Kryptoanalyse (fast) ohne Mathematik

Moritz Schön

30.03.2024

Kryptographische Grundlagen

Challenge-Response

Alice (V)		**Bob**
$S_{\text{haredSecret}}$		$S_{\text{haredSecret}}$

. . . . Setup . . . .

$C_{\text{hallenge}} \leftarrow rand()$

$$\xrightarrow{C_{\text{hallenge}}}$$

$R_{\text{esponse}} \leftarrow$
$\text{MAC}(C_{\text{hallenge}}, S_{\text{haredSecret}})$

$$\xleftarrow{R_{\text{esponse}}}$$

$R_{\text{esponse}} =$
$\text{MAC}(C_{\text{hallenge}}, S_{\text{haredSecret}})$

Known-Plaintext Angriff

Alice (V)	**Eve**	**Bob**
$C_{\text{hallenge}} \leftarrow \mathit{rand}()$		
	$\xrightarrow{\quad C_{\text{hallenge}} \quad}$	
		$R_{\text{esponse}} \leftarrow \mathrm{MAC}(\dots)$
	$\xleftarrow{\quad R_{\text{esponse}} \quad}$	
$R_{\text{esponse}} = \mathrm{MAC}(\dots)$		

Known-Plaintext Angriff

$$R_{\text{esponse}} = \text{MAC}(C_{\text{hallenge}}, S_{\text{haredSecret}})$$

Known-Plaintext Angriff

$$R_{\text{esponse}}{}^{\checkmark} = \mathrm{MAC}(C_{\text{hallenge}}{}^{\checkmark}, S_{\text{haredSecret}}{}^{?})$$

Known-Plaintext Angriff

$$\exists S_{\mathsf{haredSecret}} \forall C_{\mathsf{hallenge}}, R_{\mathsf{esponse}} : R_{\mathsf{esponse}}{}^{\checkmark} = \mathrm{MAC}(C_{\mathsf{hallenge}}{}^{\checkmark}, S_{\mathsf{haredSecret}}{}^{?})$$

Known-Plaintext Angriff

$$\exists S_{\text{haredSecret}} \forall C_{\text{hallenge}}, R_{\text{esponse}} : R_{\text{esponse}}{}^{\checkmark} = \text{MAC}(C_{\text{hallenge}}{}^{\checkmark}, S_{\text{haredSecret}}{}^{?})$$

- Effizientes Finden des Shared Secrets notwendig für Angriff

Known-Plaintext Angriff

$$\exists S_{\text{haredSecret}} \forall C_{\text{hallenge}}, R_{\text{esponse}} : R_{\text{esponse}}{}^{\checkmark} = \mathrm{MAC}(C_{\text{hallenge}}{}^{\checkmark}, S_{\text{haredSecret}}{}^{?})$$

- Effizientes Finden des Shared Secrets notwendig für Angriff
- Wenn Angreifer*in Shared Secret berechnen kann, ist eine Authentifizierung möglich

Known-Plaintext Angriff

Alice (V)		**Eve**
$C_{\text{hallenge}} \leftarrow rand()$		
	$\xrightarrow{C_{\text{hallenge}}}$	
		$R_{\text{esponse}} \leftarrow \text{MAC}(\dots)$
	$\xleftarrow{R_{\text{esponse}}}$	
$R_{\text{esponse}} = \text{MAC}(\dots)$		

SA2 Kryptoalgorithmus

SA2

- Methode im Automobilsektor zur Authentifizierung gegenüber Steuergeräten zur Diagnose via UDS
  - leichtes Mitschneiden von CAN-Nachrichten
- Veraltet durch 32-Bit Challenges & Responses
- Nutzt unsicheres Primitiv von linear-feedback shift Register (LFSR)

SA2 Beispiel

$$S_{\text{haredSecret}} = 6805814A058704C11DB793C1387FA3494C$$

SA2 Beispiel

$$S_{\text{haredSecret}} = 6805814A058704C11DB793C1387FA3494C$$

$$\Downarrow$$

```
LOOP 0x5
        RSL
        BCC
                XOR 0x04C11DB7
        ADD 0xC1387FA3
        NEXT
END
```

SA2 Known-Plaintext Angriff

Demo

SMT-Solver

SMT-Solver

- automatisches und potenziell effizientes Verifizieren von Lösbarkeit
- liefert mögliche Belegung bei Lösbarkeit
- Erweiterung von SAT-Solving (Prädikatenlogik) um nicht nur boolsche Ausdrücke zu lösen sondern auch Arithmetik $\rightarrow$ höhere Abstraktionsebene
- NP-Vollständig
- aber gute Heuristiken für effizientes Lösen in vielen Fällen
- symbolische Ausführung

Intro zu SMT: `https://youtu.be/EacYNe7moSs`

SMT Beispiel

```python
import z3

solver = z3.Solver()
x = z3.BitVec('x', 32)

solver.add((0xAFA ^ (x * 2 + 0x69)) << 1 == 0x42)

if solver.check() == z3.sat:
    print(solver.model())
```

Angriff

Formalisieren

```python
import z3

MAX = z3.BitVecVal(0xFFFFFFFF, 33)

class SA2:
    def __init__(self, seed):
        self.register = z3.BitVecVal(seed, 33)
        self.carry = z3.BoolVal(False)
```

```python
@staticmethod
def xor(register, carry, operand):
    return register ^ operand, carry
```

Formalisieren

```python
@staticmethod
def add(register, _carry, operand):
    new_register = register + operand
    new_carry = z3.UGE(new_register, MAX)
    new_register = new_register & MAX
    return new_register, new_carry
```

Formalisieren

```python
@staticmethod
def rsl(register, _carry):
    new_carry = (register & 0x80000000) != 0
    new_register = (
        (register << 1) + z3.If(new_carry, z3.BitVecVal(1, 33),
        ↪  z3.BitVecVal(0, 33))
    ) & MAX
    return new_register, new_carry
```

Formalisieren

```
@staticmethod
def bcc(register, carry, operations):
    new_register = z3.If(
        carry, SA2._do_ops(register, carry, operations)[0],
        ↪  register
    )
    new_carry = z3.If(carry, SA2._do_ops(register, carry,
    ↪  operations)[1], carry)
    return new_register, new_carry
```

Formalisieren

```
@staticmethod
def loop(register, carry, iterations, operations):
    new_register = register
    new_carry = carry
    for _ in range(iterations):
        new_register, new_carry = SA2._do_ops(new_register,
        ↪  new_carry, operations)
    return new_register, new_carry

@staticmethod
def _do_ops(register, carry, operations):
    new_register = register
    new_carry = carry
    for op, *args in operations:
        new_register, new_carry = op(new_register, new_carry,
        ↪  *args)
    return new_register, new_carry
```

Darstellung als SMT Formel

```
LOOP 0x5
        RSL
        BCC
                XOR 0x04C11DB7
        ADD 0xC1387FA3
        NEXT
END
```

```
(SA2.loop, 5, [
    (SA2.rsl,),
    (SA2.bcc, [
        (SA2.xor, xorer)]),
    (SA2.add, adder)])
```

Darstellung als SMT Formel

```
solver = z3.Solver()

adder, xorer = z3.BitVecs("adder xorer", 33)
solver.add(z3.ULE(adder, MAX))
solver.add(z3.ULE(xorer, MAX))

for challenge, response in samples:
    vm = SA2(challenge)
    response = z3.BitVecVal(response, 33)

    vm.apply_op(
        SA2.loop, 5, [(SA2.rsl,), (SA2.bcc, [(SA2.xor, xorer)]),
        ↪  (SA2.add, adder)]
    )

    solver.add(vm.register == response)

if solver.check() == z3.sat:
    print(solver.model())
```

Darstellung als SMT Formel

$$\exists S_{\text{haredSecret}} \forall C_{\text{hallenge}}, R_{\text{esponse}} : R_{\text{esponse}}{}^{\checkmark} = \text{MAC}(C_{\text{hallenge}}{}^{\checkmark}, S_{\text{haredSecret}}{}^{(\checkmark)})$$

Angriff

Operationssequenz

Kerckhoffs'sches Prinzip

$$\mathrm{MAC}(\mathit{message}, S_{\mathrm{haredSecret}})$$

:= Öffentlich   := Geheim

Kerckhoffs'sches Prinzip

$$\mathrm{MAC}(\mathit{message}, S_{\text{haredSecret}})$$

$$\text{SA2: } \mathrm{MAC}(\mathit{message}, S_{\text{haredSecret}}) \mapsto S_{\text{haredSecret}}(\mathit{message})$$

$$\Rightarrow \mathrm{MAC}(\mathit{message}, S_{\text{haredSecret}})$$

$$\Rightarrow \text{Sicherheit nicht gewährleistbar}$$

:= Öffentlich := Geheim

Kerckhoffs'sches Prinzip

$$\mathrm{MAC}(message, S_{\text{haredSecret}})$$

$$\text{SA2: } \mathrm{MAC}(message, S_{\text{haredSecret}}) \mapsto S_{\text{haredSecret}}(message)$$

$$\Rightarrow \mathrm{MAC}(message, S_{\text{haredSecret}})$$

$\Rightarrow$ Sicherheit nicht gewährleistbar

Aber: Angriff schwieriger durchführbar wegen Obfuskation

:= Öffentlich := Geheim

Operationen

- `LOOP` mit `NEXT`
- `BRA`
- `BCC`
- `RSL` / `RSR`
- `ADD` / `SUB`
- `XOR`

Operationen

- `LOOP` mit `NEXT`
- `BRA`
- `BCC`
- `RSL` / `RSR`
- `ADD` / `SUB`
- `XOR`

Überführen in formalere Darstellung als Python-Objekte ✓
Bruteforcen der Operationssequenz mit Heuristiken

Operationssequenzgenerator

Herausforderungen

- Manche Operationen erfordern rekursiven Generator
- Schnelles Wachstum der Ausgangsmenge bei Support von mehr Patterns

Operationssequenzgenerator

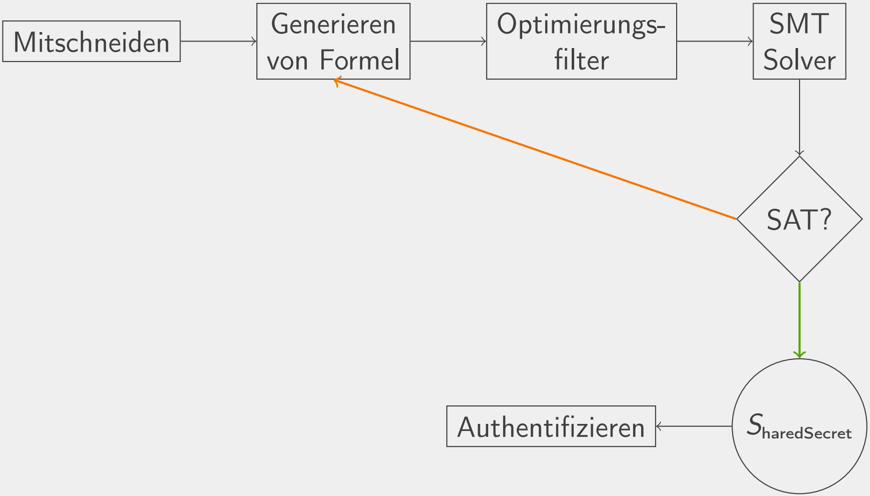
Lösungsansätze

- Maximale Rekursionstiefe
- Nutzen von Heuristiken bekannter Pattern
- Sortieren nach Lösungsgeschwindigkeit
- Deduplizieren äquivalenter Formeln

Optimierungen

```
def check_unnecessary_carry(operations):
    for start_index in range(len(operations) - 2):
        match operations[start_index:]:
            case [SA2Op.RSR, SA2Op.RSL, non_conditional, *other] \
               | [SA2Op.RSL, SA2Op.RSR, non_conditional, *other] \
                if non_conditional in NON_CONDITIONAL:
                if non_conditional == SA2Op.XOR:
                    for op in other:
                        if op == SA2Op.XOR:
                            continue
                        if op in NON_CONDITIONAL:
                            return False
                else:
                    return False
    return True
```

Ablauf



Mitschneiden
Generieren von Formel
Optimierungs-filter
SMT Solver
SAT?
$S_{\text{haredSecret}}$
Authentifizieren

Demo

Analyse

Vulnerabilität von SA2

- SMT-Solver erleichtert Suche von Konstanten im vgl. zu Bruteforce immens
- 64-bit Lösungsraum gut angreifbar
- 96-bit Lösungsraum erfordert höhere Motivation und praktikable Operationssequenz

. . . und alles ohne mathematisch strukturelle Analyse von LSFRs

Alternativen

- andere SMT-Solver
- SageMath/Mathematica/Matlab ...

Takeaways

- an obskuren Kryptoalgorithmen rumprobieren
- Don't roll your own Crypto(protocol)
- Kerckhoffs'sches Prinzip
- **Bonus:** CSPRNG verwenden

Bonus

- PRNG auf Basis von `XorShift128`
- interner State mit SMT rückrechenbar

https://youtu.be/-h_rj2-HP2E

- https://github.com/TheAlgorythm/kryptoanalyse-ohne-mathe
- https://zschoen.dev
- @TheAlgorythm@chaos.social
- hi@zschoen.dev

Danke an:

AVL