CryptoSolve: Towards a Tool for the Symbolic Analysis of Cryptographic Algorithms

Dalton Chichester [1] Wei Du [2] Raymond Kauffman [1] Hai Lin [3] Christopher Lynch [3] Andrew M. Marshall [1] Catherine A. Meadows [4] Paliath Narendran [2] Veena Ravishankar [1] Luis Rovira [1] Brandon Rozek [5]

[1]University of Mary Washington, Fredericksburg, VA, USA

[2]University at Albany–SUNY, Albany, NY, USA

[3]Clarkson University, Potsdam, NY, USA

[4]Naval Research Laboratory, Washington, DC, USA

[5]Rensselaer Polytechnic Institute, Troy, NY, USA

WRLA 2022

High Level Framework

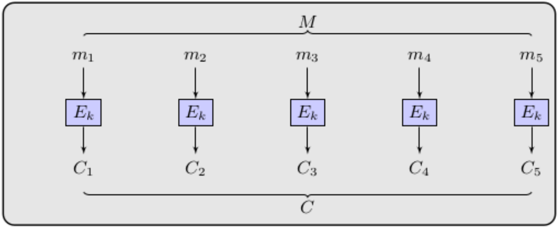
We are presenting a *preliminary* version of a tool that automatically synthesizes and verifies cryptographic algorithms.

Outline:

- Map the security property from the classical computational definition to a **symbolic security** equivalent. [Meadows, 2021]
- Apply **symbolic techniques** such as term rewriting and unification to verify cryptographic algorithms.
- Automatically **synthesize cryptosystems** that satisfy the security property.

Cryptographic Mode of Operation

- At this point, the tool supports the verification of symbolic security and invertibility of recursively defined *modes of operation* with an xor-operation and encryption function.
- A cryptographic mode of operation takes a message of arbitrary size and uses a block cipher to encrypt a fixed size parts of a message.



Core Security Question

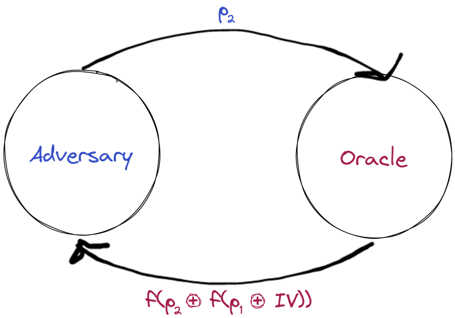
- We considered the computational security property *IND$-CPA*.
- That is, ciphertext indistinguishability from random under chosen plaintext attack.
- An *adversary* carefully selects plaintexts to send to an oracle in hopes of breaking symbolic security. An *oracle* returns the ciphertext according to a mode of operation.

Can an adversary force the cryptosystem to produce an equivalent sequence of ciphertexts modulo some equational theory? If so, we call the cryptosystem symbolically insecure.

Symbolic History

- Interactions between an adversary and an oracle in a cryptosystem can be modeled by a symbolic history.

Mode: Cipher Block Chaining



History: $[IV, p_1, f(p_1 \oplus IV), p_2, f(p_2 \oplus f(p_1 \oplus IV))]$

Symbolic Problem

- The adversary then takes the symbolic history, and tries to find a *computable substitution*[1] for their plaintexts to make some sequence of ciphertexts equivalent.

Symbolic History: $[IV, p_1, f(p_1 \oplus IV), p_2, f(p_2 \oplus f(p_1 \oplus IV))]$

Unification Problem: $f(p_1 \oplus IV) =_E ?\ f(p_2 \oplus f(p_1 \oplus IV))$

$$p_1 = ?? \quad p_2 = ??$$

[1] More on constraints of computable substitutions later. Example: adversary cannot compute f.

Related Work

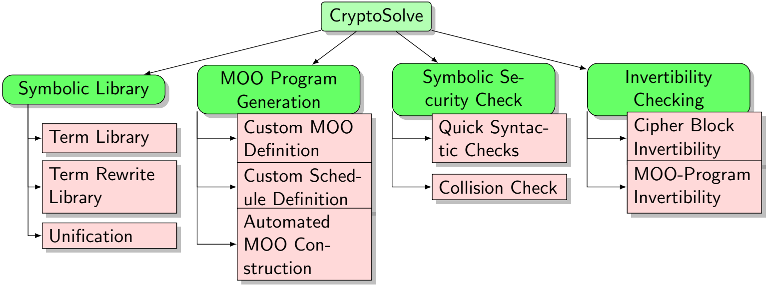
Tools closely related to ours include:

- ZooCrypt: Analyzes chosen plaintext and chosen cipher-text security public-key encryption schemes built from trapdoor permutations and hash functions.
- Linisynth: Generates and verifies multi-party computation schemes using free-xor compatible garbled circuits.

The goal of CryptoSolve, however, is to serve as a tool for designing and experimenting with multiple types of cryptosystems, security properties, and algorithms.

Tool Overview

Below is a categorized representation of the current capabilities of our tool.



Symbolic Library: Terms

```
f = Function("f", arity=1)
xor = Function("xor", arity=2)
IV = Constant("IV")
p1 = Variable("p1")

# Construct CBC term
c1 = f(xor(p1, IV))

# Helpful Methods
p1 in c1 # True
depth(c1) # 2
```

Substitutions

```
p2 = Variable("p2")
c2 = f(xor(c1, p2))

sigma1 = SubstituteTerm()
sigma1.add(p1, Constant("0"))

sigma2 = SubstituteTerm()
sigma2.add(p2, Constant("0"))

# Compose Substitions
sigma = sigma1 * sigma2

# Apply Substitution
c2 * sigma # f(xor(f(xor(0, IV)), 0))
```

Rewriting

```
x = Variable("x")
example_term = xor(xor(p1, p1), xor(p1, p1))

# Rule: xor(x, x) -> 0
xor_rule = RewriteRule(xor(x, x), Constant("0"))

# Application of a rule
xor_rule.apply(example_term)
# {'': 0, '1': xor(0, xor(p1, p1)), '2':
    xor(xor(p1, p1), 0)}
```

Algorithms for finding variants and performing narrowing are also included.

Unification

- In our library, unification returns a set of substitutions which each represent a most general unifier.
  - $\{\}$ means no unifiers were found.
  - $\{$`SubstituteTerm()`$\}$ is the identity unifier.

```
y = Variable("y")
a, b = Constant("a"), Constant("b")

# Syntactic Unification
unif({Equation(xor(x, y), xor(a, b))})
# {{ x -> a, y -> b }}
```

$MOO_{\oplus}$ Terms

We currently support analyzing modes of operations that are consisted of $MOO_{\oplus}$ terms.

- These terms are defined over the signature $\{\oplus/2, 0/0, f/1\}$ with the xor equational theory and f as a free function symbol.
- The xor equational theory can be represented as a combination of the Associative-Commutative (AC) equational theory and the rewrite system $\{x \oplus x \rightarrow 0, x \oplus 0 \rightarrow x\}$.

Computable Substitutions

Recall that the adversary wishes to find a *computable substitution* for their plaintexts to make some sequence of ciphertexts equivalent.

- A substitution σ is computable w.r.t a symbolic history if σ maps each variable to a term built up using the operators 0 and $\oplus$ on terms returned by the oracle earlier than x in P.

Note that the adversary cannot compute the f block cipher and must instead rely on ciphertexts received from the oracle.

Example

Consider the following symbolic history:
$[IV, p_1, f(p_1 \oplus IV), p_2, f(p_2 \oplus f(p_1 \oplus IV))]$

p_1 Computable Substitution Components: $0, IV$

p_2 Computable Substitution Components: $0, IV, p_1, f(p_1, \oplus IV)$

Unification and MOO Analysis

Currently we have two different unification algorithms for $MOO_{\oplus}$ terms which ensure the computable substitution constraint.

- f-rooted local unification
- $\oplus$-rooted local unification [Lin and Lynch, 2020]

Security Library: MOO Program

We currently support several well-known cryptosystems and allow for users to define their own.

```
@MOO.register("cipher_block_chaining")
def cipher_block_chaining(iteration, nonces, P, C):
  f = Function("f", 1)
  i = iteration - 1
  if i == 0:
    return f(
      xor(P[0], nonces[0])
    )
  return f(
    xor(P[i], C[i-1])
  )
```

MOO Schedules

So far we have assumed that the oracle immediately replies to the adversary. We support custom schedule types as well.

```
@MOO_Schedule.register("even")
def even_schedule(iteration: int) -> bool:
  return iteration % 2 == 0
```

MOO Security Checks

With the MOO Program and Schedule defined we can check for symbolic security.

```
moo_result = moo_check(
  moo_name = "cipher_block_chaining",
  schedule_name = "every",
  unif_algo = p_syntactic, # f-rooted local
  length_bound = 10
)
```

Interaction length bounds are included as this problem has been shown to be undecidable. [Lin et al., 2021]

MOO Invertibility Check

- It is not a given that any $MOO_{\oplus}$ Program (even secure ones) are *invertible*.
    - Invertible modes of operations would allow the original plaintext to be retrieved given the ciphertext and decryption function f^{-1}.

```
# CBC is invertible
print(moo_result.invert_result) # True
```

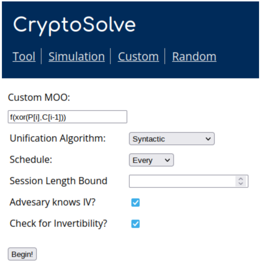
MOO Generator

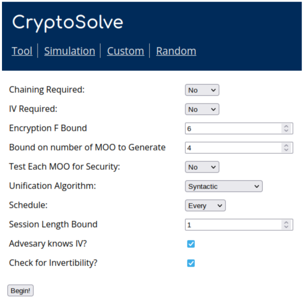
- Builds singly recursive definitions using the xor and f function, and recursive references to prior cipher blocks.
- Current Limitations:
  - A single nonce IV is used.
  - The base case is fixed to IV.
  - Only single recursion is used.
  - Signature is limited to $\Sigma = \{\oplus/2, 0/0, f/1\}$

```
from symcollab.moe import MOOGenerator
gen = MOOGenerator()
next(gen) # f(P[i])
```

User Interface

- We support testing symbolic security and invertibility for both custom modes of operation and procedurally generated modes of operation.





Experiments

- Using `MOOGenerator`, we ran and recorded the results of many automatically generated modes of operation.

Secure MOOs Found via Automatic Generation and Testing	
1	$C_0 = IV, C_i = f(f(f(P[i-1]) \oplus r) \oplus C[i-1])$
2	$C_0 = IV, C_i = f(f(f(P[i])) \oplus C[i-1] \oplus r)$
3	$C_0 = IV, C_i = f(f(P[i]) \oplus C[i-1]) \oplus C[i-1]$
4	$C_0 = IV, C_i = f(f(f(P[i]) \oplus r \oplus C[i-1]))$
5	$C_0 = IV, C_i = f(f(P[i]) \oplus C[i-1]) \oplus f(C[i-1])$

Table: Examples of secure MOOs found using the MOO generator

Conclusions

We presented a tool for the symbolic analysis of cryptographic algorithms. It supports:

- Checking symbolic security and invertibility.
- User-defined and automatic generation of modes of operation.
- Constraints on the generated modes of operation.
  - Requiring an initialization vector in the recursive definition.
  - Bounding the number of times f is applied.

Future Work

We plan to expand our tool beyond the current security properties by using our techniques to analyze:

- Additional Cryptosystems
- Symbolic Authenticity
- Multi-Party Computation (e.g Garbled Circuits)

We also plan to further improve the current work by:

- Improve `MOOGenerator` and Webpage.
- Expanding the signature to include hash functions.
- Improving the efficiency of security checking by discovering syntactic heuristics.

Lin, H. and Lynch, C. (2020).
Local xor unification: Definitions, algorithms and application to cryptography.
IACR Cryptol. ePrint Arch., 2020:929.

Lin, H., Lynch, C., Marshall, A. M., Meadows, C. A., Narendran, P., Ravishankar, V., and Rozek, B. (2021).
Algorithmic problems in the symbolic approach to the verification of automatically synthesized cryptosystems.
In Konev, B. and Reger, G., editors, *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings*, volume 12941 of *Lecture Notes in Computer Science*, pages 253–270. Springer.

Meadows, C. (2021).
Moving the bar on computationally sound exclusive-or.
In Bertino, E., Shulman, H., and Waidner, M., editors, *Computer Security – ESORICS 2021*, pages 275–295, Cham. Springer International Publishing.

Questions?

Thank you!

Check out our project's homepage to install and run our tool:
`https://symcollab.github.io/CryptoSolve/`