MTAT.07.003 Cryptology II

Reduction Types

Sven Laur
University of Tartu

Motivation

Security of most cryptographic constructions is based on *intractability*.

- ▷ So far provable lower bounds are *trivial* for all computational problems.
- ▷ It is also *highly* unlikely that such proofs *do* exist in a *compact* form.

Hence, it is *impossible* to prove security of cryptographic constructions.

- ▷ We can prove security only with respect to *intractability assumptions*.
- ▷ All cryptographic proofs reduce a new problem to *known* problems.
- ▷ The exact nature of security guarantees depends on a *paradigm*.
- ▷ However, a *decay* in security compared to *basic primitives* is inevitable.

In this course, we do not question the *validity* of common cryptographic assumptions nor study how to device *intractable* computational problems.

Classical Reductions

Many-one reductions

Common computational problems are puzzles in the following form.

▷ Find a solution (*witness*) w for a *puzzle* x such that $(x, w) \in A$.

If we can convert any puzzle x of a type A into a puzzle $f(x)$ of a type B such that solution to puzzle $f(x)$ implies solution to puzzle x

$$\forall x \in \{0,1\}^* : (\exists u : (f(x), u) \in B) \Rightarrow (\exists w : (x, w) \in A) \enspace ,$$

then we have a *many-one reduction* $A \ \leq_m B$.

Now the properties of f determine the usefulness of the *reduction*.

▷ The efficiency of f determines the closeness of puzzles A and B.

▷ Correspondence between witnesses determines structural properties.

EdgeCover and SetCover problems

EdgeCover Problem:

- ▷ Given a graph $G = (V, E)$ find a minimal set of edges C such that all vertices are covered: $\forall u \in V\ \exists v \in V : \{u, v\} \in C$.
- ▷ Given a graph $G = (V, E)$ and a number k is there a set of edges C such that all vertices are covered and $|C| \leq k$.

SetCover Problem:

- ▷ Given a universe of sets $\mathcal{U} = \{S_1, \ldots, S_n\}$ find a minimal set of sets $\mathcal{C} \subseteq \mathcal{U}$ such that $\mathcal{C}$ contains all elements of $\mathcal{U}$: $\bigcup_{S \in \mathcal{U}} S = \bigcup_{S \in \mathcal{C}} S$.
- ▷ Given a universe of sets $\mathcal{U} = \{S_1, \ldots, S_n\}$ and a number k is there set of sets $\mathcal{C} \subseteq \mathcal{U}$ such that all elements are covered and $|W| \leq k$.

EdgeCover $\leq_m$ SetCover

Reduction. Given a connected graph $G = (V, E)$, let the universe $\mathcal{U}$ consist of all edges $\mathcal{U} = E$. Then the set of vertices V consists of all elements.

- ▷ For obvious reasons, edge cover and set cover coincide.
- ▷ A time to compile one puzzle to another is linear is the size of the graph.
- ▷ A time to detect non-connected graphs is $\mathrm{O}(|E| \cdot |V|)$.

Questions

- ▷ Is this reduction tight?
- ▷ Does the reduction preserve the structure of the problem?
- ▷ Does there exist a reduction to other direction?

Black-box reductions

Many-one reductions are quite restrictive, as they act as *compilers*.

- ▷ They cannot be used for interactive protocols.
- ▷ Sometimes it makes sense to call a solver out several times.

Let $\mathcal{B}$ be a solver for a puzzle of type B. Then an algorithm $\mathcal{A}$ that uses $\mathcal{B}(\cdot)$ as an *oracle* to solve a puzzle A is known as a *black-box reduction*.

- ▷ If the algorithm $\mathcal{A}$ is deterministic then $\mathcal{A}^{\mathcal{B}}$ must always output a correct answer in *reasonable* time for all valid inputs x.
- ▷ If the algorithm $\mathcal{A}$ is randomised then the success of $\mathcal{A}^{\mathcal{B}}$ must be *reasonably* large for all *reasonable* solvers $\mathcal{B}$ and all valid inputs x.

The exact meaning and security implications of a black-box reduction depends on what is considered reasonable in the security analysis.

Deterministic reductions

Most deterministic reductions are just *code wrappers*, which adjust inputs so that a solver $\mathcal{B}$ can process them without problems.

Discrete Logarithm. Let $\mathbb{G} = \langle g \rangle$ be a multiplicative group generated by the element g. Then for any elements $y, z \in \mathbb{G}$ the discrete logarithm $\log_z y$ is defined as the smallest integer x such that $z^x = y$ and $\perp$ if $y \notin \langle z \rangle$.

An example. If there exists an algorithm $\mathcal{B}$ that can compute $\log_g y$ for all $y \in \mathbb{G}$, then there exists an algorithm $\mathcal{A}$ that can compute $\log_z y$ and the running time of $\mathcal{A}$ is roughly twice as long as the running time of $\mathcal{B}$.

Proof. Consider the following construction:

$$\mathcal{A}^{\mathcal{B}}(y, z)$$

$$\Big[\ \textbf{return}\ \mathcal{B}(y) \cdot \mathcal{B}(z)^{-1}$$

Randomised reductions

Not all algorithms are equally successful for all inputs. Hence, it makes sense to define *advantage* over a subset of all puzzles $X \subseteq \{0,1\}^*$:

$$\mathsf{Adv}_X^{\mathsf{succ}}(\mathcal{A}) = \Pr\left[x \leftarrow X, w \leftarrow \mathcal{A}(x) : \ (x,w) \in A\right] \ .$$

Similarly, we can talk about average time-complexity of the algorithm $\mathcal{A}$.

Most randomised reductions provide following type of closeness guarantees

$$\mathsf{Adv}_Y^{\mathsf{succ}}(\mathcal{B}) \geq \varepsilon \qquad \Longrightarrow \qquad \mathsf{Adv}_X^{\mathsf{succ}}(\mathcal{A}^{\mathcal{B}}) \geq \rho(\varepsilon)$$

provided that ε is not *negligible* (cannot be ignored).

Random self-reducibility

A puzzle is *randomly self-reducible* if we can efficiently reduce any problem instance to a uniformly chosen instance. As a result, the worst-case running time and average-case running time are tightly connected.

Theorem. Discrete logarithm problem is randomly self-reducible.

PROOF. Let $\mathcal{B}$ be an algorithm for computing discrete logarithm and q the size of the group $|\mathbb{G}|$. Then the following randomised algorithm

$$\mathcal{A}^{\mathcal{B}}(y)$$
$$\left[\begin{array}{l} x \xleftarrow[u]{} \mathbb{Z}_q \\ \textbf{return } \mathcal{B}(y \cdot g^x) - x \end{array}\right.$$

behaves identically for all inputs and the expected running time is roughly the average-case complexity of the algorithm $\mathcal{B}$.

White-box reductions

Oracle calls to a sub-routine $\mathcal{B}$ might lead to sub-optimal solution, as it might be possible to optimise the code $\mathcal{A}^{\mathcal{B}}$ further by analysing $\mathcal{B}$.

More formally, a *white-box* reduction is a mapping $\mathcal{B} \mapsto \mathcal{A}_{\mathcal{B}}$ such that $\mathcal{A}_{\mathcal{B}}$ is *reasonably* efficient and successful for all *reasonable* solvers $\mathcal{B}$.

▷ The correspondence *does not have to* be efficiently computable.

Let $\mathcal{A}_*$ be an optimal solver. Then the white-box reduction $\mathcal{A}_{\mathcal{B}} \equiv \mathcal{A}_*$ is the best reduction we can propose. However, it is *nearly useless*, since it does not *connect* the puzzles A and B in any way.

▷ Useful white-box reductions are strictly constructive.

▷ Not many white-box reductions are known.

▷ White-box reductions are not allowed by *some paradigms*.

Models of Computation

Algorithms and strategies

A *randomised function* also known as *randomised strategy* is a mapping

$$f : \{0,1\}^* \times \Omega \to \{0,1\}^*$$

where Ω is a *randomness space*, i.e., the output $f(x) = f(x;\omega)$ depends on a *non-deterministic choice* $\omega \in \Omega$.

A *randomised algorithm* $\mathcal{A} : \{0,1\}^* \times \Omega \to \{0,1\}^*$ is a randomised function that has a finite, precise and complete description:

▷ a Boolean circuit or a circuit family (*hardware design*),

▷ a program for an ordinary computer (*finite automaton*),

▷ a program for an idealised computing device:
- ◇ a program for universal Turing Machine,
- ◇ a program for universal Random Access Machine.

Universal Turing Machine

Universal Turing Machine is a Turing Machine that takes in

- ⋄ a program code ϕ,
- ⋄ arguments $x_1, \ldots, x_n$,
- ⋄ randomness $\omega \in \{0,1\}^*$

and outputs either a single value or vector.

The cells of a random tape ω are filled by tossing a fair coin: $\omega_i \xleftarrow[u]{} \{0,1\}$.

Universal Turing Machine may also read dedicated network tapes:

- ⋄ a single read only tape for incoming messages,
- ⋄ a single write only tape for outgoing messages.

Universal Random Access Machine

Universal Random Access Machine is an idealised computing device:

- ▷ It has infinite number of data registers $\mathsf{R}[0], \mathsf{R}[1], \mathsf{R}[2], \ldots$.
- ▷ It has infinite number of code registers $\mathsf{C}[0], \mathsf{C}[1], \mathsf{C}[2], \ldots$.
- ▷ It has a program counter PC
- ▷ It has a stack pointer SP

At the beginning a program is loaded form the tape to the code registers and PC and SP is set to zero. Next the following loop is executed:

- ▷ Read and interpret command at location $\mathsf{C}[\mathsf{PC}]$
- ▷ Halt if $\mathsf{C}[\mathsf{PC}]$ is zero.

Interpreted commands form a simple assembly-like language.

Time-complexity

Let $\mathcal{A}$ be a randomised algorithm and let $t(x, \omega)$ denote the number of elementary steps that are needed to obtain $\mathcal{A}(x, \omega)$.

Then for each input we can define:

▷ average running time $\mathbf{E}\left[t(x)\right]$,

▷ maximal running time $\max_{\omega \in \Omega} t(x, \omega)$.

Similarly, for all $\Bbbk$-bit inputs we can define:

▷ average running time $\mathbf{E}\left[t\right]$ if we fix distribution over inputs $x \in \{0, 1\}^{\Bbbk}$,

▷ maximal running time $\max_{x \in \{0,1\}^{\Bbbk}} \max_{\omega \in \Omega} t(x, \omega)$.

Finally, we can consider a *t-time algorithm* $\mathcal{A}$ that is halted after t elementary steps. The corresponding invalid output is denoted by $\perp$.