntactic and semantic definition of such a scheme.

Puncturable Pseudorandom Function: A PPRF is a special case of PRF[4] that has a special "puncturable" point. Let *K* index the PRF underlying the PPRF that can be obtained by sampling Gen_{PRF}. A PPRF has an efficiently-computable function Punc() that takes a PRF index and a point and returns a *punctured point*:

4 [KL07] provides a detailed definition of PRFs.

$$K\{x^*\} = Punc(K, x^*).$$

Note that $K\{x^*\}$ is simply notation. To be a valid PPRF, the following must hold:

1. 1. For every x^* and x, the output of the function obtained from the PRF when indexed by the punctured point $K\{x^*\}$ on input x matches the output of the function obtained from the PRF when indexed by K on input x. In other words, for any x^* and $K \leftarrow Gen_{PRF}$,

$$for\ any\ x^* \forall x\ != x^*\ PRF_K(x) = PRF_{Punc(K,x^*)}(x)$$

2. 2. There is no polynomial time distinguisher than can distinguish between (x^*, $K\{x^*\}$, $PRF_K(x^*)$) and (x^*, $K\{x^*\}$, u) where $u \leftarrow \{0,1\}^*$.

Bitansky and Vaikuntanathan use the PPRF to generate explicit randomness for a public-key FE scheme that underlies their construction. They leverage this special type of randomness and apply the arguments of [CLTV15] at a crucial point in their proof. See Section 4.2 for more information.

The PPRF property that bounds the probability that a distinguisher is able to differentiate between its output and a truly random sequence of bits is key in their proof that their construction is an IO.

Indistinguishability Obfuscation Scheme: The construction's output is an IO scheme. An IO scheme takes as input a single circuit C and defines a scheme with a single operation Obfuscate:[5]

- **Obfuscate**: Obfuscate takes a circuit C as a parameter and constructs c: $c = Obfuscate(C)$

To be syntactically well-defined, c must behave exactly the same as C. Precisely,

$$\forall x \in X, c(x) = C(X).$$

To be indistinguishably obfuscated, a circuit c_1 generated by $iO()$ from C_1 must look the same as a circuit c_2 generated by an independent invocation of $iO()$ on C_2 when C_1 and C_2 compute the same output. This, too, can be captured in the context of a guessing game, G_{io}, between an adversary A and a challenger C. Adversary A picks any two circuits C_1 and C_2 from {C}, a set of circuits that perform the same computations. Adversary A gives C_1 and C_2 to C. Challenger C chooses $b \leftarrow \{0,1\}$ uniformly at random and returns $iO(C_b)$ to C. C wins the guessing game if it is able to determine whether or not A returned $iO(C_1)$ or $iO(C_2)$. Define an experiment that uses G_{io} and accepts the parameter, b, that determines which circuit to obfuscate and results in a single bit b' that represents the challenger's guess.

$$Expt(G_{io}(A, C), b) = b'$$

If $b = b'$, then the adversary wins; otherwise, the adversary loses. If C cannot win the guessing game with greater than

$$\frac{1}{2} + \epsilon$$

probability (where ε defines the security of the scheme, usually in terms of the security parameter),

5 This definition of an IO scheme is based on the definition in [BB01] and [GGH13].

then *iO* is an indistinguishable obfuscator. In other words, if

$$\left|Pr\left[Expt\left(\mathrm{G}_{io}\left(A,C\right),0\right)=1\right]-Pr\left[Expt\left(\mathrm{G}_{io}\left(A,C\right),1\right)=1\right]\right|\leq\epsilon.$$

then *iO* is an indistinguishable obfuscator. An important part of the definition of IO is that a possessor of an obfuscated version of a program is not explicitly prevented from repeatedly running the program on different inputs and, after enough time, learning all possible program outputs.

3.2 Techniques

Bitansky and Vaikuntanathan borrow techniques from *Function-Private Functional Encryption in the Private-Key Setting*, a work by Zvika Brakerski and Gil Segev. It can be said that Bitansky and Vaikuntanathan model their work on this result.

Function-Private Functional Encryption: The idea behind FPFE was given in Section 1.1 but a formal definition is important. Just like an FE, an FPFE is a set of four algorithms: Setup, Keygen, Encrypt, and Decrypt. Moreover, the syntactic requirements for these functions in a valid FPFE are identical to the requirements in an FE. See Section 2 for the specifics.

The crucial difference between an FE and an FPFE is in the definition of security. According to the informal definition from above, the goal of an FPFE is to prevent an adversary from being able to learn anything about f from FK_f = Keygen(MSK, f) or x from m = Encrypt(EK, x), besides $f(x)$. To formalize this notion, we define a guessing game $\mathrm{G}_{\mathrm{fpfe}}$ between an adversary A and a challenger C. The challenger begins by instantiating an instance of the FPFE *fpfe*. Adversary A chooses any two functions f_0 and f_1 and any two inputs x_0 and x_1 and sends those four objects to C. Challenger C selects $\mathrm{b} \leftarrow \{0,1\}$ uniformly at random and returns FK_b = Keygen(MSK, f_b) and m_b = Encrypt(EK, x_b) to A. Adversary A wins the guessing game if it can detect whether C returned a function-private function key for f_0 or f_1. To be fair to C, all of A's queries must meet the following condition:

$$f_0\left(x_0\right)=f_1\left(x_1\right).$$

If the adversary did not abide by the restriction, it could trivially win the game. Define an experiment using $\mathrm{G}_{\mathrm{fpfe}}$

$$Expt\left(G_{fpfe}\left(A,C\right),b\right)=b'$$

If A cannot win the guessing game with more than

$$\frac{1}{2}+\epsilon$$

probability (where ε defines the security of the scheme, usually in terms of the security parameter), then the scheme is considered to be *fully function private*. In other words,

$$\left|Pr\left[Expt\left(\mathrm{G}_{fpfe}\left(A,C\right),0\right)=1\right]-Pr\left[Expt\left(\mathrm{G}_{fpfe}\left(A,C\right),1\right)=1\right]\right|\leq\epsilon.$$

then *fpfe* is an FPFE. In *Function-Private Functional Encryption in the Private-Key Setting*, Brakerski and Segev describe how to construct an FPFE scheme from any existing private-key FE. Their result

provides the intuition Bitansky and Vaikuntanathan use for their construction and, maybe more importantly, their work is an excellent example of the two-key paradigm employed by Bitansky and Vaikuntanathan for the IO construction.

At a high level, Brakerski and Segev construct a FPFE for function *f* from an underlying FE *fe* through application of the two-key paradigm. In building a function key for *f*, *f*'s description is encrypted twice: first with a SKE scheme and again with the underlying FE's MSK (just as would be done by any FE Keygen). The result is a function key FK_{fpfe} that hides the description of *f* from anyone who does not possess the symmetric private-key used for the first encryption.

The underlying FE's decryption scheme is capable of reversing the encryption of FK_{fpfe} with the MSK. Without access to the symmetric private key, however, it cannot reverse the symmetric private-key encryption of *f*. Therefore, to evaluate FK_{fpfe} the FPFE's decryption function requires the input to *f* and a copy of the symmetric private-key used to encrypt it. To evaluate FK_{fpfe} on an input *c*, the function secret key given to the scheme's users decrypts *f*'s description and then applies *c* to the decrypted function.

More formally, Brakerski and Segev construct their FPFE given an underlying FE *fe* (with operations *fe*.Encrypt, *fe*.Setup, *fe*.Keygen, and *fe*.Decrypt) as follows. Besides *fe*, the construction also requires an SKE *ske*. As in any FE, their FPFE has four operations: Setup, Encrypt Decrypt, and Keygen.

- **Setup**: The Setup algorithm generates a MSK from the security parameter using the underlying FE and SKE.

$$MSK_{fpfe} = \left(fe.MSK, fe.Setup(\lambda), SK = SKE.Setup(\lambda)\right).$$

- **Keygen**: Keygen generates a function-hiding functional key for a function U_f:

$$SK_{f_{fpfe}} = Keygen\left(MSK_{fpfe}, U_f\right),$$

where U_f is related to *f*. The details of the conversion between U_f and *f* are below.

- **Encrypt**: The Encrypt algorithm allows the holder of MSK_{fpfe} to encrypt an input *x* in such a way that it can be later used in the evaluation of *f*. In addition to function input, the encrypted value must contain enough information to undo the double encryption of Keygen. In other words, it must contain *SK*. Therefore,

$$m = Encrypt\left(MSK_{fpfe} = \left(fe.MSK, SK\right), x\right) = fe.Encrypt\left(fe.MSK, \left(x, SK\right)\right)$$

- **Decrypt**: The Decrypt algorithm undoes the Encrypt operation and effectively calculates f(x):

$$y' = Decrypt\left(SK_{f_{fpfe}}, m = fe.Encrypt\left(fe.MSK, \left(SK, x\right)\right)\right) = fe.Decrypt\left(SK_{f_{fpfe}}, \left(SK, x\right)\right)$$

Although this is conceptually simple, Brakerski and Segev show that security of the construction as defined is impossible to prove. Fortunately, it is possible to prove the security of a construction almost identical to the one above. Here is Brakerski and Segev's final construction:

- **Setup**: The Setup algorithm generates a MSK from the security parameter using the underlying FE and SKE.

$$MSK_{fpfe} = (fe.MSK, fe.Setup(\lambda), SK = SKE.Setup(\lambda), SK' = SKE.Setup(\lambda)).$$

The multiple invocations of *ske*.Setup() are independent of one another.

- **Keygen**: Keygen generates a function-hiding functional key for a function

$$U_f : SK_{f_{fpfe}} = Keygen(MSK_{fpfe}, U_f)$$

where U_f is related to *f*. The details of the conversion between U_f and *f* are below.

- **Encrypt**: The Encrypt algorithm allows the holder of MSK_{fpfe} to encrypt an input *x* in such a way that it can be later used in the evaluation of *f*. In addition to function input, the encrypted value must contain enough information to undo the double encryption of Keygen and support the security proof. Therefore,

$$m = Encrypt(MSK_{fpfe} = (fe.MSK, SK, SK'), x) = fe.Encrypt(fe.MSK, (x, \varnothing, SK, \varnothing))$$

- **Decrypt**: The Decrypt algorithm undoes the Encrypt operation and effectively calculates

$$f(x): y' = Decrypt(SK_{f_{fpfe}}, m = fe.Encrypt(fe.MSK, (SK, x))) = fe.Decrypt(SK_{f_{fpfe}}, (SK, m))$$

The description of the syntactical definition of author's construction is not complete until we describe the construction of U_f from *f*. $SK_{f_{fpe}}$ generated by FPFE.Keygen is actually a functional key for U_f. Because it is constructed within the context of FPFE.Keygen, the construction can access *fe*.MSK, *SK*, *SK'* and, of course, *f*. U_f is defined as follows:

$$U_f(m, m', k, k') = \begin{cases} Dec(k, c)(m) & if\ k \neq \varnothing \\ Dec(k', c')(m') & otherwise \end{cases}$$

where

$$c = SKE.Enc(SK, f)$$
$$c' = SKE.Enc(SK', f).$$

It is important to notice that U_f reveals nothing about *f as long as SK and SK'* are kept hidden. In other words, distributing U_f, much less $SK_{f_{fpe}}$, does not compromise the secrecy of the function *f*. This is the key to the author's construction.

The FE upon which the FPFE is constructed is a private-key FE. Again, in a private-key FE only the possessor of the MSK is able to produce messages that can be used as inputs to a function key. Because it relies on the FE, the FPFE is a *private-key FPFE*. There is no value in giving a precise definition for such an object, but an intuitive understanding is important.

Look closely at Encrypt in the FPFE and see that only a person who holds the symmetric keys and *fe*'s MSK is able to create inputs to the function-hiding function key. The symmetric keys and *fe*'s MSK

make up a sort of private key for the FPFE. In order to create an input applicable to the function-hiding function key, a user must ask for their input to be encrypted. The holder of the secret key acts like generator who takes inputs and issues tokens that grant access to an evaluator. It is because of this simile that people refer to these schemes as token-based FPFEs.

4 Indistinguishability Obfuscation from Functional Encryption

Bitansky and Vaikuntanathan point out that public-key FPFE is *almost* IO. In fact, they say that "... any meaningful notion of public-key functional encryption that is *also* function-hiding would already imply some sort of obfuscation." So, is it possible to construct a public-key FPFE? Brakerski and Segev answer this in the negative. However, their answer is not the end of the story. Their assertion that public-key FPFE is impossible comes from a specific part of the definition of FPFE. In an FPFE, the user must not be able to learn anything about the function given only an input. In a public-key FPFE, the user could, conceivably, generate inputs over and over and over until he/she learns the entire function. Strictly speaking, this means that the scheme is not truly function hiding.

Recall the principles of the definition and use of IO: A user possesses the obfuscated circuit and, completely independently, can construct arbitrary inputs and run the program. Most importantly, the definition does not prevent the possessor of an obfuscated program from, given enough time, determining the program's output for every single input. This is the crucial difference between public-key FPFE and IO. IO such a scenario (where a user can learn every program output for every program input) is explicitly allowed.

Based on this and the construction and technique in Brakerski and Segev, Bitansky and Vaikuntanathan's first attempt at creating an IO from FE is a thought experiment that relies on a VBB program obfuscator and a token-based FPFE. In a world where VBB obfuscators existed, it would be possible to create IO from a token-based FPFE as follows:

1. Instantiate a private-key FPFE *fpfe*.
2. Instantiate the VBB obfuscator *O*.
3. Use *fpfe*.Keygen to generate a function-hiding function key FK_f for an input function *f*.
4. Use *O* to obfuscate *fpfe*.Encrypt()
5. Give the user FK_f and *O*(*fpfe*.Encrypt())

By definition of VBB program obfuscation, the user with access to *O*(*fpfe*.Encrypt()) would not be able to learn the secret key that hid *f* but could create his/her own inputs to *f*. In other words, the function *f* would be hidden (a criteria for IO) and the user could generate inputs independently (another criteria for IO). Unfortunately, VBB program obfuscation does not exist. This does not mean that the idea is useless. In fact, it is very nearly this exact construction that Bitansky and Vaikuntanathan define and prove to be secure.

To overcome the hurdle of not having a VBB program obfuscator, the authors recursively apply a FPFE on prefixes of the description of *fpfe*.Encrypt() itself. At each step of the recursion, there is a function-

hiding function key that outputs *n* bits of the *fpfe*.Encrypt() function using an encryption of *n-1* bits of the function as input. When the recursive procedure is invoked to generate the encryption of a single bit of the *fpfe*.Encrypt() function, the process ends. When the recursion is unrolled, there are a series of function-hiding function keys that, when applied to the output from the previous invocation, generate an obfuscated version of the *fpfe*.Encrypt().

They call this recursive obfuscator *rO.Obf* and define it on three parameters: a circuit size *n*, a circuit *C* and a security parameter λ. The recursive obfuscation process is actually general enough for the authors to put their entire construction into recursive terms. An IO for a circuit C: $\{0,1\}^n \rightarrow \{0,1\}$ is created with a single invocation *rO.Obf*(*n*, *C*, 1^λ).

4.1 Recursive Obfuscator Construction

rO.Obf takes three parameters:

1. *C*: a circuit to obfuscate,
2. *n*: the size of *C*,
3. 1^λ: the security parameter

It also requires the instantiation of three underlying cryptographic objects:

1. *fe*: public-key FE scheme with sub-exponential security
2. *ske*: a symmetric private-key encryption scheme with sub-exponential security
3. *pprf*: a PPRF

The proof that the recursive obfuscator generates an IO requires that *fe* and *ske* are required to have sub-exponential security properties. See Section 4.2 for details.

The output of *rO.Obf* is a pair (*E*, *FK*):

1. *E*: An obfuscated encryption circuit
2. *FK*: a function-hiding function key for the circuit *C'*, a circuit derived from an encrypted version of *C* (See below for details on how *C'* is related to *C*).

When *rO.Obf* is invoked to construct an IO, *E* and *FK* match tools required by the thought-experimental design outlined above. Conceptually, *E* is the VBB obfuscated token generator and *FK* is the function-hiding function key for the circuit. Passing the output of *E* on the user's input *x* to *FK* will yield the output of *C*(*x*). More precisely, to calculate *C* for some input *x*, a user simply invokes *fe*.Dec(*FK*,*E*(*x*)).

From this high level description of *rO.Obf* it is possible to define the algorithm precisely:

1. Is *C* a single bit (n = 1)?
    1. Yes: Return (*C*(0), *C*(1)).
    2. No: Construct the values required to build a recursive call to *rO.Obf*:

1. Generate a master public key and master secret key from *fe*: MPK_n, MSK_n = *fe*.Setup(1^λ).

MSK_n is used to generate a function key for *C'* (see Step 5) and MPK_n is used to generate inputs for that key (see Step 7).

2. Generate (independently from one another) two symmetric secret keys: SK_0 = *ske*.Setup(), SK_1 = *ske*.Setup().

This step parallels the **Setup** operation of the construction in Brakerski and Segev's final construction. The two symmetric keys are used in Steps 3 and 4 to hide *C* from a snooper who has access to a function key that computes that circuit.

3. Encrypt *C* using those secret keys: CT_0 = *ske*.Encrypt(SK_0, *C*), CT_1 = *ske*.Encrypt(SK_1, *C*).

This step also parallels the construction in Brakerski and Segev (the construction of U_f). The input circuit *C* is encrypted using the symmetric private keys. This has the effect of creating a circuit that is hidden from anyone who does not possess that encryption/decryption key.

4. Define the circuit *C'* using *U*, the universal circuit: *C'*(x_n, *SK*, *B*) = U(*ske*.Decrypt(*SK*, CT_B), x_n)

This step, too, parallels the definition of U_f in Brakerski and Segev's construction. The derivative circuit *C'* takes three parameters: An input, a symmetric key and a value 0 or 1. *C'* will decrypt one of the two encrypted versions of *C* (selected by the third parameter) using the symmetric key and then invoke that circuit on the input. Although the authors do not specify this in their construction, the proof of security makes it clear that both CT_0 and CT_1 are hardwired into the definition of *C'*.

5. Define *FK*: *FK* = *fe*.Keygen(MSK_n, *C'*)

Using the underlying FE, this step generates a function key for *C'*. In effect, *FK* is a function-hiding function key for *C*, when properly invoked.

6. Generate the PPRF seed K_i: $K_i = Gen_{pprf}(1^\lambda)$.

The seed K_i is used in Step 7 to generate input that helps obscure the token-generating circuit *E*.

7. Define *E*: *E* = {$E_0(x_{n-1})$, $E_1(x_{n-1})$} where E_0 = {*fe*.Encrypt(MPK_n, ([x_{n-1},0], SK_0, 0); PPRF_K_i([x_{n-1}, 0]))} and E_1 = {*fe*.Encrypt(MPK_n, ([x_{n-1},0], 1], SK_0, 0); PPRF_K_i([x_{n-1},1]))} and x_{n-1} is the first *n-1* bits of x_n.[6]

Before describing the meaning of *E*, look at its type. *E* is a pair of circuits. Each pair operates on the first *n-1* bits of x_n, an input to this invocation of *rO.Obf*. It is possible that x_n is [x_{n-1}, 0] or [x_{n-1},1]. *E0* handles the former case and *E1* handles the latter. At runtime, the circuit *E0* is used if the n^{th} bit of x_n is 0 and the circuit *E1* is used if the n^{th} bit of x_n is 1.

Since the Encrypt method for an FE scheme usually only takes two parameters (the key and the input to

6 The notation [<*bit array*>, <*bit*>] (as in [x_{n-1}, 0]) denotes a new bit array that is the concatenation of <*bit*> with <*bit array*>.

encrypt), it is odd to see a third parameter here. It is jarring but not surprising. Because this is a public-key FE scheme, the Encrypt operation has to make use of randomness (See Section 2 for the rationale and the definition of Encrypt in a public-key FE). In this construction, the authors are specifying the randomness explicitly through an invocation of a PPRF. The output from the PPRF plays an important role in the proof of security.

As described before, think of *E* as the token-generator. The circuits *E0* and *E1* will generate an input to *fe*.Decrypt that corresponds to *FK* (from Step 5). Again, if the n^{th} bit of x_n is 0, then *E0* is used and if the n^{th} bit of x_n is 1, then *E1* is used. When the selected circuit is invoked with the given input, the output will be a value that can be paired with *FK* and passed to *fe*.Decrypt. The result of the decryption will be, in effect, the output of *C* on the input x_n – exactly what we want!

8. Define E_o, and obfuscation of the circuit *E*: *rO.Obf*(*n*-1, E_o, 1^λ)

Using a recursive invocation *rO.Obf* the algorithm generates E_o, an obfuscated version of the token-generating circuit *E*. With respect to Brakerski and Segev's construction, E_o is the VBB obfuscation of *E*.

9. Output (E_o, *FK*)

4.2 Proof

Contrary to the intricacy of the construction, the proof is relatively straightforward. Recall the guessing game G_{io} defined above and the experiment that captures the meaning of indistinguishable obfuscation:

$$Pr\left[Expt\left(\mathrm{GG}_{io}\left(A,C\right),0\right)=1\right]-Pr\left[Expt\left(\mathrm{GG}_{io}\left(A,C\right),1\right)=1\right]\leq\epsilon\ .$$

The adversary in G_{io} chooses C_0 and C_1 from a class {C} that computes the same program and gives them to the challenger. The challenger constructs C_b as

$$C_b=rO.Obf\left(n,C_b,\lambda\right).$$

Therefore, the adversary has to be able to distinguish between *rO.Obf*(*n*,C_0, *λ*) and *rO.Obf*(*n*,C_1, *λ*) with some nonneglibible probability. Define *D* to be a polynomial time distinguisher that takes an obfuscated circuit and returns a single bit. The output bit is 1 if the obfuscated circuit is based on C_0 and the output bit is 0 otherwise. Given such a distinguisher, the guessing game definition of IO can be restated:

$$Pr\left[D\left(io\left(C_0\right)\right)=1\right]-Pr\left[D\left(io\left(C_1\right)\right)=1\right]\leq\epsilon\ .$$

By definition *iO*(*C0*) = *rO.Obf*(*n*, *C0*, *λ*)and *iO*(*C1*) = *rO.Obf*(*n*, *C1*, *λ*). Therefore, it is possible to bound their difference (in the eye's of the distinguisher *D*) based on the underlying construction. According to the construction, the output of *rO.Obf*(*n*, C_b, *λ*) can be altered in the following ways:

1. Embed SK_1 instead of using SK_0 in Step 7. SK_1 will be used as the key for decrypting the enciphered copy of *C* embedded in *C'* and encrypted, ultimately, in *FK*. Name this parameter α.
2. Choose to redefine the inputs to SKE.Encrypt() in Step 3 so that $CT_0^{\varphi_0}$ and $CT_0^{\varphi_1}$ are

ciphertexts of two different circuits, C_{φ_0} and C_{φ_1}, respectively, where φ_0 and φ_1 are either {0,1}. Note that even though the construction calls for φ_0 to equal φ_1 in all cases (because the construction is really just encrypting the same circuit with different keys), this does not need to be the case here.

Through a combination of these three parameters, α, φ_0 and φ_1, it is possible to define a set of hybrids H(α, φ_0, φ_1). H(0, 0, 0) corresponds to an obfuscation of C_0 and H(0,1, 1) corresponds to an obfuscation of C_1. Define a distinguisher D_H that can distinguish between the different hybrid worlds. If it is possible to prove that

$$\left(\delta = \left| Pr\left[D_H\left(H\left(0,0,0\right)\right)=1\right] - Pr\left[D_H\left(H\left(0,1,1\right)\right)\right]\right| \right) \leq \epsilon$$

then the construction is an IO. For the remainder of the review of the proof, we will refer to δ as the distance between two hybrid worlds in the eyes of D_H.

How then, can we get from H(0,0,0) to H(0,1,1) in a way that we can calculate the actual difference in probability that D_H can distinguish between the worlds?

1. H(0,0,0) and H(0,0,1): The difference between these worlds is in φ_1. By implication, the distance between the worlds is bounded by the security of the SKE *ske*.
2. H(0,0,1) and H(1,0,1): The difference between these worlds is in α.
3. H(1,0,1) and H(1,1,1): The difference between these worlds is in φ_0. By implication, the "distance between the worlds is bounded by the security of the SKE *ske*.
4. H(1,1,1) and H(0,1,1): The difference between these worlds is in α.

To effectively calculate the distance between the worlds in (2) and (4) requires another hybrid argument. The difference between the hybrid worlds in these equations is due to the construction of E_o. As a result, the difference is really the accumulation of the distances between the *n-1* different outputs that result from the use of the recursive obfuscator to concoct that obfuscated circuit.

It is here that the requirement of sub-exponential security of the SKE *ske* is important. Because they are n-1 different distances to accumulate, if *ske* had only polynomial time security, the distances would accumulate too quickly and exceed the limits required for IO.

However, thanks to the properties of the PPRF used to generate the randomness used by *fe*.Encrypt[7] and the sub-exponential security properties of the the SKE *ske*, the authors are able to prove an upper-bound on the difference between the hybrid worlds H(0,0,1) and H(1,0,1) and H(1,1,1) and H(0,1,1).

Algebraic manipulation of the differences in the probability of success of D_H for the different hybrids leads that author to conclude that their construction is an IO.

7 Independently creating of two E_os with separate seeds from the PPRF is like creating public samplers that cannot be distinguished. The argument and proof are formalized by [CLTV15].

5 Future Directions and Open Questions

Although this result is impressive, there are several paths for improvement. First and foremost, it would be great to show that the underlying cryptographic objects in the authors' construction did not need to have sub-exponential security. Second, the authors report that it would be an improvement if the construction could use a private-key FE scheme since this would obviate the need for employing a PPRF.

6 Conclusion

Program obfuscation, of which IO is a type, gives software developers a powerful practical tool for confidently distributing their programs knowing that proprietary implementation details and algorithms are hidden from prying eyes, preventing circumvention of DRM schemes and defeating piracy, releasing patches with the assurance that hackers could not reverse engineer the update to infer the security vulnerability.

Program obfuscation is also a useful academic tool, beneficial to high-level cryptographic operations use it as a primitive. Researchers have shown that it offers the requisite power to implement public key cryptography [BB01]! Even a weak notion of program obfuscation like IO is a useful cryptographic primitive, “powerful enough to give rise to almost any known cryptographic object” [BV15].

In *Indistinguishability Obfuscation from Functional Encryption*, Bitansky and Vaikuntanathan propose an IO scheme constructed from FE primitives. Their result represents an advance in the state-of-the-art to cryptography researchers who have searched for IO constructions built on standard cryptographic assumptions. By describing an IO construction from FE, Bitansky and Vaikuntanathan show that FE implies IO and, therefore, establish their equivalence (“up to some sub-exponential loss”).

Dictionary of Acronyms

FE: Functional encryption
IO: Indistinguishability obfuscation
FPFE: Function private functional encryption
SKE: Symmetric key encryption
VBB: Virtual Black Box
PPRF: Puncturable Pseudorandom Function
IBE: Identity-based Encryption
ABE: Attribute-based Encryption

Works Cited

BB01 Barak, Boaz, et al. "On the (im) possibility of obfuscating programs." *Advances in cryptology—CRYPTO 2001*. Springer Berlin Heidelberg, 2001.

BSW15 Boneh, Dan, Amit Sahai, and Brent Waters. "Functional encryption: Definitions and challenges." *Theory of Cryptography*. Springer Berlin Heidelberg, 2011. 253-273.

BS14 Brakerski, Zvika, and Gil Segev. *Function-private functional encryption in the private-key setting*. Cryptology ePrint Archive, Report 2014/550, 2014. http://eprint. iacr. Org, 2014.

BV15 Bitansky, Nir, and Vinod Vaikuntanathan. "Indistinguishability Obfuscation from Functional Encryption." (2015).

CLTV15 Canetti, R.; Lin, H.; Tessaro, S. & Vaikuntanathan, V. Dodis, Y. & Nielsen, J. (Eds.) “Obfuscation of Probabilistic Circuits and Applications” *Theory of Cryptography.* Springer Berlin Heidelberg, 2015. 468-497 .

GGH13 Garg, S.; Gentry, C.; Halevi, S.; Raykova, M.; Sahai, A.; Waters, B., "Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits," *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on* , vol., no., pp.40,49, 26-29 Oct. 2013 doi: 10.1109/FOCS.2013.13

KL07 Jonathan Katz and Yehuda Lindell. 2007. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC.