How to Obfuscate Programs Directly

Joe Zimmerman

Program Obfuscation

– Goal: make a program P “unintelligible” while preserving its functionality [BGI+01]

Program Obfuscation

- Goal: make a program P “unintelligible” while preserving its functionality [BGI+01]

- Extremely powerful primitive

Program Obfuscation

– Goal: make a program P “unintelligible” while preserving its functionality [BGI+01]

– Extremely powerful primitive

– Gold standard: <u>Virtual Black-Box (VBB) security</u> [BGI+01]
  – Obfuscated program O(P) no better than oracle access

Program Obfuscation

- Goal: make a program P "unintelligible" while preserving its functionality [BGI+01]

- Extremely powerful primitive

- Gold standard: Virtual Black-Box (VBB) security [BGI+01]
  - Obfuscated program O(P) no better than oracle access

- Weaker definition: indistinguishability obfuscation (*iO*) [BGI+01]

Program Obfuscation

– Several candidate obfuscators known
    [GGH+13b, BR14, BGK+14, PST14, GLSW14, AGIS14, MSW14]

Program Obfuscation

– Several candidate obfuscators known
  [GGH+13b, BR14, BGK+14, PST14, GLSW14, AGIS14, MSW14]

  – Fundamental building block: multilinear maps
    [BS03, GGH13a, CLT14, GGH14, CLT15]

  – VBB security in generic multilinear map model
    [GGH+13b, BGK+13]

Program Obfuscation

- In all known constructions, cost remains astronomical
    - Can be improved for some specific function families [LPS04, Wee05, CD08, CRV10, AGIS14, SZ14]

Program Obfuscation

– In all known constructions, cost remains astronomical

  – Can be improved for some specific function families [LPS04, Wee05, CD08, CRV10, AGIS14, SZ14]

– Obstacles to efficient general-purpose obfuscation:

  1. Circuits must be converted to matrix branching programs: depth d → size 4^d [Bar86]

Program Obfuscation

– In all known constructions, cost remains astronomical

  – Can be improved for some specific function families [LPS04, Wee05, CD08, CRV10, AGIS14, SZ14]

– Obstacles to efficient general-purpose obfuscation:

1. Circuits must be converted to matrix branching programs: depth d → size 4^d [Bar86]

2. Known multilinear maps have "noise" parameter that grows with the degree

This work

– In all known constructions, cost remains astronomical

– Can be improved for some specific function families [LPS04, Wee05, CD08, CRV10, AGIS14, SZ14]

– Obstacles to efficient general-purpose obfuscation:

~~1. Circuits must be converted to matrix branching programs: depth d → size 4^d [Bar86]~~

2. Known multilinear maps have “noise” parameter that grows with the degree

This work

- New construction:
  - Obfuscate general circuits *directly*
  - No matrix branching programs

This work

- New construction:
    - Obfuscate general circuits *directly*
    - No matrix branching programs
- Evaluation of obfuscated circuit mirrors structure of original

This work

- New construction:
  - Obfuscate general circuits *directly*
  - No matrix branching programs
- Evaluation of obfuscated circuit mirrors structure of original
- Number of ring ops polynomial in original circuit size

This work

- New construction:
  - Obfuscate general circuits *directly*
  - No matrix branching programs
- Evaluation of obfuscated circuit mirrors structure of original
- Number of ring ops polynomial in original circuit size

	Degree of multilinearity	Obfuscation size (# ring elements)	Evaluation time (# ring operations)
Via Barrington's Thm. [GGH+13b, BR14, BGK+14]	$O(4^d n + n^2)$	$O(4^d n + n^2)$	$O(4^d n + n^2)$
[AGIS14]	$O(2^d n + n^2)$	$O(8^d n + n^2)$	$O(8^d n + n^2)$
[AGIS14] + [Gie01]	$O(2^{(1+\varepsilon)d} n + n^2)$	$O(2^{(1+\varepsilon)d} 4^{2/\varepsilon} n + n^2)$	$O(2^{(1+\varepsilon)d} 4^{2/\varepsilon} n + n^2)$
This work	$O(2^d n + n^2)$	$\boldsymbol{O(d^2 s^2 + n^2)}$	$\boldsymbol{O(d^2 s^2 + n^2)}$

Performance for circuits of input length n, size s, and depth d.

This work

- New construction:
  - Obfuscate general circuits *directly*
  - No matrix branching programs
- Evaluation of obfuscated circuit mirrors structure of original
- Number of ring ops polynomial in original circuit size
- Prove VBB obfuscation in generic model

	Degree of multilinearity	Obfuscation size (# ring elements)	Evaluation time (# ring operations)
Via Barrington's Thm. [GGH+13b, BR14, BGK+14]	$O(4^d n + n^2)$	$O(4^d n + n^2)$	$O(4^d n + n^2)$
[AGIS14]	$O(2^d n + n^2)$	$O(8^d n + n^2)$	$O(8^d n + n^2)$
[AGIS14] + [Gie01]	$O(2^{(1+\varepsilon)d} n + n^2)$	$O(2^{(1+\varepsilon)d} 4^{2/\varepsilon} n + n^2)$	$O(2^{(1+\varepsilon)d} 4^{2/\varepsilon} n + n^2)$
This work	$O(2^d n + n^2)$	$\boldsymbol{O(d^2 s^2 + n^2)}$	$\boldsymbol{O(d^2 s^2 + n^2)}$

Performance for circuits of input length n, size s, and depth d.

This work

- For “noisy” maps [GGH13a, CLT13, GGH14, CLT15]: concrete efficiency improvements
  - Due to noise, cost of ring operations is still exp(d)
  - Still requires FHE for P/poly

This work

- For “noisy” maps [GGH13a, CLT13, GGH14, CLT15]: concrete efficiency improvements
  - Due to noise, cost of ring operations is still exp(d)
  - Still requires FHE for P/poly
- For “clean” maps (open problem): obfuscation for P/poly would now be practical!
  - “Noise” not inherent – central open problem

This work

- For “noisy” maps [GGH13a, CLT13, GGH14, CLT15]: concrete efficiency improvements
  - Due to noise, cost of ring operations is still exp(d)
  - Still requires FHE for P/poly

- For “clean” maps (open problem): obfuscation for P/poly would now be practical!
  - “Noise” not inherent – central open problem

- Concurrent work:
  - [AB15]: also obfuscates circuits without converting to branching programs; achieves iO in generic model

Background: multilinear maps

– Fundamental tool for obfuscation

Background: multilinear maps

– Fundamental tool for obfuscation

– Asymmetric, composite-order [CLT13, GLW14, CLT15]

Background: multilinear maps

– Fundamental tool for obfuscation

– Asymmetric, composite-order [CLT13, GLW14, CLT15]

– Modulus $N = N_1 \ldots N_k$, multi-set U of formal indices

Background: multilinear maps

– Fundamental tool for obfuscation

– Asymmetric, composite-order [CLT13, GLW14, CLT15]

– Modulus $N = N_1 \dots N_k$, multi-set U of formal indices

– Supports the following operations:

Background: multilinear maps

- Fundamental tool for obfuscation
- Asymmetric, composite-order [CLT13, GLW14, CLT15]
- Modulus $N = N_1 \ldots N_k$, multi-set U of formal indices
- Supports the following operations:

  Setup() → (pp, sp)

Background: multilinear maps

– Fundamental tool for obfuscation

– Asymmetric, composite-order [CLT13, GLW14, CLT15]

– Modulus $N = N_1 \dots N_k$, multi-set U of formal indices

– Supports the following operations:

Setup() → (pp, sp)

Encode(sp, x, S) → $[x]_S$ (where $S \subset U$)

Background: multilinear maps

– Fundamental tool for obfuscation

– Asymmetric, composite-order [CLT13, GLW14, CLT15]

– Modulus $N = N_1 \dots N_k$, multi-set U of formal indices

– Supports the following operations:

Setup() → (pp, sp)

Encode(sp, x, S) → $[x]_S$ (where $S \subset U$)

Add(pp, $[x]_S$, $[y]_S$) → $[x+y]_S$ (arithmetic mod N)

Background: multilinear maps

- Fundamental tool for obfuscation
- Asymmetric, composite-order [CLT13, GLW14, CLT15]
- Modulus $N = N_1 \dots N_k$, multi-set U of formal indices
- Supports the following operations:

  Setup() → (pp, sp)

  Encode(sp, x, S) → $[x]_S$ (where $S \subset U$)

  Add(pp, $[x]_S$, $[y]_S$) → $[x+y]_S$ (arithmetic mod N)

  Mult(pp, $[x]_S$, $[y]_T$) → $[xy]_{S \cup T}$ (where $S \cup T \subset U$)

Background: multilinear maps

– Fundamental tool for obfuscation

– Asymmetric, composite-order [CLT13, GLW14, CLT15]

– Modulus $N = N_1 \ldots N_k$, multi-set U of formal indices

– Supports the following operations:

Setup() $\rightarrow$ (pp, sp)

Encode(sp, x, S) $\rightarrow$ $[x]_S$ (where $S \subset U$)

Add(pp, $[x]_S$, $[y]_S$) $\rightarrow$ $[x+y]_S$ (arithmetic mod N)

Mult(pp, $[x]_S$, $[y]_T$) $\rightarrow$ $[xy]_{S \cup T}$ (where $S \cup T \subset U$)

ZeroTest(pp, $[x]_U$) $\rightarrow$ “zero” if x = 0,
“nonzero” otherwise

Background: multilinear maps

- Fundamental tool for obfuscation
- Asymmetric, composite-order [CLT13, GLW14, CLT15]
- Modulus $N = N_1 \dots N_k$, multi-set U of formal indices
- Supports the following operations:

"ST"
(product notation)

Setup() $\rightarrow$ (pp, sp)

Encode(sp, x, S) $\rightarrow$ $[x]_S$ (where $S \subset U$)

Add(pp, $[x]_S$, $[y]_S$) $\rightarrow$ $[x+y]_S$ (arithmetic mod N)

Mult(pp, $[x]_S$, $[y]_T$) $\rightarrow$ $[xy]_{S \cup T}$ (where $S \cup T \subset U$)

ZeroTest(pp, $[x]_U$) $\rightarrow$ "zero" if $x = 0$,
"nonzero" otherwise

Background: multilinear maps

– Example:

$$x = \text{Encode}(sp, 2, AB) = [2]_{AB}$$

Background: multilinear maps

– Example:

$x = \mathrm{Encode}(sp, 2, AB) = [2]_{AB}$

$y = \mathrm{Encode}(sp, 3, BC) = [3]_{BC}$

Background: multilinear maps

– Example:

$x = \text{Encode}(sp, 2, AB) = [2]_{AB}$

$y = \text{Encode}(sp, 3, BC) = [3]_{BC}$

$z = \text{Encode}(sp, -6, AB^2C) = [-6]_{AB^2C}$

Background: multilinear maps

– Example:

$x = \text{Encode}(sp, 2, AB) = [2]_{AB}$

$y = \text{Encode}(sp, 3, BC) = [3]_{BC}$

$z = \text{Encode}(sp, -6, AB^2C) = [-6]_{AB^2C}$

$w = \text{Mult}(pp, x, y) = [2]_{AB} * [3]_{BC} = [6]_{AB^2C}$

Background: multilinear maps

– Example:

x = Encode(sp, 2, AB) = $[2]_{AB}$

y = Encode(sp, 3, BC) = $[3]_{BC}$

z = Encode(sp, -6, AB^2C) = $[-6]_{AB^2C}$

w = Mult(pp, x, y) = $[2]_{AB} * [3]_{BC} = [6]_{AB^2C}$

ZeroTest(pp, Add(pp, z, w)) = “zero”

Background: multilinear maps

– Example:

x = Encode(sp, 2, AB) = $[2]_{AB}$

y = Encode(sp, 3, BC) = $[3]_{BC}$

z = Encode(sp, -6, AB^2C) = $[-6]_{AB^2C}$

w = Mult(pp, x, y) = $[2]_{AB} * [3]_{BC} = [6]_{AB^2C}$

ZeroTest(pp, Add(pp, z, w)) = “zero”

(assuming AB^2C is the top-level index set U)

Background: multilinear maps

– <u>Security definition</u>:

– Intuitively, encodings $[x]_S$ hide original scalars x in Z_N

– Formally: generic model only exposes map operations [GGH+13a, BR13, BGK+13]

– See paper for details

Background: multilinear maps

- Security definition:
  - Intuitively, encodings $[x]_S$ hide original scalars x in Z_N
  - Formally: generic model only exposes map operations [GGH+13a, BR13, BGK+13]
  - See paper for details

- Note: zero-testing only possible at the top index set U
  - Stronger model: adversary can zero-test anywhere

Background: multilinear maps

- Security definition:
  - Intuitively, encodings $[x]_S$ hide original scalars x in Z_N
  - Formally: generic model only exposes map operations [GGH+13a, BR13, BGK+13]
  - See paper for details

- Note: zero-testing only possible at the top index set U
  - Stronger model: adversary can zero-test anywhere
  - Captures deterministic encodings

Background: multilinear maps

– Security definition:

– Intuitively, encodings $[x]_S$ hide original scalars x in Z_N

– Formally: generic model only exposes map operations [GGH+13a, BR13, BGK+13]

– See paper for details

– Note: zero-testing only possible at the top index set U

– Stronger model: adversary can zero-test anywhere

– Captures deterministic encodings

– Generic transformation [BWZ14]:

Background: multilinear maps

- <u>Security definition</u>:
  - Intuitively, encodings $[x]_S$ hide original scalars x in Z_N
  - Formally: generic model only exposes map operations [GGH+13a, BR13, BGK+13]
  - See paper for details
- <u>Note</u>: zero-testing only possible at the top index set U
  - Stronger model: adversary can zero-test anywhere
  - Captures deterministic encodings
  - Generic transformation [BWZ14]:
    - If M is secure with zero-testing only at U, then M' is secure with arbitrary zero-testing

Background: multilinear maps

- Security definition:
  - Intuitively, encodings $[x]_S$ hide original scalars x in Z_N
  - Formally: generic model only exposes map operations [GGH+13a, BR13, BGK+13]
  - See paper for details
- Note: zero-testing only possible at the top index set U
  - Stronger model: adversary can zero-test anywhere
  - Captures deterministic encodings
  - Generic transformation [BWZ14]:
    - If M is secure with zero-testing only at U, then M' is secure with arbitrary zero-testing
    - M → M' adds only two components to modulus N

Background: multilinear maps

– We will make essential use of composite order

Background: multilinear maps

– We will make essential use of composite order

– Direct Product Notation:

For encodings $[x]_S$, with x in $Z_N = Z_{N_1 \ldots N_k}$:

write as $[x_1, \ldots, x_k]_S$, where $x = x_i \pmod{N_i}$ by CRT

Background: multilinear maps

– We will make essential use of composite order

– <u>Direct Product Notation:</u>

For encodings $[x]_S$, with x in $Z_N = Z_{N_1 \ldots N_k}$:

write as $[x_1, \ldots, x_k]_S$, where $x = x_i$ (mod N_i) by CRT

– Addition and multiplication operate componentwise

Background: multilinear maps

– We will make essential use of composite order

– <u>Direct Product Notation:</u>

For encodings $[x]_S$, with x in $Z_N = Z_{N_1 \ldots N_k}$:

write as $[x_1, \ldots, x_k]_S$, where $x = x_i \pmod{N_i}$ by CRT

– Addition and multiplication operate componentwise

– ZeroTest(pp, $[x_1, \ldots, x_k]_U$) = "zero" $\leftrightarrow$ $x_i = 0$ for all i

Background: multilinear maps

– We will make essential use of composite order

– <u>Direct Product Notation:</u>

For encodings $[x]_S$, with x in $Z_N = Z_{N_1 \ldots N_k}$:

write as $[x_1, \ldots, x_k]_S$, where $x = x_i \pmod{N_i}$ by CRT

– Addition and multiplication operate componentwise

– ZeroTest(pp, $[x_1, \ldots, x_k]_U$) = "zero" $\leftrightarrow$ $x_i = 0$ for all i

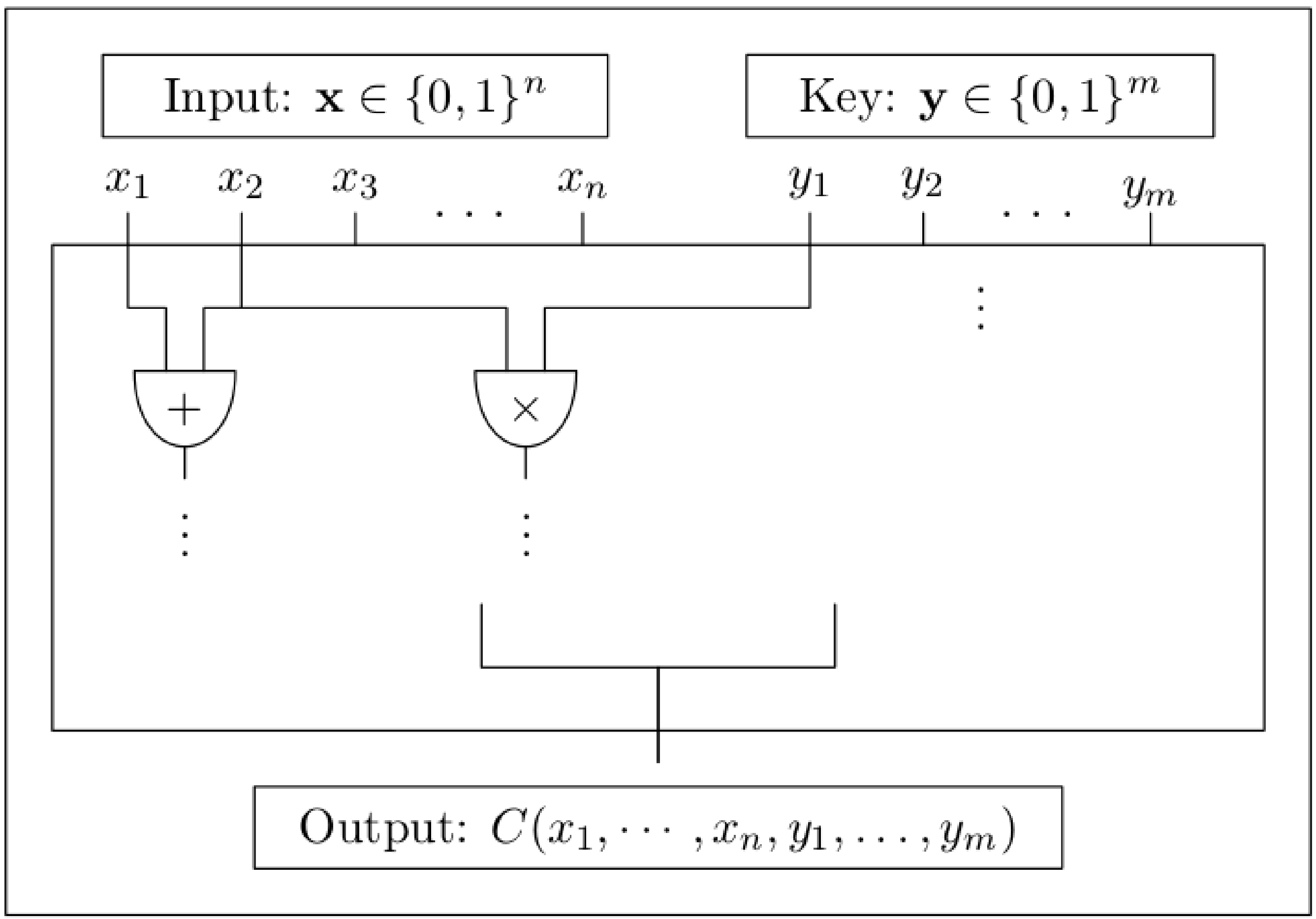
– Crucial property: adversary does not know $N_1, \ldots, N_k$,
cannot act independently on components

Our construction

– We obfuscate *keyed* arithmetic circuit families C(x, y)

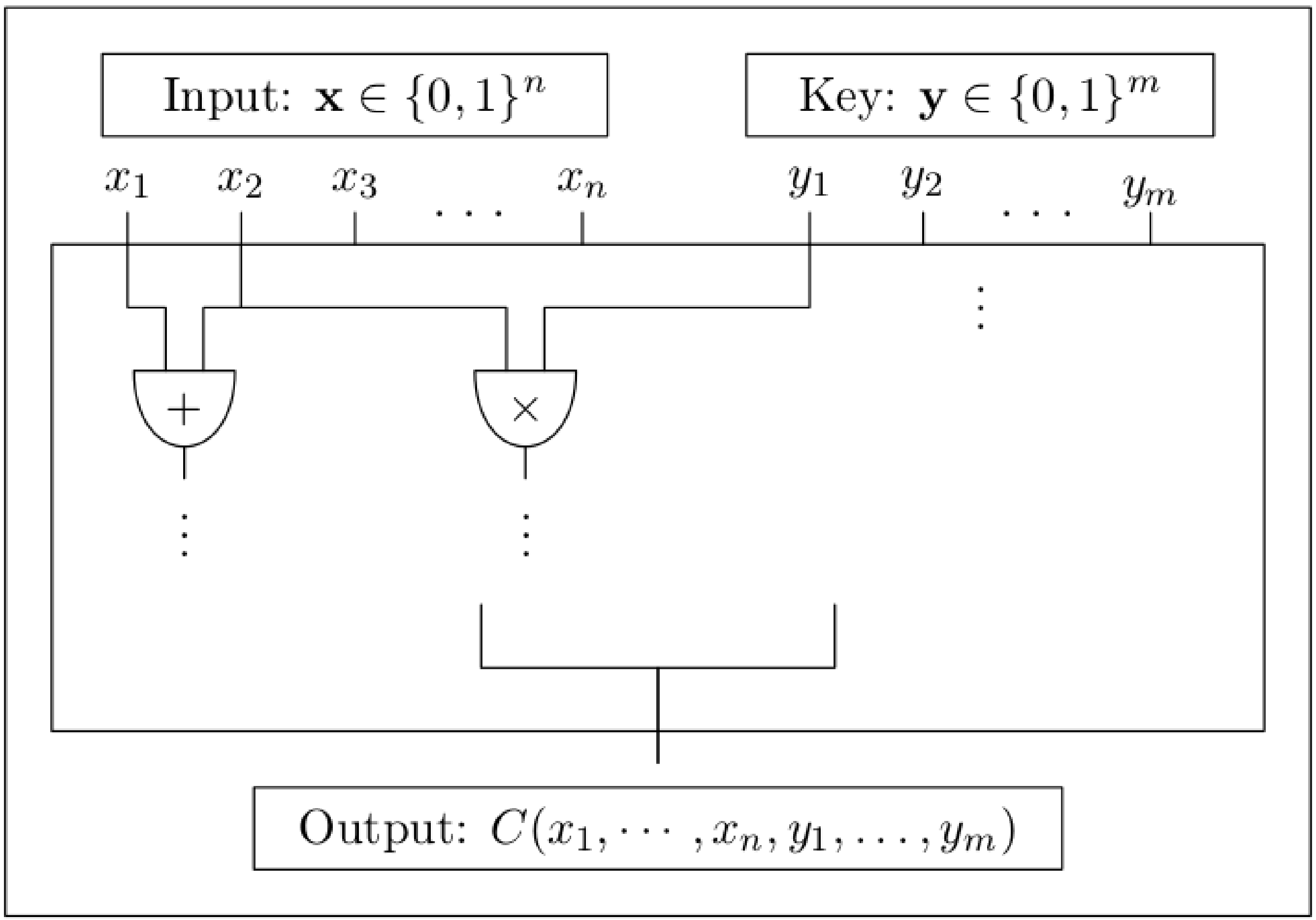
Our construction

– We obfuscate *keyed* arithmetic circuit families C(x, y)



Our construction

– We obfuscate *keyed* arithmetic circuit families C(x, y)
  – e.g., AES



Our construction

– We obfuscate *keyed* arithmetic circuit families C(x, y)

Our construction

– We obfuscate *keyed* arithmetic circuit families $C(x, y)$

– Boolean output defined by whether $C(x, y) = 0$

Our construction

– We obfuscate *keyed* arithmetic circuit families C(x, y)

– Boolean output defined by whether C(x, y) = 0

– Obfuscation need only hide secret key y

Our construction

- We obfuscate *keyed* arithmetic circuit families C(x, y)
- Boolean output defined by whether C(x, y) = 0
- Obfuscation need only hide secret key y
  - Must be able to evaluate C(., y) given $Obf_C(y)$

Our construction

- We obfuscate *keyed* arithmetic circuit families C(x, y)
- Boolean output defined by whether C(x, y) = 0
- Obfuscation need only hide secret key y
    - Must be able to evaluate C(., y) given $Obf_C(y)$
    - <u>VBB security</u>:

        "$Obf_C(y)$ no better than oracle access to C(., y)"

Our construction

– Keyed obfuscation C(., y) still general-purpose

(C can be universal circuit)

Our construction

– Keyed obfuscation C(., y) still general-purpose
  (C can be universal circuit)

– Avoid universal circuit when possible
  – Obfuscate directly

Our construction

– Keyed obfuscation C(., y) still general-purpose
    (C can be universal circuit)

– Avoid universal circuit when possible
    – Obfuscate directly

– Note: often only care about hiding key

Our construction

- Keyed obfuscation C(., y) still general-purpose
  (C can be universal circuit)

- Avoid universal circuit when possible
  - Obfuscate directly

- Note: often only care about hiding key
  - e.g., punctured PRFs [SW14]

Our construction

– Keyed obfuscation C(., y) still general-purpose
  (C can be universal circuit)

– Avoid universal circuit when possible
  – Obfuscate directly

– Note: often only care about hiding key
  – e.g., punctured PRFs [SW14]
  – Rich design space of data-oblivious algorithms

Our construction – first attempt

- First idea (not secure):
  - A keyed obfuscation of C(., y) contains:

Our construction – first attempt

- First idea (not secure):
  - A keyed obfuscation of C(., y) contains:
    - The circuit C

Our construction – first attempt

- First idea (not secure):

    - A keyed obfuscation of C(., y) contains:
        - The circuit C
        - 2n encodings, of the choices {0, 1} for each of the n bits of the input x

Our construction – first attempt

– First idea (not secure):

  – A keyed obfuscation of C(., y) contains:

    – The circuit C
    – 2n encodings, of the choices {0, 1} for each of the n bits of the input x
    – m encodings, one of each bit of the secret y

Our construction – first attempt

– First idea (not secure):

  – A keyed obfuscation of C(., y) contains:
    – The circuit C
    – 2n encodings, of the choices {0, 1} for each of the n bits of the input x
    – m encodings, one of each bit of the secret y

  – To evaluate C(x, y):

Our construction – first attempt

– First idea (not secure):

- A keyed obfuscation of C(., y) contains:
  - The circuit C
  - 2n encodings, of the choices {0, 1} for each of the n bits of the input x
  - m encodings, one of each bit of the secret y
- To evaluate C(x, y):
  - Select n of the 2n encodings for the bits of x

Our construction – first attempt

- <u>First idea (not secure):</u>
  - A keyed obfuscation of C(., y) contains:
    - The circuit C
    - 2n encodings, of the choices {0, 1} for each of the n bits of the input x
    - m encodings, one of each bit of the secret y
  - To evaluate C(x, y):
    - Select n of the 2n encodings for the bits of x
    - Produce an encoding of C(x, y) using the map's Add, Mult

Our construction – first attempt

- First idea (not secure):

  - A keyed obfuscation of C(., y) contains:
    - The circuit C
    - 2n encodings, of the choices {0, 1} for each of the n bits of the input x
    - m encodings, one of each bit of the secret y

  - To evaluate C(x, y):
    - Select n of the 2n encodings for the bits of x
    - Produce an encoding of C(x, y) using the map's Add, Mult
    - Test whether C(x, y) = 0 using the map's ZeroTest

Our construction

– Problem 1:

Our construction

- Problem 1:
  - Adversary can evaluate any computation, not just C!

Our construction

- Problem 1:
  - Adversary can evaluate any computation, not just C!
  - Standard approach [GGH+13b]:
    - Convert to branching program [Bar86]
    - “Garble” using Kilian's protocol [Kil88]

Our construction

– <u>Our solution (overview):</u>

Our construction

- Our solution (overview):
  - Composite modulus $N = N_{ev} N_{chk}$

Our construction

– <u>Our solution (overview):</u>

  – Composite modulus $N = N_{ev} N_{chk}$

  – Instead of encodings [0], [1] for each bit,
    give out encodings $[0, \alpha]$, $[1, \alpha]$ (CRT notation)

Our construction

- Our solution (overview):
  - Composite modulus $N = N_{ev} N_{chk}$
  - Instead of encodings [0], [1] for each bit,
    give out encodings $[0, \alpha]$, $[1, \alpha]$ (CRT notation)
  - Evaluation produces $[C(x, y), C(\alpha, \beta)]$ for random
    "check" vectors α, β modulo N_{chk}

Our construction

- <u>Our solution (overview):</u>
  - Composite modulus $N = N_{ev} N_{chk}$
  - Instead of encodings [0], [1] for each bit, give out encodings $[0, \alpha]$, $[1, \alpha]$ (CRT notation)
  - Evaluation produces $[C(x, y), C(\alpha, \beta)]$ for random "check" vectors α, β modulo N_{chk}
  - Subtract off *precomputed* encoding $[0, C(\alpha, \beta)]$

Our construction

– Our solution (overview):

  – Composite modulus $N = N_{ev} N_{chk}$

  – Instead of encodings [0], [1] for each bit, give out encodings $[0, \alpha]$, $[1, \alpha]$ (CRT notation)

  – Evaluation produces $[C(x, y), C(\alpha, \beta)]$ for random "check" vectors α, β modulo N_{chk}

  – Subtract off *precomputed* encoding $[0, C(\alpha, \beta)]$

Honest: $[C(x, y), C(\alpha, \beta)] - [0, C(\alpha, \beta)] = [C(x, y), 0]$

Our construction

- Our solution (overview):
  - Composite modulus $N = N_{ev} N_{chk}$
  - Instead of encodings [0], [1] for each bit, give out encodings [0, α], [1, α]   (CRT notation)
  - Evaluation produces [C(x, y), C(α, β)] for random "check" vectors α, β modulo N_{chk}
  - Subtract off *precomputed* encoding [0, C(α, β)]

Honest: [C(x, y), C(α, β)] – [0, C(α, β)] = [C(x, y), 0]

Malicious: [C'(x, y), C'(α, β)] – [0, C(α, β)]

Our construction

- <u>Our solution (overview):</u>
  - Composite modulus $N = N_{ev} N_{chk}$
  - Instead of encodings [0], [1] for each bit, give out encodings $[0, \alpha]$, $[1, \alpha]$ (CRT notation)
  - Evaluation produces $[C(x, y), C(\alpha, \beta)]$ for random "check" vectors α, β modulo N_{chk}
  - Subtract off *precomputed* encoding $[0, C(\alpha, \beta)]$

<u>Honest</u>: $[C(x, y), C(\alpha, \beta)] - [0, C(\alpha, \beta)] = [C(x, y), 0]$

<u>Malicious</u>: $[C'(x, y), C'(\alpha, \beta)] - [0, C(\alpha, \beta)]$

$= [C'(x, y), (C'-C)(\alpha, \beta)]$

Our construction

- <u>Our solution (overview):</u>
  - Composite modulus $N = N_{ev} N_{chk}$
  - Instead of encodings [0], [1] for each bit, give out encodings [0, α], [1, α] (CRT notation)
  - Evaluation produces [C(x, y), C(α, β)] for random "check" vectors α, β modulo N_{chk}
  - Subtract off *precomputed* encoding [0, C(α, β)]

<u>Honest</u>: [C(x, y), C(α, β)] – [0, C(α, β)] = [C(x, y), 0]

<u>Malicious</u>: [C'(x, y), C'(α, β)] – [0, C(α, β)]

= [C'(x, y), (C'-C)(α, β)]

- Adversary's C' will not pass ZeroTest unless C' = C (as a polynomial)

Our construction

– Problem 2:

Our construction

– Problem 2:
    – Adversary's computation can be *inconsistent*

Our construction

– <u>Problem 2:</u>

  – Adversary's computation can be *inconsistent*
  – For an input bit x_i, can supply encoding [0, α] on first use in C and [1, α] on second use

Our construction

- <u>Problem 2:</u>
  - Adversary's computation can be *inconsistent*
  - For an input bit x_i, can supply encoding [0, α] on first use in C and [1, α] on second use
  - Cannot be simulated using oracle access to C(., y)

Our construction

– <u>Problem 2:</u>

  – Adversary's computation can be *inconsistent*
  – For an input bit x_i, can supply encoding [0, α] on first use in C and [1, α] on second use
  – Cannot be simulated using oracle access to C(., y)

– <u>Our solution (overview):</u>

Our construction

– <u>Problem 2:</u>

  – Adversary's computation can be *inconsistent*
  – For an input bit x_i, can supply encoding [0, α] on first use in C and [1, α] on second use
  – Cannot be simulated using oracle access to C(., y)

– <u>Our solution (overview):</u>

  – Encodings for choices (x_i = 0, x_i = 1) have different *index sets*

Our construction

– <u>Problem 2:</u>

  – Adversary's computation can be *inconsistent*
  – For an input bit x_i, can supply encoding [0, α] on first use in C and [1, α] on second use
  – Cannot be simulated using oracle access to C(., y)

– <u>Our solution (overview):</u>

  – Encodings for choices (x_i = 0, x_i = 1) have different *index sets*
  – Introduce auxiliary “interlocking” encodings

Our construction

– <u>Problem 2:</u>
  – Adversary's computation can be *inconsistent*
  – For an input bit x_i, can supply encoding [0, α] on first use in C and [1, α] on second use
  – Cannot be simulated using oracle access to C(., y)

– <u>Our solution (overview):</u>
  – Encodings for choices ($x_i = 0$, $x_i = 1$) have different *index sets*
  – Introduce auxiliary “interlocking” encodings
  – Allow completion to top level U *only* for consistent expressions

Our construction

– Resulting algorithm (simplified; iO only):

Our construction

– Resulting algorithm (simplified; iO only):

Using $\mathsf{CM.Encode}(\mathsf{CM.sp}, \cdot)$, for $i \in [n]$, $j \in [m]$, and $b \in \{0,1\}$, generate the following encoded ring elements:

$$\hat{x}_{i,b} = [b, \alpha_i]_{X_{i,b}} \qquad \hat{u}_{i,b} = [1,1]_{X_{i,b}} \qquad \hat{y}_j = [y_j, \beta_j]_Y \qquad \hat{v} = [1,1]_Y$$

$$\hat{z}_{i,b} = [\delta_{i,b}, \gamma_{i,b}]_{X_{i,1-b}^{\deg(x_i)} Z_i W_i} \qquad \hat{w}_{i,b} = [0, \gamma_{i,b}]_{W_i}$$

$$\hat{C}^* = [0,\ C(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m)]_{Y^{\deg(\mathbf{y})} \prod_{i\in[n]} (X_{i,0} X_{i,1})^{\deg(x_i)} Z_i}$$

Our construction

– <u>Resulting algorithm (simplified; iO only):</u>

Using $\mathsf{CM.Encode}(\mathsf{CM.sp}, \cdot)$, for $i \in [n]$, $j \in [m]$, and $b \in \{0,1\}$, generate the following encoded ring elements:

$$\hat{x}_{i,b} = [b, \alpha_i]_{X_{i,b}} \qquad \hat{u}_{i,b} = [1,1]_{\underline{X_{i,b}}} \qquad \hat{y}_j = [y_j, \beta_j]_Y \qquad \hat{v} = [1,1]_Y$$

$$\hat{z}_{i,b} = [\delta_{i,b}, \gamma_{i,b}]_{\underline{X_{i,1-b}^{\deg(x_i)}} Z_i W_i} \qquad \hat{w}_{i,b} = [0, \gamma_{i,b}]_{W_i}$$

$$\hat{C}^* = [0,\ C(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m)]_{Y^{\deg(\mathbf{y})} \prod_{i \in [n]} \underline{(X_{i,0} X_{i,1})^{\deg(x_i)}} Z_i}$$

Interlocking index sets

Our construction

– <u>Main theorems</u>:

Our construction

- Main theorems:
  - We prove VBB security in generic model

Our construction

- <u>Main theorems</u>:
  - We prove VBB security in generic model
    - Requires additional techniques (e.g., straddling set systems [BGK+13])

Our construction

- Main theorems:
  - We prove VBB security in generic model
    - Requires additional techniques (e.g., straddling set systems [BGK+13])
  - Simplified version for *iO*; better efficiency parameters

Our construction

- Main theorems:
  - We prove VBB security in generic model
    - Requires additional techniques (e.g., straddling set systems [BGK+13])
  - Simplified version for *iO*; better efficiency parameters
  - Also define *succinct* obfuscation (size overhead depends only on secret y)

Our construction

- Main theorems:
  - We prove VBB security in generic model
    - Requires additional techniques (e.g., straddling set systems [BGK+13])
  - Simplified version for *iO*; better efficiency parameters
  - Also define *succinct* obfuscation (size overhead depends only on secret y)
    - Theorem: “clean” multilinear maps imply succinct obfuscation (assuming hardness of factoring)

Conclusion

- New construction:
  - Obfuscate general circuits *directly*
  - No matrix branching programs

Conclusion

– New construction:

  – Obfuscate general circuits *directly*

  – No matrix branching programs

– For “noisy” maps [GGH13a, CLT13, GGH14, CLT15]:

  – Concrete efficiency improvements

  – Remains impractical due to noise blowup

Conclusion

– New construction:

  – Obfuscate general circuits *directly*

  – No matrix branching programs

– For “noisy” maps [GGH13a, CLT13, GGH14, CLT15]:

  – Concrete efficiency improvements

  – Remains impractical due to noise blowup

– For “clean” maps (open problem):

  – Obfuscation for P/poly would now be practical!

Conclusion

– New construction:
  – Obfuscate general circuits *directly*
  – No matrix branching programs

– For “noisy” maps [GGH13a, CLT13, GGH14, CLT15]:
  – Concrete efficiency improvements
  – Remains impractical due to noise blowup

– For “clean” maps (open problem):
  – Obfuscation for P/poly would now be practical!
    – e.g., for AES: 133K ring elements, 281K ring ops

Conclusion

- New construction:
  - Obfuscate general circuits *directly*
  - No matrix branching programs
- For “noisy” maps [GGH13a, CLT13, GGH14, CLT15]:
  - Concrete efficiency improvements
  - Remains impractical due to noise blowup
- For “clean” maps (open problem):
  - Obfuscation for P/poly would now be practical!
    - e.g., for AES: 133K ring elements, 281K ring ops
  - “Noise” not inherent – central open problem