IMPLEMENTING BP-OBFUSCATION USING GRAPH-INDUCED GRADED ENCODING

Shai Halevi
Tzipora Halevi
Victor Shoup
Noah Stephens-Davidowitz

https://eprint.iacr.org/2017/104

Supported by the Defense Advanced Research Projects Agency (DARPA) and Army Research Office (ARO) under Contract No. W911NF-15-C-0236.
PROGRAM OBFUSCATION

- Make program “unintelligible”
  - Hide inner workings, only I/O should be “visible”
- Enable hiding secrets in software
  - E.g. cryptographic key, or an algorithm
- We seek an obfuscating compiler:
  - Arbitrary program in, obfuscated program out
  - Without changing the functionality
  - At most polynomial slowdown
OBfuscation IS USEful

- Commercially available ad-hoc obfuscation

  - Heuristic, trying to make reverse-engineering harder
  - Can always be broken with “enough debugging”
  - Can we get “crypto-strength” obfuscation?
CRYPTOGRAPHIC OBFUSCATION

- 1st plausible construction in [GGHRSW'13]
  - Several others since then
- Constructions have a “core component” that obfuscates “somewhat simple” programs
  - E.g., “branching programs” (BPs)
- Then a transformation that extends it to general programs
  - Using other tools (e.g., FHE, NIZK, RE, etc.)
How to Obfuscate?

- Main tool is “graded encoding” [GGH’13]
  - Like homomorphic encryption, values can be hidden by “encoding”, but still manipulated
  - Main difference: can see if the encoded value is 0

- High-level idea: run program on encoded values, check at the end if the result is zero
  - Main problem: hiding whether or not any two intermediate values are the same
  - Use randomization techniques for that
CRYPTOGRAPHIC OBFUSCATION CHALLENGES

- Security is poorly understood
- Current-day graded encoding is very costly
  - Other components make “core obfuscator” more costly still
- Previous implementation attempts:
  - [AHKM’14]: 14-bit point function
  - [LMA+’16] (5Gen): 80+ bit point function
    - More accurately 20+ nibbles
  - Note: point functions can be obfuscated much faster using special-purpose constructions
Our Work

- Obfuscate “read once branching programs”
  - Aka nondeterministic finite automata (NFA)
- Can handle ~100 states & upto 80-bit inputs
  - More accurately, 20 nibbles
- Can obfuscate some non-trivial functions
  - E.g., Substring/superstring/fuzzy match
- Still not enough for the “somewhat simple functions” that we would like to handle
OUR WORK

- Using the “graph-induced” graded encodings scheme of Gentry et al. [GGH’15]
  - Previous implementations used the encoding scheme of Coron et al. [CLT’13]
  - GGH15 seems better for NFAs with many states

- For performance reasons, could not implement one of the steps in [GGH’15]
  - Namely, the “bundling factors”
    - Implementation is only safe when used to obfuscate read-once BPs, not arbitrary BPs
**Some Details**

don’t worry, only three slides
OBfuscating BPs/NFAs

- Graphs, represented by transition matrices
  - Need to “hide” matrices, but allow them to be multiplied and compared to zero
- Begin by randomizing these matrices
  - Mainly Kilian-style randomization:
    \[ M_1 \times M_2 \times M_3 \rightarrow (M_1 R_1) \times (R_1^{-1} M_2 R_2) \times (R_2^{-1} M_3) \]
- Apply graded encoding to randomized matrices
- Can multiply encoded matrices, check for zero
  - But cannot “see” the original matrices
“Graph-induced” Graded Encoding

- Parametrized by a chain of matrices $A_i$
  
  $$A_0 \xrightarrow{M_1} A_1 \xrightarrow{M_2} A_2 \xrightarrow{M_3} \ldots \xrightarrow{M_n} A_n$$

- We encode “plaintext matrices” wrt edges

- Encoding of $M_i$ wrt $A_{i-1} \rightarrow A_i$ is a low-norm matrix $C_i$ s.t.,
  
  $$A_{i-1}C_i = M_iA_i + \text{small-error}$$

- The “hard part” is finding such a low-norm $C_i$
**“Graph-induced” Graded Encoding**

- Parametrized by a chain of matrices $A_i$

  $A_0 \xrightarrow{M_1} A_1 \xrightarrow{M_2} A_2 \xrightarrow{M_3} \ldots \xrightarrow{M_n} A_n$

- We encode “plaintext matrices” wrt edges

- Encoding of $M_i$ wrt $A_{i-1} \rightarrow A_i$ is a low-norm matrix $C_i$ s.t., $A_{i-1}C_i = M_iA_i + \text{small-error}$
  - The “hard part” is finding such a low-norm $C_i$

- It follows that $A_0 \prod_i C_i = (\prod_i M_i)A_n + \text{small-error}$
  - At least when the $M_i$’s themselves are small

- To test if $\prod_i M_i = 0$, check the size of $A_0 \prod_i C_i$
Our Main Optimizations

- Finding a small solution $C$ for $AC = B$:
  - Variant of trapdoor-sampling from [MP’12]
  - A new high-dimensional Gaussian lattice sampling
  - Working with integers in CRT representation
- Optimizing multiplication of very large matrices
  - Each matrix takes more than 18Gb to write down
- Many lower-level optimizations
  - Stash to reduce the number of samples, multi-threading strategies, memory-saving methods, …
### Some Performance Numbers

<table>
<thead>
<tr>
<th>$L$</th>
<th>$m$</th>
<th>Initialization</th>
<th>Obfuscation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3352</td>
<td>66.61</td>
<td>249.80</td>
<td>5.81</td>
</tr>
<tr>
<td>6</td>
<td>3932</td>
<td>135.33</td>
<td>503.01</td>
<td>13.03</td>
</tr>
<tr>
<td>8</td>
<td>5621</td>
<td>603.06</td>
<td>1865.67</td>
<td>56.61</td>
</tr>
<tr>
<td>10</td>
<td>6730</td>
<td>1382.59</td>
<td>4084.14</td>
<td>125.39</td>
</tr>
<tr>
<td>12</td>
<td>8339</td>
<td>3207.72</td>
<td>8947.79</td>
<td>300.32</td>
</tr>
<tr>
<td>14</td>
<td>9923</td>
<td>7748.91</td>
<td>18469.30</td>
<td>621.48</td>
</tr>
<tr>
<td>16</td>
<td>10925</td>
<td>11475.60</td>
<td>38926.50</td>
<td>949.41</td>
</tr>
<tr>
<td>17</td>
<td>11928</td>
<td>16953.30</td>
<td>44027.80</td>
<td>1352.48</td>
</tr>
<tr>
<td>18</td>
<td>12403</td>
<td>20700.00</td>
<td>out-of-RAM</td>
<td></td>
</tr>
</tbody>
</table>

#### Intel Xeon CPU, E5-2698 v3:

#### 4 x 16-core Xeon CPUs:

<table>
<thead>
<tr>
<th>$L$</th>
<th>$m$</th>
<th>Initialization</th>
<th>Obfuscation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>11928</td>
<td>16523.7</td>
<td>84542.3</td>
<td>646.46</td>
</tr>
<tr>
<td>19</td>
<td>13564</td>
<td>36272.9</td>
<td>182001.4</td>
<td>1139.36</td>
</tr>
<tr>
<td>20</td>
<td>14145</td>
<td>46996.8</td>
<td>243525.6</td>
<td>1514.26</td>
</tr>
</tbody>
</table>

68 hours

100 states, security=80, binary alphabet. $L$=input length, $m$=dimension
SOME PERFORMANCE NUMBERS

Memory vs. BP length

dec=80, 32 threads, binary alphabet

memory (GB)

BP length

obfuscation
SOME PERFORMANCE NUMBERS

Hard drive vs. BP length

sec=80, 32 threads, binary alphabet

- initialization
- obfuscation
Some Performance Numbers

- When using “nibbles” rather than bits for input:
  - Obfuscation time, disk usage, 8x increase
  - Everything else remains the same

- To handle BP of length 20 with input nibbles:
  - Init: 13hrs, obfuscate: 23 days, Eval: 25mins
  - RAM: 400GB
  - Disk space: ~10TB
CONCLUSIONS

- Cryptographic “general-purpose obfuscation” is barely feasible
  - Can handle some non-trivial functions
  - With inputs up to 20 characters (=80 bits)

- A new generation of constructions is now emerging [Lin’16,…]
  - Security is somewhat better understood
  - Practical performance still unknown
    - Could be better than previous constructions, or worse
Questions?

Thank You!
REFERENCES

- [Lin’16] Indistinguishability obfuscation from constant-degree ideal graded encoding, Eurocrypt 2016