FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Ondřej Med

Vulnerabilities and security proofs of communication protocols used by malware

Department of Algebra

Supervisor of the bachelor thesis:	Adolf Středa, Mgr.
Study programme:	Mathematics for Information Technologies
Study branch:	Mathematics for Information Technologies

Prague 2023

First and foremost, I want to express my gratitude to Adolf Středa for his availability, feedback, and unwavering support not only throughout the process of writing this thesis. I would also like to extend my thanks to Bruno Blanchet to whom I am deeply grateful for his advice and ready assistance.

Title: Vulnerabilities and security proofs of communication protocols used by malware

Author: Ondřej Med

Department: Department of Algebra

Supervisor: Adolf Středa, Mgr., Department of Algebra

Abstract: Cryptographic games with their transitions are a useful tool for proving cryptographic properties of various security protocols. We have explored together the notions of negligible functions, cryptographic games and their transitions, computational, and perfect security. This served as our basis when analyzing malware protocols, we translated each into a game that tested the property we were trying to expose. Then, using transitions based on negligible functions, we simplified said games to reach desired results.

We have decided to employ Cryptoverif as our tool for implementing these games, it is designed to create sequences of games that lead to games exposing specified properties. We translated the games into series of primitives that form an interface of this tool. Using the theory described above we anchored individual transitions in mathematical arguments and documented the proof strategy Cryptoverif employs.

To illustrate the usage, we have selected a few communication protocols used by several malware families (Emotet, Mirai, Lockcrypt) and used the tooling to prove a few characteristic properties. While this has been challenging for a few cases (especially when the techniques were not entirely inside Cryptoverif's scope) we have managed to design our games accordingly to reach the desired results. We have demonstrated Cryptoverif capabilities by revealing vulnerability in Mirai's key generation, verifying correctness of Emotet's encryption and illustrating of improper usage of one-time pad encryption by ransomware.

Keywords: computational security cryptoverif cryptographic games

Contents

Preface

When it comes to encrypting data we have two usually contradictory aims that we follow - security and practicality. While perfect security maximizes security, it usually comes at the cost of practicality. Therefore, we usually relax our security requirements to a level that still provides practical security while providing enough flexibility to cover most use cases. One way of achieving this is to base our security assumptions on mathematical problems for which we have no knowledge of an efficient (read polynomial) solution.

Furthermore, we need to restrict our attacker, because any security fails against an adversary capable of brute-forcing the whole key space, therefore we impose limitations reflecting real-world computational capabilities. Thus, we will be most interested in efficient attackers, i.e. attackers working within polynomial time.

We will also explore one of the ways of making proofs of the security of cryptographic schemes and how can this be automated.

1. Introduction

There are two types of security: perfect/unconditional and computational/conditional. Perfect security relies on long keys that are never reused, making it impractical for most purposes. It is based on information theory and, while it has some applications, conditional/computational security is more commonly used.

Computational/conditional security uses keys that are long enough to resist brute-force attacks, but they do not need to be as long as plaintexts. the keys are usually of a fixed length, and observing or interacting with the scheme should not allow an adversary to efficiently reveal the keys.

1.1 Perfect security

Perfect security is based on the idea formulated in the following definition from Katz and Lindell [2007]:

Definition 1. *an encryption scheme over a message space M is perfectly secret if for every probability distribution over M, every message $m \in M$, and every ciphertext $c \in C$ for which $P[C = c] > 0$ holds:*

$$P[M = m|C = c] = P[M = m]$$

This means that plaintext is wholly determined by key in decryption, thus we need to have at least as many keys as we have possible plaintexts.

Lemma 1. *Let us have a perfectly-secret encryption scheme over a message space M, and let K be the key space. Then $|K| \geq |M|$.*

Proof. Assume that $|K| < |M|$. Let us consider ciphertext $c \in C$ and $M(c) = \{dec(c, k)|k \in K\}$ - set of all decryption of C. It follows that $|M(c)| \leq |K|$. Applying the assumption we arrive at $|M(c)| < |M|$, thus we can take $m \in M \setminus M(c)$.

Clearly, now:

$$P[X = m|C = c] = 0 \neq P[X = m]$$

, which is in contradiction with the definition of perfect security.

□

Another limitation that we mentioned is key reuse. To comply with the definition of perfect secrecy the combined number of possible messages can not exceed the number of possible keys. Therefore, it is possible to reuse the same key, but the key set must be larger than the set of all concatenations of the messages.

1.2 Complexity and Big O notation

To compare functions and their asymptotic behavior, we use Big O notation, which is defined in the following manner:

Definition 2. *We say a function f is of class g if there is a positive number M such that for values x sufficiently large inequality $f(x) < Mg(x)$ holds:*

$$\exists x_0 \in \mathbb{R} : \exists M \in \mathbb{R}^+ \forall x > x_0 : f(x) \leq Mg(x)$$

This gives us a more formal way to classify the asymptotic speed of growth of functions, where g provides an upper bound. We can also order these classes by subset relation, i.e. $O(x) \subset O(x^2)$ because x^2 is strictly larger than x for numbers larger than 1. However, $x^2 \in O(x^2)$ and certainly outgrows every polynomial of form Mx for values larger than M. Using similar arguments we can prove that $O(x^n) \subset O(x^{n+1}) \subset O(k^x) \subset O(x!)$ where $k, n \in \mathbb{N}, k > 1$.

There are two definitions we will be using closely connected to this notation:

Definition 3. *Function $f(x)$ is superpolynomial if there is no $k \in \mathbb{N}$ such that $f(x) \in O(x^k)$.*

Intuitively, a superpolynomial function grows asymptotically faster than any polynomial.

Definition 4. *Function $f(x)$ is negligible if for every $k \in \mathbb{N}$ holds $f(x) \in O(x^{-k})$.*

Intuitively, a negligible function grows asymptotically slower than any fractional polynomial. Later on, we introduce a different but equivalent definition of negligibility. We will call adversaries working in polynomial time *efficient adversaries.*

1.3 Computational security

In our chosen approach we introduce a scaling parameter $n \in \mathbb{N}$ for a cryptographic scheme that is determined for each session. This usually represents the length of a key for a given scheme. Scaling n allows us to meet challenges presented by faster adversaries while preserving operability for honest parties. an increase in n should be much more impactful for potential attacks than for an honest party.

We consider the computational time of an adversary and its chance of success as functions of n. It is also common to define times of actions of an honest party such as encryption and decryption using a key as functions of n.

This will allow us to respond to more computationally powerful polynomial attacks while preserving the operability of the scheme for honest parties. We view computational time and the chance of success of an adversary as functions of n and rather than demanding perfect security, we introduce two relaxations:

1. *An adversary may have a negligible chance to succeed.*
2. *We consider only adversaries running in polynomial time: $A(n) \in O(n^k)$.*

We need both of these relaxations. If we only use a finite set of keys, randomly choosing from this set would provide a success probability of $1/|K|$. While this probability is small it is still non-zero, so we need to allow for a small chance of success. Similarly, if an adversary can attempt all possible values of K (brute-force attack), we must restrict the adversary's computational power.

Suppose that $|K| \in O(n^k)$, the computational time of an honest party is $p(n)$, then a simple brute-force attack runs in a time $A(n) \in O(p(n)n^k)$. So unless $p(n)$ is superpolynomial, then brute-force attack succeeds in polynomial time. Therefore the key set must be superpolynomial.

We will use definitions of asymptotic security and of negligible function with a proposition on its properties from Katz and Lindell [2007]:

Definition 5. *a scheme is conditionally secure if every probabilistic polynomial time adversary succeeds in breaking the scheme with only negligible probability.*

Katz and Lindell [2007] uses a different but equivalent definition of negligibility.

Definition 6. *a function f is negligible if for every polynomial $p(n)$ there exists an N such that for all integers $n > N$ it holds that $f(n) < 1/p(n)$.*

$$f \textit{ negligible} \iff \forall p(n) \exists N \forall n > N : f(n) < 1/p(n) \tag{1.1}$$

Lemma 2 (Equivalence of definitions of negligibility)**.** *For function $f(n)$ these conditions are equivalent:*

1. $\forall p(n) \in \mathbb{R}[x], \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N} : (n > n_0 \implies f(n) < \frac{1}{p(n)})$
2. $\forall k \in \mathbb{N} : (f(n) \in O(x^{-k}))$

Proof. Firstly, we will prove (1) $\implies$ (2):

Suppose there is $k \in \mathbb{N} : f(n) \notin O(x^{-k})$. That implies:

$$\forall n_0 \exists n > n_0 : f(n) > x^{-k}$$

Now let us apply the first definition on polynomial x^k, we obtain:

$$\exists m_0 : \forall m > m_0 : f(n) < \frac{1}{x^k} = x^{-k}$$

Which is a contradiction and $f(n) \in O(x^{-k})$ for all $k \in \mathbb{N}$.

(2) $\implies$ (1): We will prove this once again by reaching a contradiction, so assume that:

$$\forall k \exists M \exists n_0 \forall n > n_0 : f(n) < Mx^{-k}$$
$$\exists p \in \mathbb{R}[x] \forall n_0 \exists n > n_0 : \frac{1}{p(n)} \leq f(n)$$

However, if there is one value for n_0 for which the statement holds, then it holds for all larger values since n_0 is just a lower bound on n. Combining the two statements we obtain:

$$\frac{1}{p(n)} < \frac{M}{n^k}$$
$$M \cdot p(n) > n^k$$

Let's take n_0 larger than 2 a define $k = \deg p(n) + M + \sum a_i$ where $p(n) = \sum a_i x^i$, then by simple observation we can conclude that we have reached a contradiction.

□

Lemma 3. *Let f and g be negligible functions.*

1. *the function $h(n) = f(n) + g(n)$ is negligible.*
2. *For any positive polynomial p, the function $h(n) = p(n)f(n)$ is negligible.*

Proof. Consider polynomial $q(n)$, using definition 6 for $2q$ we obtain:

$$\exists N \forall n > N : f(n), g(n) < \frac{1}{2q(n)}$$

which trivially yields the desired result:

$$\exists N \forall n > N : f(n) + g(n) < \frac{1}{p(n)}$$

Similarly, we will apply definition on polynomial $q(n)p(n)$:

$$\exists N \forall n > N : f(n) < \frac{1}{q(n)p(n)}$$
$$\exists N \forall n > N : f(n)p(n) < \frac{p(n)}{q(n)p(n)} = \frac{1}{q(n)}$$

□

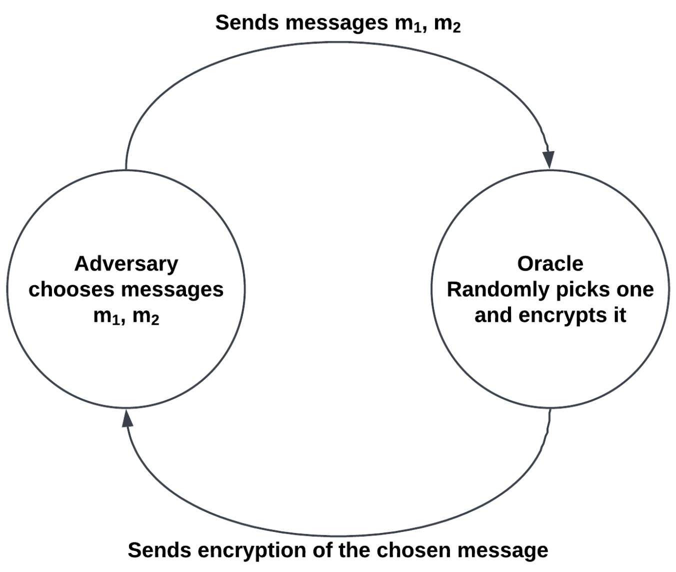
The second result implies that an adversary repeating an attack with a negligible probability of success polynomially-many times still has only a negligible chance. This means that we may ignore attacks that are simple repetitions of the same attack.

1.4 Cryptographic games

Cryptographic games are built on top of protocols and may involve honest parties, oracles, and adversaries. an adversary usually represents a potential attacker that can read public messages, and intercept or resend them. an oracle is usually a passive party other players in the scheme can freely interact with. It may cipher messages, generates keys, etc and its running time is considered immediate. Single usage of an oracle shouldn't be generally considered a limiting factor, we may consider the runtime of a call to an oracle to be $O(1)$.

The games are used to explore properties of protocols, i.e. introducing an adversary into an encryption and authentication scheme (thus creating a game from a scheme) can be used to test whether the scheme is secure.

A simple example is the left-right game, which involves only one adversary and an oracle. the adversary generates two messages and lets the oracle randomly choose one of them to encrypt and send back to the adversary. the adversary then guesses which of the two initial messages the resulting encrypted message corresponds to.

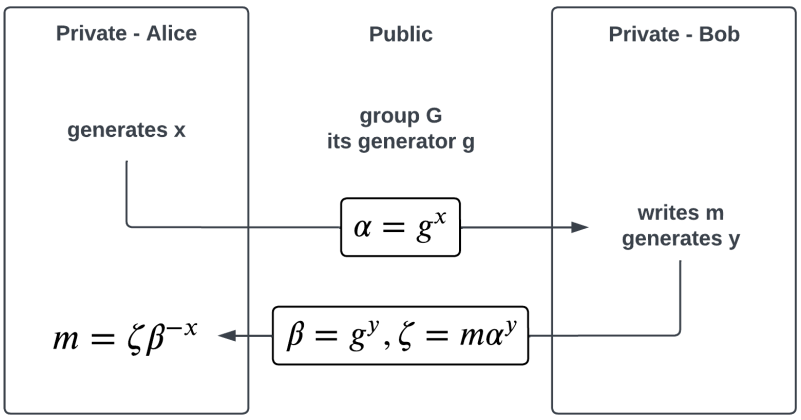

Sends messages m_1, m_2
Adversary chooses messages m_1, m_2
Oracle Randomly picks one and encrypts it
Sends encryption of the chosen message

If an encryption scheme is secure against the left-right distinguishing game with an adversary choosing what messages to play the game with, we refer to this property as IND-CPA, which stands for indistinguishable under chosen plaintext attack.

1.5 ElGamal left-right game

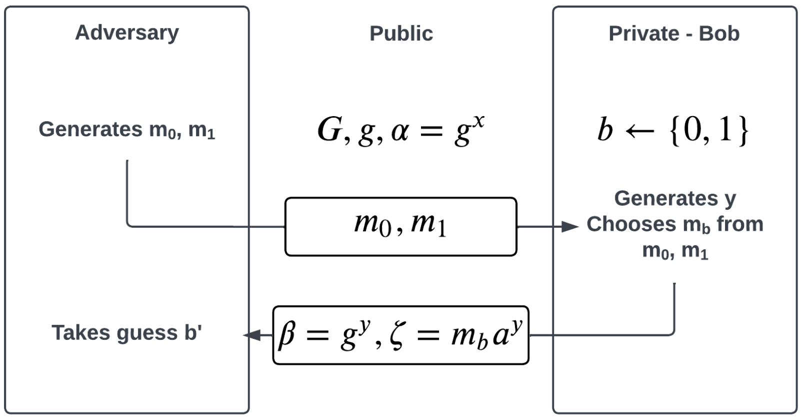
We will use the well-known ElGamal encryption protocol to illustrate what cryptographic games are. Let us first walk through the protocol:

1. the first (receiving) party generates a private key x from known group $\mathbb{Z}_q$ and computes $\alpha = g^x$ where g is publicly known group generator. It then publishes α.
2. the second (sending) party generates a private key y from $\mathbb{Z}_q$, computes the shared secret $\delta = \alpha^y$, encrypted message $\zeta = m \cdot \delta$ and its own public key $\beta = g^y$, where m is a message for first party. It publishes ζ, β.
3. the first party decrypts the encrypted message using ζ, β as $m = \zeta / \beta^x$. It only needs to calculate the inverse, which can be easily done if the order of the group is known.



A left-right distinguishing game applied to this protocol proceeds in the following way (it is separated into points so that it corresponds to the game representation later):

1. the first party generates a private key x from a known group $\mathbb{Z}_q$ and computes $\alpha = g^x$ where $g \in G$ is a generator.
2. an adversary generates random coins r that serve as its decision mechanism and two messages m_0, m_1.
3. the second party generates a bit b that determines what messages from m_0, m_1 will be encrypted and another key y. Then it computes the shared secret δ and finally calculates the encrypted message ζ.
4. the adversary creates an estimate of b using public information α, β, ζ denoted b' .



Pi-calculus is another method for representing protocols in cryptography, it is particularly useful for its well-definable syntax that can be easily parsed by a program. Further on, we will encounter a different variation of this syntax when we get to implementing protocols. This is a representation of the ElGamal protocol from Shoup [2004]:

ElGamal protocol in Pi-calculus

```
$x \hookleftarrow \mathbb{Z}_q, \alpha \leftarrow \gamma^x$
$r \hookleftarrow R, (m_0, m_1) \leftarrow A(r, \alpha)$
$b \hookleftarrow \{0, 1\}, y \hookleftarrow \mathbb{Z}_q, \beta \leftarrow \gamma^y, \delta \leftarrow \alpha^y, \zeta \leftarrow \delta \cdot m_b$
$b' \leftarrow A(r, \alpha, \beta, \zeta)$
```

Symbols have the following meaning:

1. $\leftarrow$ assigns the result of the calculation that immediately follows the arrow.

2. $\hookleftarrow$ uniformly draws an element from the set that immediately follows the arrow.

3. $A(args)$ represents the decision of an adversary based on the information provided in its arguments.

We also define events when we want to talk about certain states that may or may not take place in a game. In our case, an event we might want to observe is that an adversary has made a correct guess $b = b'$.

Our goal is to prove that an adversary has at most negligibly higher chance of correctly guessing b than 1/2. We estimate the probability of an event $P[S] = P[b = b']$. If the given probability can't be easily obtained from the original game, we may transform the game into different where we observe a corresponding event that is linked to the original up to a negligible probability.

1.6 Transitions of games

Transitions (or transformations, we used these term interchangeably) between games may be based on these arguments:

1. Failure events

2. Indistinguishability

3. Semantic changes

Let us consider a set of cryptographic games and define a relation between the two games to exist if and only if an adversary has at most a negligible chance of differentiating between them. Thanks to the properties of negligible functions, as described in lemma 3, this relation is transitive and from the definition follows that it is also reflexive and symmetric. Therefore, we have an equivalence relation on the set of all cryptographic games.

Within the resulting classes of equivalence of this set, we are transitioning from one game to another in pursuit of a simpler expression of the same game that will allow us to derive sought-after qualities. Specifically, we aim to reach games that contain known mathematical problems which we may take advantage of.

1.6.1 Transitions based on failure events

To transition between two games, we may define an event that distinguishes the two games from each other. For example, such an event could be a collision between hash values. Elimination of this event would allow us to transition to a game that assumes that the values are different.

Generally, in order to make such transition, we need to prove that, unless a failure event takes place, the games are identical and just as importantly that the probability of the failure event is negligible.

This allows us to discard cases where both parties generate the same private key and other improbable scheme-breaking events. Thanks to these transitions we may simplify the game at the cost of permitting a negligible chance of the games not being equal.

More formally, we base our transitions on the following lemma from Shoup [2004]:

Lemma 4. *Let S_{old}, S_{new}, F be events defined in some probability distribution, and suppose that $S_{old} \wedge \neg F \iff S_{new} \wedge \neg F$. Then $|P[S_{old}] - P[S_{new}]| \leq P[F]$.*

Proof.

$$\begin{aligned} |P[S_{old}] - P[S_{new}]| &= |P[S_{old}, F] + P[S_{old}, \neg F] - P[S_{new}, F] - P[S_{new}, \neg F]| \\ &= |P[S_{old}, F] + P[S_{old}, \neg F] - P[S_{new}, F] - P[S_{old}, \neg F]| \\ &= |P[S_{old}, F] - P[S_{new}, F]| \\ &\leq P[F] \end{aligned}$$

In the first two steps, we split probabilities of S_{old}, S_{new} according to F, and then apply the presume equality. Finally, what remains is a difference of two probabilities, which is bound in F.

□

If we want to prove that $|P[S_{old}] - \mathit{const}|$ is negligible using a failure event, we are in fact using the following upper bound:

$$\begin{aligned} |P[S_{old}] - \mathit{const}| &= |P[S_{old}] - P[S_{new}] + P[S_{new}] - \mathit{const}| \\ &\leq |P[S_{old}] - P[S_{new}]| + |P[S_{new}] - \mathit{const}| \\ &\leq P[F] + |P[S_{new}] - \mathit{const}| \end{aligned}$$

Thanks to lemma 6 we can discard any negligible summands. Therefore, if the probability of the failure event is negligible, we can transition from one game to another.

1.6.2 Transitions based on indistinguishability

Transitions based on failure events allow us to discard unfavorable instances of the game as long as there are negligibly few of them. On the other hand, transitions based on indistinguishability allow us to make minor changes to all instances of the game as long as an adversary has at most a negligible chance of differentiating between the game modified and unmodified game.

In transitions based on indistinguishability, we may model the noticeability of the changes with events S_{old}, S_{new}. The success chance of an adversary is tied to the difference of probabilities of S_{old}, S_{new}, therefore, we need to prove that the difference is negligible. Suppose events are defined as an adversary taking correct guess in a corresponding left-right distinguishing game, and that the original aim was to prove that $|P[S_{old}] - 1/2]|$ is negligible, the transition can be mathematically expressed as:

$$\begin{aligned} |P[S_{old}] - 1/2]| &= |P[S_{old}] - P[S_{new}] + P[S_{new}] - 1/2]| \\ &\leq |P[S_{old}] - P[S_{new}]| + |P[S_{new}] - 1/2]| \end{aligned}$$

The point of the transition is that estimating or bounding of $|P[S_{new} - 1/2]|$ is possibly easier than $|P[S_{old}] - 1/2|$. However, we still need to prove that $|P[S_{old}] - P[S_{new}]|$ is negligible. To prove this, we define an algorithm D that combines both distinguishing games and chooses between them based at random (more formally this can be defined with distribution of the input).

An adversary in this combination of the two games has to guess what game was played and we have to prove that his advantage is negligible. An advantage refers to the difference between the success chance of an adversary and the chance of uniformly distributed guesses.

1.6.3 Semantic transitions

These transitions include substitution and other semantically equivalent adjustments, as they are a useful tool for automated transitions. They may be mathematically unimportant but are essential for pattern matching used by some automated provers.

2. Cryptoverif and Mirai example

2.1 Cryptoverif and implementation of cryptographic games

Cryptoverif (as described in an article on the underlying algorithm Blanchet [2008], formal description of the language Blanchet [2017], and manual Blanchet and D. [2022]) is a prover that allows the implementation of custom protocols and provides access to a library of predefined cryptographic primitives. Cryptoverif takes advantage of custom types to distinguish different between variables, types can have different properties but can also serve to distinguish messages as described a little later.

Cryptoverif uses notation of the different interfaces. One interface models protocols as communication over several channels, where messages are then relayed by specifying where individual messages are published and from what channels are messages read by individual parties.

We will be using the other interface that models individual interactions as oracles. These oracles read and publish messages, the format of a message determines which other oracles can receive it. Messages are tuples of variables, terms, and constants, and the tuple of types of individual elements of a message is referred to as a format.

2.2 Mirai family with an example of game transformation

One of the protocols we studied is from the Mirai malware family[1], which we chose for its simplicity and a glaring flaw. As described in Středa and Neduchal [2018], the malware generates a 4-byte long key and then computes xor of all 4 bytes of the key, this is then used for message encryption.

Game 0

```
$(k1, k2, k3, k4) < -\mathbb{Z}_{2^{32}}, k \leftarrow k_1 \oplus k_2 \oplus k_3 \oplus k_4$
$p \leftarrow P, m \leftarrow enc(p, k)$
```

Game 1

```
$k \hookleftarrow \mathbb{Z}_{2^8}$
$p \leftarrow P, m \leftarrow enc(p, k)$
```

This is a transition based on indistinguishability. To prove that it is valid we will design a distinguishing game and prove that the adversary's advantage is negligible. In this case, it may be a bit of a mosquito-killing cannon approach. We will do it nonetheless to demonstrate this proof technique.

Claim 5. *Game 0 and Game 1 are indistinguishable.*

[1]A malware family refers to a group of malware instances that have a common codebase.

Proof. We will design a distinguishing game around the following algorithm that takes two keys and a bit as an input, all drawn uniformly from respective sets. The bit allows us to switch between games, if it is zero then the game 0 takes place as $\oplus bk_2$ becomes ineffectual.

Distinguishing algorithm

```
$D(k_1, k_2, b):$
$r \hookleftarrow R, (m_0, m_1) \leftarrow A(r)$
$p \leftarrow P, m \leftarrow enc(p, k_1 \oplus bk_2)$
```

An adversary may have a chance to distinguish between outputs of encryption only if the distribution of k_1 and $k_1 \oplus k_2$ are different. If we reduce the problem to just one bit of key length, then we can easily observe that both distributions are uniform over $\{0, 1\}$. Now, we can extend this argument over any key length thanks to the independence of individual bits.

□

2.2.1 Mirai: Protocol-specific primitiva

To enable Cryptoverif to work with this scheme we will take advantage of the predefined `xor` macro and define a type key. Cryptoverif understands our function formally by the inputs and outputs, additionally, it takes into account the properties we provide it with.

Listing 2.1: Cryptoverif xor implementation

```
1  def MiraiAddition(key, key2, key3, add) {
2
3  fun add(key2, key3):key.
4
5  equation forall key1:key2,key2:key2; add(key1, key2) = add(
      key2, key1).
6
7  equiv(KeyGeneration(add))
8      k1 <-R key2;
9      k2 <-R key3;
10     OracleKey() := return(add(k1, k2))
11     <=(0)=>
12     k1 <-R key;
13     OracleKey() := return(k1).
14 }
```

In the Listing 2.1 we define a `MiraiAddition` macro[2] that formally considers three different types `key, key2, key3`. However, in our case, they will be the same. It defines a function `add` that takes two latter types and produces a result of type `key` (see line 3).

For this function, we write an equation that ensures that it is commutative (see line 5). We also know that generating two keys with uniform distribution and xoring them is the same as generating just one with uniform distribution,

[2]All Cryptoverif code used here is a part of attachments as described in A.1.

this is handled by the equivalence `KeyGeneration` (see line 7). This will allow Cryptoverif to replace a complicated generation of a key with a simpler one when processing the game.

2.2.2 Mirai: Oracles

In the Listing 2.2 we have a total of three different oracles that are maintained by three independent processes. Honest parties `A` and `B` joined by an adversary `Adv`. All oracles have access to variables of their respective processes, this allows us to have the same secret for `A` and `B`. `A` also has a message `m` that it encrypts and sends.

All oracles have specified input and output, input is specified inside parentheses following the name of the oracle, and output is specified by the **`return`** command. As seen on line 2 oracle `A` doesn't take any arguments and can execute its action without a prompt from other participants. It sends a message of type `A: party, B: party, cipher: key` that matches inputs of both oracle `B` and the adversary `Adv`.

Both `B` and the adversary react to this message, `B` decrypts the message and an adversary blindly guesses a key and tries to do the same. All three oracles record what version of the plaintext message reached them (see lines 4, 10, 18).

Listing 2.2: Cryptoverif oracles implementation

```
1  let processA(keyA:bytestring, m:bytestring) =
2    OA() :=
3      cipher <- xor(m, keyA);
4      event asent(m);
5      return(A, B, cipher).
6
7  let processB(keyB:bytestring) =
8    OB(=A, =B, cipher:bytestring) :=
9      let plain = xor(cipher, keyB) in
10     event breceived(plain);
11     return().
12
13 let processAdv() =
14   OAdv(=A, =B, cipher:bytestring) :=
15     let (k1:bytestring, k2:bytestring, k3:bytestring, k4:
          bytestring) = keygen() in
16     keyguess <- add(add(k1, k2), add(k3, k4));
17     let messageguess = xor(cipher, keyguess) in
18     event AdversaryGuess(messageguess);
19     return().
```

2.2.3 Mirai: Queries

The tracking of events allows us to make queries about what kind of message reached individual oracles. The following query instructs Cryptoverif to try to prove that if `B` decrypted a message `m`, `A` has sent the same message:

Listing 2.3: Mirai query

```
query m: key;
  event(breceived(m)) ==> event(asent(m)).
```

Cryptoverif will fail to make this proof because encryption by xoring is not secure against resending attacks. We haven't implemented interception of messages and resending attacks in our code, however, Cryptoverif also takes into account other than specified adversaries that have these abilities. Thus it is not thus necessary to always specify an adversary nor is it intended usage, but it makes our exemplary case more readable.

The following query asks Cryptoverif to prove that our adversary hasn't correctly guessed the correct message:

Listing 2.4: Mirai secrecy query

```
query m: key;
  (event(asent(m)) && event(AdversaryGuess(m))) ==> false.
```

Cryptoverif will succeed in proving this secrecy of `m` with an upper estimation of the chance of success of the adversary. The output looks like this:

Listing 2.5: Cryptoverif output

```
Proved event(asent(m)) && event(AdversaryGuess(m)) ==>
   false in game 10 up~to probability 1 / |key|.
```

We can see that we have obtained an upper bound that precisely estimates the chance of an adversary. Cryptoverif was able to make the necessary transformation and arrive at this result thanks to our primitive specified before. It is also interesting to inspect the final version of the game (it was shortened for the sake of readability):

Listing 2.6: Final game of Mirai

```
Game 10 is
     ...
     ((
       OA() :=
       cipher: key <- xor(message, sharedkey);
       event asent(message);
       return(A, B, cipher)
     ) | (
       OB(=A, =B, cipher_1: key) :=
       return()
     ) | (
       OAdv(=A, =B, cipher_2: key) :=
       k1_9 <-R key;
       event AdversaryGuess(k1_9);
       return()
     ))
```

Oracle `B` was effectively removed from the game as it is not a relevant party for this query. The adversary uses only a generated byte as its guess, which is independent of any message sent by `B`. This simplification was possible due to the identical distribution of uniformly drawn keys and a sum of three uniformly drawn elements `keyAdv` $\oplus$ `keyA` $\oplus$ `m`. Cryptoverif was able to deduce this thanks to the library-defined xor operation that enables these transformations.

2.3 Proof strategy

Cryptoverif attempts to find proof of its queries as described in the following pseudocode:

Listing 2.7: Pseudocode approximation of Cryptoverif proof strategy

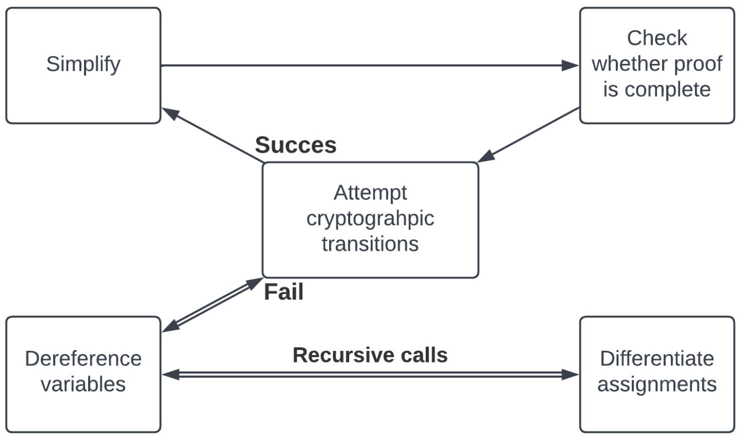
```
Simplify()
while "queries unsatisfied":
    CryptographicTransformations()
    Simplify()

def CryptographicTransformations():
    for transformation in possible_transformations:
        transformation.try()
        if transformation.successful:
            return

        advice = transformation.advice
        if advice is not None:
            advice()
```

Generally, if a function inside Cryptoverif fails to execute a transformation, it produces an *advice* - an instruction on what other transformation should be attempted before this one.

`advice` in the pseudocode above is a placeholder for two different functions. `RemoveAssign` and `SArename` are functions that execute semantic transformation on the game. The first one tries to remove assignments of a variable in order to create terms that are recognized by cryptographic transformations. however, if the term saved inside this to-be-removed variable includes different variables it may call itself again on these inner variables. It may also call `ASrename` if there are multiple definitions of a single variable, `SArename` will attempt to split the variable into several variables with different names if it is possible.


Simplify
Check whether proof is complete
Succes
Attempt cryptograhpic transitions
Fail
Dereference variables
Recursive calls
Differentiate assignments

The `Simplify` function goes through the game and collects information on all points of the game. This includes a list of defined variables and possible values of relevant boolean terms. This information may be further processed through a system of rewriting rules that is based on and extends the Knuth-Bendix completion algorithm[3]. The preprocessed data is then used in the manipulation of the game itself. This includes for example determining values of conditional expressions if the collected information supports only one option. It also restructures the code in order to limit the size of states in individual points, meaning that new definitions are pushed downward.

[3]The Knuth-Bendix completion algorithm creates from a set of rules a rewriting system that aims to find identical objects.

3. Emotet C&C protocol

To showcase a more advanced protocol, we will take a look at and implement an example from the Emotet family described by VMRay Labs Team [2022].

From the perspective of the malicious binary the cryptographic protocol runs approximately like this:

1. Using the Elliptic Curve Diffie-Hellman (ECDH) protocol, a shared secret is conceived.
2. Using SHA256 hash function, the AES256 key is derived from the shared secret.
3. The binary creates a message `m` and sends a packet of the form `ECDH_public_key|AES256(SHA256(m)|m)|random_bytes`

The fact that Emotet uses common primitives, allows us to take advantage of Cryptoverif's library. We even have access to the same elliptic curve which is used in the protocol. However, Cryptoverif does not describe ciphers and hash functions by their inner working but rather by their properties. Therefore we will use the following primitives to model points 1-3:

1. `DH_X25519`
2. `CollisionResistant_hash`
3. `IND_CPA_INT_CTXT_sym_enc`

`DH_X25519` is an implementation of this specific elliptic curve that is commonly used in elliptic cryptography. The primitive we will be using perfectly fits what is being used by Emotet and we don't need to make further assumptions.

`CollisionResistant_hash` is Cryptoverif's implementation of a one-input hash function resistant to a collision of hash values. SHA256 is currently widely used and consider to be collision-resistent, this primitive fits our needs well.

`IND_CPA_INT_CTXT_sym_enc` is an implementation of a generic cipher satisfying two requirements. The first one is that any two messages are indistinguishable under a chosen-plaintext attack (IND-CPA). The second requirement demands that an adversary having access to pairs of ciphertexts and plaintexts can't construct a ciphertext that would decrypt to any message. This is referred to as the integrity of ciphertext (INT-CTXT).

For an adversary to forge a ciphertext decrypting to `SHA256(m)|m` would mean that he has an advantage against SHA256 or AES256 such as knowing the key. Without any advantage, a random guess would have a chance to succeed of only 2^{-256} which is currently considered sufficiently safe. Otherwise an adversary would have to be able to construct collision of hash values or have access to encryption with the secret key, because the ciphertext determines hash value of the plaintext and plaintext itself.

Both SHA256 and AES256 are currently considered safe, thus, we will assume that this implementation of tagged encryption satisfies INT-CTXT.

3.1 Implementing the protocol

We will call the party representing an infected device `A` and a command unit `B`. Similarly to our Mirai implementation we will be using the oracle interface. This time we will take advantage of the `query secret` command without implementing an attacker manually. This is the implementation of the key exchange and derivation:

Listing 3.1: Emotet: key exchange and derivation

```
let processA(pubS: G, pubR: G, mess: text) =
  OA1() :=
    a~<-R Z;
    ga <- exp(g,a);
    return(A, B, ga);

  OA3(=B, =A, gb:G) :=
    shared <- exp(gb, a);
    aeskey <- sha(shared);
    ...

let processB(pubS: G, pubR: G) =
  OB2(=A, =B, ga:G) :=
    b <-R Z;
    gb <- exp(g,b);
    shared <- exp(ga, b);
    aeskey <- sha(shared);
```

`exp` is a function from the library implementation of the ECDH key exchange, together with a group element type `G`, and an exponent type `Z` they provide a simple way to access complicated properties of elements of an elliptic curve. `sha` is our adjustment of the library-defined hash function that can take a key as an argument, we define the key as constant inside our `sha` function. Rest of the interaction follows:

Listing 3.2: Emotet: encryption

```
let processA(pubS: G, pubR: G, mess: text) =
  OA1() :=
    a~<-R Z;
    ga <- exp(g,a);
    return(A, B, ga);

  OA3(=B, =A, gb:G) :=
    shared <- exp(gb, a);
    aeskey <- sha(shared);

    mess <-R text;
    cipher <- aes(mess, aeskey);

    event Asent(mess);
    return(A, B, cipher).

```

```
let processB(pubS: G, pubR: G) =

  OB4(=A, =B, cipher:text) :=
    let injbot(decipher) = deaes(cipher, aeskey) in
    event Breceived(decipher);
    return().
```

However, the code itself doesn't let us know what primitives lie below, so this is the complete interface DH_X25519 offers (as implemented in several of the examples provided in the distribution of Cryptoverif):

Listing 3.3: Elliptic curve interface

```
expand DH_X25519(
  (* types *)
  G,  (* Public keys *)
  Z,  (* Exponents *)
  (* variables *)
  g,    (* base point *)
  exp,  (* exponentiation function *)
  mult, (* multiplication function for exponents *)
  G8,
  g8,
  exp_div8,
  exp_div8', (* a~symbol that replaces exp_div8 after game
     transformation *)
  pow8,
  G8_to_G,
  is_zero,
  is_zero8
).
```

For Cryptoverif this is a very simple proof, in essence it is three secure cryptographic primitives chained well together without vulnerabilities toward public channels. This argument does not hold in general, nevertheless, in the case of Emotet the primitives are combined safely. The last game is a bit cryptic, but we still can obtain insight from inspecting it:

Listing 3.4: Emotet final game

```
Game 4 is
     Ostart() :=
     message <-R text;
     return();
     ((
       OA1() :=
       a~<-R Z;
       ga: G <- G8_to_G(exp_div8(g8, a));
       return(A, B, ga);
       OA3(=B, =A, gb_1: G) :=
       shared: G <- G8_to_G(exp_div8(pow8(gb_1), a));
       aeskey_1: aeskey <- hash(dummy, shared);
       mess_1 <-R text;
       r <-R enc_seed;
15        cipher_1: text <- enc_r(mess_1, aeskey_1, r);
16        event Asent(mess_1);
17        return(A, B, cipher_1)
18      ) | (
19        OB2(=A, =B, ga_1: G) :=
20        b <-R Z;
21        gb_2: G <- G8_to_G(exp_div8(g8, b));
22        shared_1: G <- G8_to_G(exp_div8(pow8(ga_1), b));
23        aeskey_2: aeskey <- hash(dummy, shared_1);
24        return(B, A, gb_2);
25        OB4(=A, =B, cipher_2: text) :=
26        let injbot(decipher: text) = deaes(cipher_2,
             aeskey_2) in
27        event Breceived(decipher);
28        return()
29      ))
```

We can see that Cryptoverif replaces our encryption function with random encryption (lines 13-15) as it is indistinguishable from the perspective of an adversary. This is a transition based on indistinguishability and proof of that would be very similar to the proof 2.2. In implementation, this is however hard-coded as a property of the IND_CPA_INT_CTXT and the encryption function of the IND_CPA_INT_CTXT primitive is considered random.

Besides that we can see symptoms of the proof strategy that was described toward the end of the previous chapter, most of our code was replaced with definitions as a result of RemoveAssign, this is especially notable on lines 8, 22.

4. Ransomware inpsired games

Until now, we have been considering encryption schemes run by malware for its own need for communication between different instances. This time we are looking at another encryption scheme but from a LockCrypt ransomware. Its purpose is to encrypt data on a victim's drive and then demand ransom for their decryption. We will consider two games modeling LockCrypt ransomware encryption in two different situations.

4.1 One-time pad data encryption

The vulnerability we want to show has been described in Harpaz [2018] and Malwarebytes Labs [2018]. LockCrypt first generates a large key buffer that is then used during encryption.

Harpaz [2018] describes in detail, how the key is derived. It uses a random number generator but that is not important for the observation we are trying to make. The key is then used in two encryption rounds as described by the following code:

Listing 4.1: LockCrypt xor pad generation round 1

```
def level1_crypt(file, file_size, key, key_size):
    iterations = file_size >> 2

    k = 0
    for i in range(4, file_size - 6, 2):
        if k > key_size:
            k = 0

        section = file[i:i+4]
        keypart = key[k:k+4]

        inp = int.from_bytes(section, byteorder='little')
        out = inp ^ int.from_bytes(keypart, byteorder='
           little')

        file[i:i+4] = out.to_bytes(4, byteorder='little')

        k += 4
```

Listing 4.2: LockCrypt xor pad generation round 2

```
def rol32(val, shift):
    return (val << shift) | (val >> (32 - shift))

def level2_crypt(file, file_size, key, key_size):
    iterations = file_size >> 2

    key_iterator = 0
    for i in range(4, iterations - 6, 4):
        if key_iterator > key_size:
            key_iterator = 0

        section = file[i:i+4]
        keypart = key[k:k+4]

        inp = endian_change('<I', section)[0]
        inp = rol32(inp, 5)
        out = inp ^ endian_change('<I', keypart)[0]
        out = endian_change('>I', out)

        file[i:i+4] = out
        key_iterator += 4
```

The bodies of the two `for` cycles in both rounds describe what happens to a 4-byte section of a plaintext. `inp` represents the impacted plaintext section, `out` is its value after encryption and `keypart` is 4-byte long section of the pre-generated key.

We can see that during the first round the stride is 2 bytes. Because of this, most bytes (except for the first few bytes) are altered twice during this round in two consecutive iterations. In the second round we can see that the stride has increased to 4 bytes, therefore, every byte is altered just once by this round. This time the round uses rotation and position swapping. These are fortunately only permutations, which means we can easily reverse them.

The key does not rely on the to-be-encrypted file at all, thanks to this we have a constant key schedule that is the same for all files. Each bit of the plaintext is xored together with up to 3 predetermined bits of the key schedule. We can reverse the permutations and with access to large enough plaintext, we can determine the key schedule that is the same for all files. There are a few edge cases that are better described in Harpaz [2018] but these concern only a small number of bytes.

Suppose then that we indeed have access to a large enough plaintext of an encrypted file and another encrypted file we want to decrypt. We can model this situation easily enough, for simplicity we presume that the key schedule is completely random but constant across files:

Listing 4.3: Multiple uses of xor pad

```
let processA(fileA :bytes, fileB :bytes) =
  OAttacker() :=
    pad <-R bytes;
    encA <- xor(fileA, pad);
    encB <- xor(fileB, pad);
    return(encA, encB).
```

Once again, we will be taking advantage of the pre-defined xor makro and alongside it the `query secret` command provided by Cryptoverif, however, this time we will explicitly provide public variables:

Listing 4.4: Query

```
expand Xor(bytes, xor, zero).

```

```
query secret unbacked_file public_vars backed_file.
```

Cryptoverif can't prove the secrecy of the file that we want to encrypt, this is due to the repeated usage of the key pad. Cryptoverif output looks like this:

Listing 4.5: Cryptoverif output: several times used pad

```
Game 1 is
     Ostart() :=
     backed_file <-R bytes;
     unbacked_file <-R bytes;
     return();
     OAttacker() :=
     pad <-R bytes;
     encA: bytes <- xor(backed_file, pad);
     encB: bytes <- xor(unbacked_file, pad);
     return(encA, encB)

RESULT Could not prove secrecy of unbacked_file with public
    variables backed_file.
```

Now let us represent contents of a computer's filesystem as a single file. Naturally, now Cryptoverif can, thanks to not having a second file dependent on `pad` variable, replace it with a randomly drawn element to illustrate the secrecy property we are seeking.

Listing 4.6: Cryptoverif output: one-time pad

```
Game 4 is
     Ostart() :=
     unbacked_file <-R bytes;
     return();
     OAttacker() :=
     pad_1 <-R bytes;
     return(pad_1)


Proved secrecy of unbacked_file in game 4
Adv[Game 1: secrecy of unbacked_file] <= 0 + Adv[Game 4:
   secrecy of~unbacked_file]
Adv[Game 4: secrecy of unbacked_file] <= 0
RESULT Proved secrecy of unbacked_file
All queries proved.
```

The final version of the one-time pad game has been significantly changed by Cryptoverif's routines. It has used the assumption that a xor product of two uniformly drawn variables has the same distribution as one such variable as long as no other variables or terms depend on them. It was thus able to replace `unbacked_file` with newly assigned to variable `pad_1`.

We can see the assumption used in the default library of primitives (lines 11-15):

Listing 4.7: Xor assumption

```
def Xor(D, xor, zero) {
```

```

param nx.

fun xor(D,D):D.
const zero: D.
equation builtin ACUN(xor,zero).

(* Xor is a one-time pad *)

equiv(remove_xor(xor))
      foreach ix <= nx do a <-R D; Oxor(x:D) := return(xor(
         a,x))
      <=(0)=>
      foreach ix <= nx do a <-R D; Oxor(x:D) := return(a).
}
```

The `builtin ACUN` is equation macro for pairs `(f, zero)` where `f` is a binary function and `zero` is an element from the set, where `f` is defined. It specifies that the function is commutative, $\forall x : x\texttt{f}x = \texttt{zero}$ and a few other properties.

Conclusion

We have connected the theory behind the implementation of cryptographic games with underlying mathematical concepts. We then applied these concepts to real-world cryptographic protocols used by malware in order to provide their formal analysis.

In contrast to the usual process of designing new cryptographic protocols and primitives, malware authors often resort to modifications of known designs or even design their own cryptosystems. While our approach is sometimes limited by the necessity to use certain level of abstraction, it still provided formal approach to an analysis of such objects. Interestingly, some of the chosen examples were out of the Cryptoverif's scope. Nevertheless, the tooling itself was still sufficient to hint at a proof of vulnerability.

The advantage of reliance on cryptographic games and in extension to Cryptoverif is that the proofs are usually easy to interpret and well-traceable. Since the design of protocols and primitives is rather special in comparison to the mainstream cryptography, there is a potential for improvement by building upon Cryptoverif's library and extending it with other commonly used primitives.

Bibliography

B. Blanchet. Cryptoverif: A computationally-sound security protocol verifier, 2017. URL https://bblanche.gitlabpages.inria.fr/CryptoVerif/cryptoverif.pdf. [Online; accessed 5-May-2023].

B. Blanchet and Cadé D. Cryptoverif: Computationally sound, automatic cryptographic protocol verifier user manual, 2022. URL https://bblanche.gitlabpages.inria.fr/CryptoVerif/manual.pdf. [Online; accessed 5-May-2023].

Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, October–December 2008. Special issue IEEE Symposium on Security and Privacy 2006. Electronic version available at http://doi.ieeecomputersociety.org/10.1109/TDSC.2007.1005.

Bruno Blanchet. personal communication, 2023.

Tomer Harpaz. Decrypting the LockCrypt ransomware, 2018. URL https://unit42.paloaltonetworks.com/unit42-decrypting-lockcrypt-ransomware/. [Online; accessed 6-May-2023].

J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* First Edition. CRC PRESS, New York, 2007. ISBN 1584885513.

Malwarebytes Labs. LockCrypt ransomware: weakness in code can lead to recovery, 2018. URL https://www.malwarebytes.com/blog/news/2018/04/lockcrypt-ransomware. [Online; accessed 6-May-2023].

Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Paper 2004/332, 2004. URL https://eprint.iacr.org/2004/332. https://eprint.iacr.org/2004/332.

A. Středa and J. Neduchal. Let's play Hide 'N Seek with a botnet., 2018. URL https://blog.avast.com/hide-n-seek-botnet-continues. [Online; accessed 5-May-2023].

VMRay Labs Team. Malware analysis spotlight: Emotet's use of cryptography, 2022. URL https://www.vmray.com/cyber-security-blog/malware-analysis-spotlight-emotets-use-of-cryptography/. [Online; accessed 5-May-2023].

A. Attachments

A.1 Code repository

Attached is a zip file that contains all code written for this thesis together with instructions on how to use it. It is also accessible at:
github.com/MedOndrej/Vulnerabilities-Security-proofs-of-malware-protocols