Code Obfuscation

- Encrypting programs, maintaining functionality
  - Only the functionality should be “visible” in the output
- Example of recreational obfuscation:

```perl
@P=split//,".URRUU\c8R";@d=split//,"\nrekcuh xinU /
leP rehtona tsuJ";sub p{
  @p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+
  =$f=!fork;map{$P=$P[$f^ord ($p{$_})&6];$p{$_}=/
  ^$P/ix?$P:close$_}keys%p;p;p;p;p;map{$p{$_}=~/^[P
  .]/&& close$_}p;wait
until?$;map{/^r/&<&$>}p;$_=$d[$q];sleep
rand(2)if/\S/;print
```

-- Wikipedia, accessed Oct-2013

- Rigorous treatment [Hada’00, BGIRSVY’01,…]
Why Obfuscation?

- Hiding secrets in software
- AES encryption
Why Obfuscation?

- Hiding secrets in software

Plaintext

\@P=spli\//",".URRUU\c8R";\@d=spli\//"\nrekcah
xinU/\lreP rehtona tsu];\textbf{sub} \texttt{p} {
@p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=$f=!fork;map\{\$P=\$P[\$f\^ord
($p{$_})&6];\$p{$_}=/
^\$P/ix?$P:close$_}keys%p)p;p;p;p;map\{\$p{$_}=~/^[P.]/&& close$_}%p;wait
until$_;map\{/^r/&&$_>\}p;$_=$d[q];sleep
rand(2)if/\S/;print

AES encryption $\rightarrow$ Public-key encryption
Why Obfuscation?

- Hiding secrets in software

Game of Go

- Uploading my expertise to the web

Two Celestial Beings playing a game of Go to determine the fate of the universe

Why Obfuscation?

- Hiding secrets in software

- Uploading my expertise to the web without revealing my strategies
Defining Obfuscation

- An efficient public procedure $\text{OBF}(\ast)$
- Takes as input a program $C$
  - E.g., encoded as a circuit
- Produce as output another program $C'$
  - $C'$ computes the same function as $C$
  - $C'$ at most polynomially larger than $C$

- How to define security?
Defining Obfuscation Security

• Want the output to reveal **only functionality**
  • E.g., If prog. depends on secrets that are not readily apparent in I/O, then the encrypted program does not reveal these secrets

• [B+01] show that this is **impossible** in general
  • Thm: If secure encryption exists, then there are secure encryption schemes for which it is possible to recover the secret key from any program that encrypts.
    • Such encryption schemes are **unobfuscatable**
Defining Obfuscation

• Okay, some function are bad, but can we do “as well as possible” on every given function?

• [B+01] suggested the weaker notion of “indistinguishability obfuscation” (iO)
  • Gives the “best-possible” guarantee
  • Turns out to suffice for many applications
Indistinguishability Obfuscation

- **Def:** If $C_1, C_2$ compute the same function (and $|C_1| = |C_2|$) then $O(C_1) \approx O(C_2)$
  - Indistinguishable even if you know $C_1, C_2$

- **Note:** Inefficient $iO$ is always possible
  - $O(C) =$ lexicographically $1^{st}$ circuit computing the same function as $C$

- Canonicalization is inefficient (unless $P=NP$)
Best-Possible Obfuscation [GR07]

Indist. Obfuscation

Best Obfuscation

Some circuit $C$

$C(x)$

≈

Computationally Indistinguishable

Indist. Obfuscation

Padding

Some circuit $C$

$C(x)$
Many Applications of iO

- AES $\Rightarrow$ public key encryption [GGH+13, SW13]
- Witness encryption: Encrypt $x$ so only someone with proof of Riemann Hypothesis can decrypt [GGSW13]
- Functional encryption: Noninteractive access control [GGH+13], $\text{Dec}(\text{Key}_y, \text{Enc}(x)) \Rightarrow F(x, y)$
- Many more (all since last year)...
- One notable thing iO doesn’t give us (yet): Homomorphic Encryption (FHE)
Beyond \(iO\)

- For very few functions, we know how to achieve stronger notions than \(iO\)
  - “Virtual Black Box” (VBB)
- Point-functions / cryptographic locks
  \[
  f_{a,b}(x) = \begin{cases} 
  b & \text{if } x = a \\
  \bot & \text{otherwise}
  \end{cases}
  \]
- \([C97, \text{CMR98, LPS04, W05}]\)
- Many extensions, generalizations \([\text{DS05, AW07, CD08, BC10, HMLS10, HRSV11, BR13}]\)
Obfuscation vs. HE

- Somewhat reminiscent of MMAPs vs. HE…
Obfuscation from MMAPs, 1st Try

• MMAPs give us one function that we can get in the clear: equality-to-zero (at level \( k \))
  • Can we build on it to get more functions?

• Consider the “universal circuit”
  • \( U: \{0, 1\}^{n'} \times \{0, 1\}^n \rightarrow \{0, 1\}, \ U(C, x) = C(x) \)
  • Encode the bits of \( C \) at level 1
  • For \( x \), provide level-1 encoding of both 0 and 1
  • Carry out the computation “in the exponent”
  • Use zero-test to check if the result is zero
1st Try Does Not Work

- Attack: comparing intermediate values
  - Checking if two intermediate wires carry the same value
  - Checking if the computation on two different inputs yield the same value on some intermediate wire
- If two equal intermediate values ever happen, they can be recognized using zero-test
- Must randomize all intermediate values in all the computations
  - But such that the final result can still be recognized
Obfuscating Arbitrary Circuits

- A two-step construction
- Obfuscating “shallow circuits” (NC1)
  - This is where the meat is
  - Using multilinear maps
  - Security under a new (ad-hoc) assumption
- Bootstrapping to get all circuits
  - Using homomorphic encryption with NC1 decryption
  - Very simple, provable, transformation
NC¹ Obfuscation $\Rightarrow$ P Obfuscation

Diagram:
- Input: $F$
- Homomorphic Encryption
- Process: $x$, $F(x)$
- Output: $F(x)$

Explanation:
If $\pi$ describes homomorphic evaluation that takes $x,F$ to $c$, then use $sk$ to decrypt $c$.
NC¹ Obfuscation $\rightarrow$ P Obfuscation

Output of P obfuscator

Homomorphic Encryption

F

\[ F(\text{x}) \]

x

+ $\rightarrow$ F(x)

Encrypted-result \( C \) + eval transcript \( \pi \)

@P=split//",".URRUU\c8R \
;@d=split//","\nrekcah \
xinU / lreP rehtona tsuJ";\textbf{sub} \
p{ @p{"r$p"…

CondDec

F(x)
Conditional Decryption with \textit{iO}

- We have \textit{iO}, not “perfect” obfuscation

- But we can adapt the CondDec approach
  - We use \textit{two} HE secret keys
iO for CondDec $\rightarrow$ iO for All Circuits

\[ \pi, x, \text{ and two ciphertexts } c_0 = \text{Enc}_{\text{PK}_0}(F(x)) \text{ and } c_1 = \text{Enc}_{\text{PK}_1}(F(x)) \]

\[ \approx \]

\[ \pi, x_i \text{'s, and two ciphertexts } c_0 = \text{Enc}_{\text{PK}_0}(F(x)) \text{ and } c_1 = \text{Enc}_{\text{PK}_1}(F(x)) \]
Analysis of Two-Key Technique

- 1st program has secret SK0 inside (not SK1).
- 2nd program has secret SK1 inside (not SK0).
- But programs are indistinguishable
- So, neither program “leaks” either secret.

- Two-key trick is very handy in iO context.
- Similar to Naor-Yung ’90 technique to get encryption with chosen ciphertext security
$\text{NC}^1$ Obfuscation
Construction Outline

- Describe Circuits as **Branching Programs** (BPs) using Barrington’s theorem [B86]

- **Randomized** BPs (RBPs) a-la-Kilian [K88]
  - Additional randomization to counter “simple relations”

- Encode RBPs “in the exponent” using MMAPs
  - Use zero-test to get the output
(Oblivious) Branching Programs

- A specific way of describing a function
- This length-9 BP has 4-bit inputs

The $A_{i,b}$'s are square matrices
Each position $i$ is controlled by one input bit
(Oblivious) Branching Programs

- A specific way of describing a function
- This length-9 BP has 4-bit inputs
(Oblivious) Branching Programs

- A specific way of describing a function
- This length-9 BP has 4-bit inputs

```
A_{1,0}  A_{2,0}  A_{3,0}  A_{4,0}  A_{5,0}  A_{6,0}  A_{7,0}  A_{8,0}  A_{9,0}
A_{1,1}  A_{2,1}  A_{3,1}  A_{4,1}  A_{5,1}  A_{6,1}  A_{7,1}  A_{8,1}  A_{9,1}
```

```
0 1 1 1 0
```

- Multiply the chosen 9 matrices together
  - If product is 1 output 1. Otherwise output 0.
(Oblivious) Branching Programs

- A specific way of describing a function
- Length-$m$ BP with $n$-bit input is a sequence $(j_1, A_{1,0}, A_{1,1}), (j_2, A_{2,0}, A_{2,1}), \ldots, (j_m, A_{m,0}, A_{m,1})$
  - $j_i \in \{1, \ldots, n\}$ are indexes, $A_{i,b}$'s are matrices
- Input $x = (x_1, \ldots, x_n)$ chooses matrices $A_{i,x_{j_i}}$
  - Compute the product $P_x = \prod_{i=1}^{m} A_{i,x_{j_i}}$
  - $F(x) = 1$ if $P_x = I$, else $F(x) = 0$

- Barrington’s Theorem [B86]: Poly-length BPs can compute any function in NC1
Kilian’s Randomized BPs

- Choose random invertible matrices $R_i$
  - $B_{i,\sigma} = R_{i-1}^{-1} \times A_{i,\sigma} \times R_i$ (with $R_0 = R_{m+1} = I$)
- Multiplying the $B_{i,\sigma}$’s yields the same product as multiplying the corresponding $A_{i,\sigma}$’s
Kilian’s Randomized BPs

- Each sequence of $B_{i,\sigma}$’s ($i = 1 \ldots m$) is uniformly random subject to having same product as $A_{i,\sigma}$’s
Kilian’s Protocol ➔ BP-Obfuscation?

- Encode RBP for $U(C, x)$ with the $C$ part fixed
  - Publish only one $B_{i,\sigma}$ for the bits of $C$, both $B_{i,\sigma}$’s for $x$
  - Can evaluates $U(C, x)$ for any $x$.
- “Simple relations” exist between the $B_{i,\sigma}$’s
  - Since we give both $B_{i,\sigma}$’s for some $i$’s
- Enable some attacks
  - “Partial-evaluation” attacks: Compare partial products across different evaluations
  - “Mixed-Input” attacks: Compute products that do not respect the BP structure
“Partial Evaluation” Attacks

- Evaluate the program on two inputs $x, x'$, but only multiply matrices from step $i_1$ to $i_2$,
  \[ P = \prod_{i=i_1}^{i_2} B_{i,x_{ji}}, \quad P' = \prod_{i=i_1}^{i_2} B_{i,x_{ji}} \]

- Check if $P = P'$

- Roughly, you learn if in the computations of the circuits for $C(x), C(x')$, you have the same value on some internal wire
“Mixed Input” Attack

- Inconsistent matrix selection:
  - Product includes e.g., $B_{2,0}$ and $B_{4,1}$
  - These two steps depend on the same input bit
- Roughly, you learn what happens when fixing some internal wire in the circuit of $C(x)$
  - Fixing the wire value to 0, or to 1, or copying value from another wire, …
Countering “Simple Relations”

- Additional randomization steps
- Different works use slightly different forms of additional randomization
  - “Multiplicative bundling” [GGHRHS’13, BR’13]
  - “Straddling” [BGKPS’13, PTS’14]
  - “Abelian component” [CV’13]
- Can conjecture [GGHRHS’13, BR’13] or prove [BGKPS’13, CV’13, PTS’14] that no simple relations exist
Completing the construction

- Put randomized matrices “in the exponent”
  - Set multi-linearity parameter to $m$
  - Encode all elements at level 1
  - Publish one matrix for the bits of $C$, both for $x$

- To compute $C(x)$
  - Choose the encoded matrices corresponding to $x$
  - Multiply matrices “in the exponent” using $\odot,\oplus$
  - Use zero-test to check if the result is the identity
Security of Obfuscation

- No polynomial relation of degree $\leq m$, except those that involve the output matrix
  - Output relations are the same when we obfuscate two circuits with identical functionality
- By (generalized) über-assumption, the two distributions over encodings are indistinguishable
- Mission Accomplished
Summary

- We can obfuscate a computation by:
  1. Randomizing the internal values
  2. Putting the randomized values “in the exponent” and computing on them using MMAPs