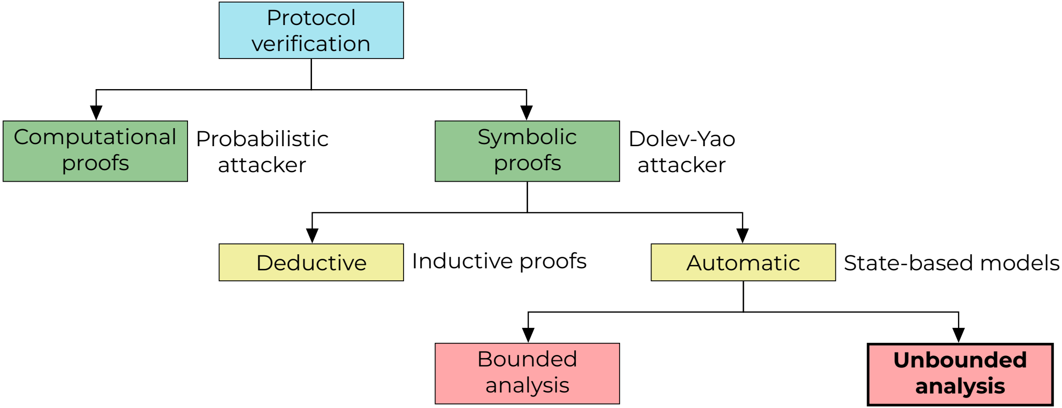
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Formal Analysis of Real-World Security Protocols

Lecture 1: Terms and Equational Theories

Aleksi Peltonen | 2024-10-28

Recap: protocol verification



Recap: formal verification

Recall our goals:

- We want to **formally analyze (real-world) protocols**
- Find attacks, or prove that protocols are “secure”

To achieve this, we plan to:

- Capture all possible interactions between the protocol and an attacker in a mathematical model M
- Formally state our desired security goal φ in terms of this model
- Then, either **prove that** $\mathrm{M} \vDash \varphi$ or **find a counterexample** that shows $\neg(\mathrm{M} \vDash \varphi)$ (i.e., an attack)

The models are complex, but designed with **automation** in mind

Model components

What **components** do we need to model protocols?

1. All possible sent and received messages
2. All possible protocol behaviors
3. The attacker
4. Security properties that we want to verify

Lecture 1

Lecture 2

Lecture 3

This lecture

Modeling Messages as Terms

Equational Theories

Equational Theories for Cryptographic Primitives

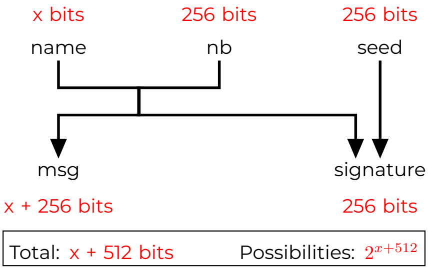
Term Rewriting

Modeling Messages as Terms

Messages

- In reality, protocols send and receive **bitstrings**
- We can model this… but we don't know how to **automate** the resulting analysis
- Observation: maybe we don't care about all bitstrings, only some are relevant
- Choice: **focus on how the bits were computed**, not on values

Alice is expecting a message containing her name, a new random value, and a signature with Bob's private key. Which messages would Alice accept?



Messages

Very informal intuition

- If someone generates a new large random value, we do not care about the actual bits, only that it is “fresh” and is unlikely to match any other bitstrings we have seen
- The only way anyone can find the bitstring `hash(‘Dog’)` is by being incredibly lucky, or by computing this hash themselves from the string `‘Dog’` **We omit luck from our model!**
- Similarly for encryptions and signatures: outputs look like random bits; negligible chance to coincide with other computed values

Signatures

A **signature** describes the non-logical symbols of a formal language.

Formally, a signature Σ is a set of **function symbols** and a function $Ar : \Sigma \to \mathbb{N}$. Function symbols of arity 0 are called **constants**.

Example

$\Sigma = \{Alice, Bob, Charlie, hash, pair, exp\}$, where $Alice$, Bob, and $Charlie$ are constants (i.e., $Ar(Alice) = Ar(Bob) = Ar(Charlie) = 0$), and $hash$, $pair$, exp are functions with $Ar(hash) = 1$ and $Ar(pair) = Ar(exp) = 2$

Terms

A **term** is recursively constructed from constants, variables, and function symbols.

Example

t = $(x + y) \times (1 + z)$ is a term built from the constant 1, variables x, y, and z, and the function symbols $+$ and $\times$

Let Σ be a signature, $\mathcal{V}$ a set of variables, and $\mathcal{C}$ a set of constants. We call the set $\mathcal{T}_\Sigma(\mathcal{V} \cup \mathcal{C})$ the **term algebra** over Σ.

We can use terms to *represent messages!*

Modeling messages as terms

Terms represent messages by the way they were constructed.

Basic terms: Alice, Bob, x, y, z, ServerNonce, ‘some_string’

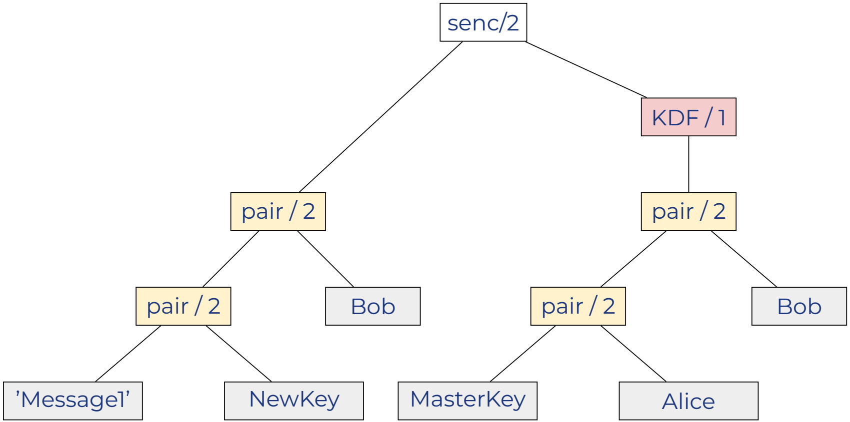
Function symbols: pair/2, exp/2, hash/1, sign/2, verify/3
senc/2, sdec/2

We often use common shorthands in tools and writing:

<x, y>	for	pair(x, y)
<x, y, z>	for	pair(pair(x, y), z)
x^y	for	exp(x, y)

Terms are trees

senc(<'Message1', NewKey, Bob>, KDF(<MasterKey, Alice, Bob>))



Variables and types

- Terms can contain **variables** (e.g., hash(X))
- Variables can have *type annotations* that restrict the possible values that they can be instantiated with:
  - X is a variable without type annotation
  - ~X is a *fresh* variable that can only be instantiated with randomly generated values
  - $X is a public variable that can only be instantiated with values that are known to all parties
- We use terms to..
  - ..construct **sent and received messages** according to a protocol specification
  - ..determine the **attacker's knowledge**: Which terms does the attacker know? Which terms can the attacker construct?

Example: Basic attacker derivation

- Assume that the attacker starts out knowing all public information:
  - Public constants, text strings, public variables:
    - Alice, Bob, ‘alice’, ‘bob’, ~nonce, $identifier, . . .
  - Algorithms:
    - senc, sdec, hash, KDF, . . .
- The attacker can generate new fresh values
- From the known values, the attacker can compute e.g.,
  - senc(hash(<Alice, ‘alice’>), KDF(AttackerKey))
- After learning a fresh term NonceBob from a message, the attacker can also compute e.g.,
  - senc(hash(<Bob, NonceBob>), KDF(AttackerKey))

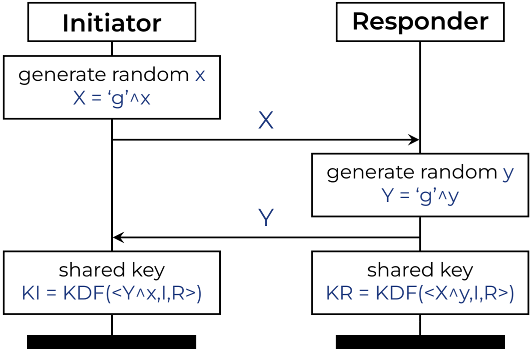
Syntactic equality

- **Syntactic equality:** two terms are the same, if and only if they are syntactically equivalent
- We decided earlier that the outputs of cryptographic primitives are supposed to look random. Syntactic equality encodes this intuition

'dog'	$=$	'dog'	
'dog'	$\neq$	hash(X)	for all X
hash(X)	$=$	hash(Y)	if and only if X = Y
senc(m1, k1)	$=$	senc(m2, k2)	if and only if m1 = m2 and k1 = k2
senc(m, k)	$\neq$	hash(X)	for all m, k, X

Example: Diffie-Hellman key exchange

Now, with a more formal syntax for messages, we can revisit our example from the previous lecture.



Normal execution: key is secret

- In a normal execution, $x \neq y$ (i.e., both are freshly generated)
  - The initiator derives KI = KDF(Y^x)
  - The responder derives KR = KDF(X^y)
- The (network) attacker knows
  - X = ‘g’^x
  - Y = ‘g’^y
- ..but has no way to construct KI or KR without knowing
  - X^y = (‘g’^x)^y, or Y^x = (‘g’^y)^x
- ..which it cannot, since there is no way to extract x or y
- This corresponds to the *hardness of the discrete logarithm problem*

Exponentiation as expected?

- In a normal execution, $x \neq y$ (i.e., both are freshly generated)
  - The initiator derives KI = KDF(Y^x)
  - The responder derives KR = KDF(X^y)
- By syntactic equality:
  - KI = KR if and only if (‘g’^y)^x = (‘g’^x)^y
- Since $x \neq y$, *this never holds*
- Thus, in normal execution of this model, **the initiator and the responder compute different, non-equal terms**
- We need something else to fix this

Equational Theories

Equational theories

An **equational theory** is a set of rules that determine which terms are considered equivalent.

Motivation:

- Some messages (such as exponentiation) can be constructed in more than one way
- Convenient modeling for cryptographic primitives
- Allows us to model degenerate cases of cryptographic primitives

Equational theories

Definitions:

- An **equation** over the signature Σ is a pair of terms $s, t \in \mathcal{T}_\Sigma(\mathcal{V})$ that defines when the terms are considered equal
  - For example, instead of modeling exponentiation, we use an equation to define the expected equality: (X^Y)^Z = (X^Z)^Y
  - …which implies (‘g’^y)^x = (‘g’^x)^y
- An **equational theory** is a tuple (Σ, E) of a signature Σ and a set of equations E

Pairing

For pairing, we use an equational theory to model splitting pairs.

Functions symbols:

pair/2	pair two terms (pair(x, y) is often written as <x, y>)
fst/1	extract the first element from a pair
snd/1	extract the second element from a pair

Equational theory:

fst(<x, y>)	=	x
snd(<x, y>)	=	y

Equational Theories for Cryptographic Primitives

Tamarin’s built-in equational theories

Name	Description
`hashing`	Defines a hash function `h`
`asymmetric-encryption`	Asymmetric encryption
`symmetric-encryption`	Symmetric encryption
`signing`	Basic signatures
`revealing-signing`	Signatures that allow plaintext extraction
`multiset`	Multisets (bags) in messages
`xor`	Exclusive-or
`diffie-hellman`	Diffie-Hellman style exponentiation
`bilinear-pairing`	Bilinear pairing
`natural numbers`	Natural numbers and counters

Tamarin's built-in equational theories

Name	Description
`hashing`	Defines a hash function `h`
`asymmetric-encryption`	Asymmetric encryption
`symmetric-encryption`	Symmetric encryption
`signing`	Basic signatures
`revealing-signing`	Signatures that allow plaintext extraction
`multiset`	Multisets (bags) in messages
`xor`	Exclusive-or
`diffie-hellman`	Diffie-Hellman style exponentiation
`bilinear-pairing`	Bilinear pairing
`natural numbers`	Natural numbers and counters

Basic symmetric encryption

For basic symmetric encryption schemes, we use an equational theory to model decryption.

Functions symbols:

senc/2	encrypt a message using a key
sdec/2	decrypt a message using a key

Equational theory:

sdec(senc(m, k), k) = m

Basic asymmetric encryption

For basic asymmetric encryption schemes, where the public key can be computed from the private key, we use an equational theory to model decryption.

Functions symbols:

aenc/2	encrypt a message using a public key
adec/2	decrypt a message using the private key
pk/1	compute the public key from a private key

Equational theory:

adec(aenc(m, pk(sk)), sk) = m

Basic signature scheme

For basic signature schemes, we use an equational theory to model signature verification.

Functions symbols:

sign/2	sign a message with a (private) signing key
verify/3	verify a signature for a message and a verification key
pk/1	compute the verification key from signing key
true	a constant representing 'true'

Equational theory:

verify(sign(m, sk), m, pk(sk)) = true

Diffie-Hellman

Diffie-Hellman modular exponentiation is a complex example.

Functions symbols:

^	exponentiation in the group (modulo some large prime)
*	multiplication
inv/1	inverse
1	a constant representing ‘1’

Equational theory:

(x ^y) ^z	= x ^(y * z)	x ^1	= x	x * y	= y * x
(x * y) * z	= x * (y * z)	x * 1	= x	x * inv(x)	= 1

Exclusive-or

Functions symbols:

$\oplus$	exclusive-or of two terms
zero	a constant representing an all-zeroes bitstring

Equational theory:

$x \oplus y = y \oplus x$

$(x \oplus y) \oplus z = x \oplus (y \oplus z)$

$x \oplus \text{zero} = x$

$x \oplus x = \text{zero}$

Note: this is a very coarse approximation of xor that does not work well with other primitives and needs to be handled with care.

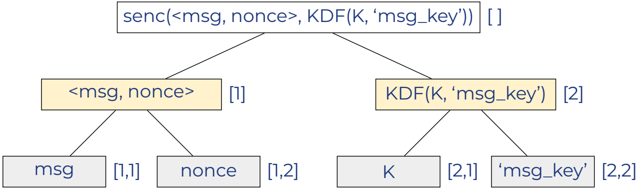
Further primitives

- **The previous were only basic examples** corresponding to some of Tamarin's built-in schemes
- Tamarin (and other symbolic tools) can handle many more primitives, such as
  - Multisets, blind signatures, bilinear pairings, . . .
- We can also provide much more accurate model of various different signature schemes, Diffie-Hellman groups, or elliptic curves, etc.
- **We will return to user-specified equational theories later in the course!**

Term Rewriting

Positions

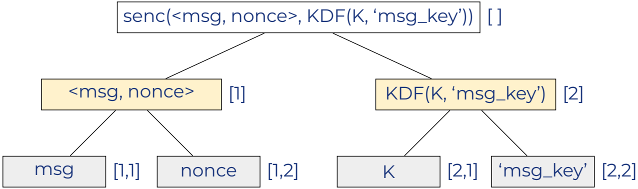
- Recall from earlier that terms are structured as *trees*
- Each node in the tree has a unique **position** (think path) indicating its place in the tree
- A position *p* is a sequence of natural numbers



Subterms

- Each position p of a term t is the start of a unique **subterm** $t|_p$

$$
\begin{aligned}
t|_{[]} &= \mathrm{senc}(<\mathrm{msg}, \mathrm{nonce}>, \mathrm{KDF}(K, \text{‘msg\_key’})) \\
t|_{[1]} &= <\mathrm{msg}, \mathrm{nonce}> \\
t|_{[2]} &= \mathrm{KDF}(\mathrm{K}, \text{‘msg\_key’}) \\
t|_{[1,1]} &= \mathrm{msg} \\
t|_{[1,2]} &= \mathrm{nonce} \\
t|_{[2,1]} &= \mathrm{K} \\
t|_{[2,2]} &= \text{‘msg\_key’}
\end{aligned}
$$



Substitutions

A **substitution** is a mapping $\sigma : \mathcal{V} \to \mathcal{T}$ from variables to terms.

We write $t\sigma$ to denote applying the substitution σ to the term t.

This replaces each variable x in t by the term $x\sigma$.

Example

For $t = \text{senc}(< \text{msg}, \text{nonce} >, x)$ and $\sigma = \{x \mapsto \text{KDF}(K, \text{‘msg\_key’})\}$, we can apply the substitution $x\sigma = \text{KDF}(K, \text{‘msg\_key’})$ to get $t\sigma = \text{senc}(< \text{msg}, \text{nonce} >, \text{KDF}(K, \text{‘msg\_key’}))$.

Unification and matching

Unification determines if two terms with variables can be made equal. Two terms u and v are said *unifiable* if there exists a substitution σ, called a *unifier*, such that $u\sigma = v\sigma$.

Example

The terms $t_1 = x$ and $t_2 = 2y$ are unifiable with e.g., $\sigma = \{x \mapsto 2, y \mapsto 1\}$

A term t_1 **matches** another term t_2 if there is a substitution σ s.t. $t_1 = t_2\sigma$

Example

The term $t_1 = 1$ matches the term $t_2 = y$ with $\sigma = \{y \mapsto 1\}$

Term rewriting

- A **rewrite rule** $l \to r$ over a signature Σ is an ordered pair of terms (l, r) with $l, r \in \mathcal{T}_\Sigma(\mathcal{V})$
  - Indicates that the left-hand side l can be replaced by the right-hand side r
  - Can be applied to a term s if the left term l matches some subterm of s, i.e., there is some substitution σ s.t., the subterm of s at position p is the result of applying σ to l
- The outcome is the result of replacing the subterm at position p in s by the term r with the substitution σ applied
- A **rewrite system** $\mathcal{R}$ is a set of rewrite rules

Term rewriting example

Example

The *distributive property* of binary operations is a rewriting rule, which states that $x \times (y + z) \to x \times y + x \times z$. Consider the term $s = a \times (b + 1)$ and the substitution $\sigma = \{x \mapsto a, y \mapsto b, z \mapsto 1\}$. We can apply the substitution σ and the rewrite rule $l \to r$ to obtain $t = a \times (b + 1) = a \times b + a \times 1 = a \times b + a$.

Summary

Summary

- We have now learned that..
    - **terms** can be used to represent messages,
    - **equations** specify when two terms are considered equal, and
    - **term rewriting rules** can be used to replace terms with other terms.
- Why is this important?
    - Tamarin models protocols as **multiset rewriting rules with equations**
    - We can model messages as ground terms!
- In the next lecture, we will learn about modeling states as **multisets of facts** and protocol executions as a **transition system** operating on them

Reading material

Recommended reading:
[Bas+25, Ch. 3–3.1.4, 7], [Mei13, Ch. 2]

[Bas+25] D. Basin, C. Cremers, J. Dreier, and R. Sasse. **Modeling and Analyzing Security Protocols with Tamarin: A Comprehensive Guide.** Draft v0.9.5. May 2025.

[Mei13] S. Meier. **Advancing Automated Security Protocol Verification.** PhD thesis. ETH Zurich, 2013.

Additional reading

Additional reading: [BN98, Ch. 2, 10–11], [Dre+18]

[BN98] F. Baader and T. Nipkow. **Term Rewriting and All That.** Cambridge University Press, 1998.

[Dre+18] J. Dreier, L. Hirschi, S. Radomirovic, and R. Sasse. **Automated Unbounded Verification of Stateful Cryptographic Protocols with Exclusive OR.** In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). 2018.