Simple Diffie-Hellman Algebra

John D. Ramsdell
The MITRE Corporation
CPSA Version 2.2.12

November 9, 2012

The current attempts to support an algebra for Diffie-Hellman are experimental and they all known to have flaws.

A natural way to model protocols that use Diffie-Hellman is with an algebra that includes a sort for exponents, one with members that form an Abelian group. Implementation experiments have shown that it is difficult to implement such an algebra within the current CPSA framework without significant revisions to CPSA.

This paper describes a simple Diffie-Hellman algebra in which the exponent is not an Abelian group. Instead, the algebra only captures the commutative law of exponents. The algebra requires few changes to the basic formalism used by CPSA [3]. An atom that originates in a trace need not be carried as long as it is in an exponent of something carried. Similarly, in a skeleton, an atom assumed to be uniquely originating need not be carried as long as it is in an exponent of something carried. (See Section 2).

The trade-off for ease of implementation is a loss of coverage. Protocols that make use of the associativity or the inverses of exponents cannot be correctly analyzed using this algebra. In particular, group Diffie-Hellman protocols typically cannot be handled. CPSA with the simple Diffie-Hellman algebra can analyze restricted Diffie-Hellman protocols, such as the basic Diffie-Hellman Key Exchange Protocol. This protocol serves as an example in this paper.

```
(herald "Diffie-Hellman Key Exchange" (algebra diffie-hellman))

(defprotocol dhke diffie-hellman
  (defrole init (vars (a b akey) (x y expn))
    (trace
     (send (enc "i" (exp (gen) x) (invk a)))
     (recv (cat (enc (exp (gen) y) (invk b))
                (enc a b (exp (exp (gen) y) x))))
     (send (enc "i" a b (exp (exp (gen) y) x))))
    (uniq-orig  x))
  (defrole resp (vars (a b akey) (x y expn))
    (trace
     (recv (enc "i" (exp (gen) x) (invk a)))
     (send (cat (enc (exp (gen) y) (invk b))
                (enc a b (exp (exp (gen) x) y))))
     (recv (enc "i" a b (exp (exp (gen) x) y))))
    (uniq-orig y)))

(defskeleton dhke (vars (a b akey))
  (defstrand resp 3 (a a) (b b))
  (non-orig (invk a) (invk b)))

 (defskeleton dhke (vars (a b akey))
   (defstrand init 2 (a a) (b b))
   (non-orig (invk a) (invk b)))
```

Figure 1: Diffie-Hellman Key Exchange CPSA Input

$$\begin{aligned}A \to B &: \{\!|\text{``i''}, \mathsf{g}^x\,|\!\}_{a^{-1}}\\ B \to A &: (\{\!|\,\mathsf{g}^y\,|\!\}_{b^{-1}}, \{\!|a, b|\!\}_{\mathsf{g}^{xy}})\\ A \to B &: \{\!|\text{``i''}, a, b|\!\}_{\mathsf{g}^{xy}}\end{aligned}$$

Alice (A) freshly generates an exponent x, signs the exponentiated value with her private uncompromised asymmetric key a^{-1}, and sends it to Bob (B). Bob freshly generates an exponent y, signs the exponentiated value with his private uncompromised asymmetric key b^{-1}, and sends it to Alice along with the public signing keys encrypted with the newly generated symmetric key g^{xy}. Alice confirms the symmetric key by signing the public keys too. Alice ensures her messages cannot be confused with Bob's by adding the tag constant "i" within her signed data.

The protocol in CPSA syntax is presented in Figure 1. CPSA concludes

Sorts:	$\top$, D, A, S, and E		
Subsorts:	$\mathsf{D} < \top$, $\mathsf{A} < \top$, $\mathsf{S} < \top$, and $\mathsf{E} < \top$		
Base Sorts:	D, A, and E (S and $\top$ omitted)		
Operations:	$(\cdot,\cdot)$	$: \top \times \top \to \top$	Pairing
	$\{\!\lvert \cdot \rvert\!\}_{(\cdot)}$	$: \top \times \mathsf{S} \to \top$	Symmetric encryption
	$\{\!\lvert \cdot \rvert\!\}_{(\cdot)}$	$: \top \times \mathsf{A} \to \top$	Asymmetric encryption
	"..."	$: \top$	Tag constants
	$(\cdot)^{-1}$	$: \mathsf{S} \to \mathsf{S}$	Symmetric key inverse
	$(\cdot)^{-1}$	$: \mathsf{A} \to \mathsf{A}$	Asymmetric key inverse
	g	$: \mathsf{S}$	Generator
	$(\cdot)^{(\cdot)}$	$: \mathsf{S} \times \mathsf{E} \to \mathsf{S}$	Exponentiation
Equations:	$(a^{-1})^{-1} \approx a \qquad s^{-1} \approx s \qquad (h^x)^y \approx (h^y)^x$		
	where $a\colon \mathsf{A}$, $s, h\colon \mathsf{S}$, and $x, y\colon \mathsf{E}$		

Figure 2: Simple Diffie-Hellman Algebra Signature and Equations

there is key agreement from the perspective of both Alice and Bob using this input. Notice that g^{xy} is written `(exp (exp (gen) x) y)`. There is no multiplication is this algebra, and g^{xy} is more accurately written as $(\mathsf{g}^x)^y$.

1 Order-Sorted Message Algebra

CPSA models a message as an equivalence class of terms over a signature. In particular, CPSA uses order-sorted quotient term algebras [1] for message algebras. This formalism enables the use of well-known algorithms for unification and matching in the presence of equations [4, Chapter 8] while providing a sort system for classifying messages.

CPSA provides a Diffie-Hellman algebra that extends the Basic Crypto Algebra with two new sorts, `base` and `expn`, and two new operations, `(gen)`, a constant of sort `base` for exponentiation, and `(exp` G X`)`, the exponentiation operation [2].

This paper presents a very simple message algebra for analyzing protocols using Diffie-Hellman, called the Simple Diffie-Hellman Algebra (SDHA), that suffices for this document. There are five SDHA sorts: $\top$, the sort of all messages, and sorts for data (D), asymmetric keys (A), symmetric keys (S), and exponents (E). Every symmetric key is a message (written $\mathsf{S} < \top$), and so forth for the other non-$\top$ sorts. The operations used to form terms are given by the signature in Figure 2. Notice that the encryption and key inverse operations are overloaded.

There are two equations in SDHA associated with key inverse. For asymmetric key $a\colon \mathsf{A}$, $(a^{-1})^{-1} \approx a$, and for symmetric key $s\colon \mathsf{S}$, $s^{-1} \approx s$. The equation for exponentiation is $(h^x)^y \approx (h^y)^x$, where $h\colon \mathsf{S}$ and $x, y\colon \mathsf{E}$. Unification and matching in this algebra produce a finite number of most general unifiers, that is, the unification type is finitary. For example, $\{x \mapsto u, y \mapsto v\}$ and $\{x \mapsto \mathsf{g}^v, u \mapsto \mathsf{g}^y\}$ unify x^y and u^v. Unification is finitary because of an approximation. To unify a^{xy} and b^{wz}, we unify a^{xy} with g^{uv} and g^{vu} with b^{wz}, where u and v are freshly generated variables. In other words, for the purpose of unification only, the equation for exponentiation is $(\mathsf{g}^x)^y \approx (\mathsf{g}^y)^x$. Expository Haskell code for unification and matching with just the equation for exponentiation is presented in Appendix A.

Origination assumption in roles and skeletons can only be applied to a subset of the messages of an algebra—the atoms. In SDHA, asymmetric keys and exponents are atoms, but symmetric keys are not. (Similarly, in the implemented Diffie-Hellman algebra, messages of sort `base` are not atoms.)

A message t_0 is *carried by* t_1, written $t_0 \sqsubseteq t_1$ if t_0 can be derived from t_1 given the right set of keys, that is $\sqsubseteq$ is the smallest reflexive, transitive relation such that $t_0 \sqsubseteq t_0$, $t_0 \sqsubseteq (t_0, t_1)$, $t_1 \sqsubseteq (t_0, t_1)$, and $t_0 \sqsubseteq \{\!|t_0|\!\}_{t_1}$.

The introduction of the Diffie-Hellman algebra requires a new relation on messages. A message t_0 is *held by* t_1, written $t_0 \preceq t_1$ iff t_1 carries t_0 or t_1 is in the exponent of a carried term. That is $\preceq$ is the smallest reflexive, transitive relation such that $t_0 \preceq t_0$, $t_0 \preceq (t_0, t_1)$, $t_1 \preceq (t_0, t_1)$, $t_0 \preceq \{\!|t_0|\!\}_{t_1}$, $t_1 \preceq t_0^{t_1}$.

2 Strand Spaces

A run of a protocol is viewed as an exchange of messages by a finite set of local sessions of the protocol. Each local session is called a *strand* [5]. The behavior of a strand, its *trace*, is a non-empty sequence of messaging events. An *event* is either a message transmission or a reception. In the Basic Crypto Algebra, a message originates in a trace if it is carried by some event and the first event in which it is carried is a transmission. For Diffie-Hellman, a message *originates* in a trace if it is held by some event and the first event in which it is held is a transmission.

A similar change was made for skeletons. An atom assumed to be uniquely originating need only be held by some term within a skeleton, it no longer must be carried.

3 Derivable Messages

Suppose T is a set of messages. Let $\rightarrow$ be a reduction relation on sets of messages defined as follows:

$$\begin{array}{l} \{(t_0, t_1)\} \cup T \rightarrow \{t_0, t_1\} \cup T \\ \{\{\!|t_0|\!\}_{t_1}\} \cup T \rightarrow \{t_0, \{\!|t_0|\!\}_{t_1}\} \cup T \quad \text{if } t_1^{-1} \in D(T) \text{ and } t_0 \notin T \end{array}$$

The minimum decryption set $M(T)$ is the normal form of relation $\rightarrow$, i.e. $T \rightarrow^* M(T)$ and there is no T' such that $M(T) \rightarrow T'$.

A message t is *derivable* from T, iff $t \in D(T)$, where

$$\begin{aligned} D^0 &= M(T) \\ D^{n+1} &= \left\{ \begin{array}{c|l} (x, y) & x, y \in D^n \\ \{\!|x|\!\}_y & x, y \in D^n, y\colon \mathsf{A} \vee y\colon \mathsf{S} \\ x^y & x, y \in D^n, x\colon \mathsf{S}, y\colon \mathsf{E} \end{array} \right\} \\ D(T) &= \textstyle\bigcup_{n \in \mathbb{N}} D^n \end{aligned}$$

4 Conclusion

In this algebra, one cannot analyze group Diffie-Hellman protocols, as they make use of the associativity of exponents. On the other hand, this algebra appears to allow the analysis of the basic Diffie-Hellman Key Exchange. In terms of implementation, allowances have to be made for the fact that unification may produce more than one most general unifier.

References

[1] Joseph A. Goguen and Jose Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[2] John D. Ramsdell. *CPSA User Guide.* The MITRE Corporation, 2009. In `http://hackage.haskell.org/package/cpsa` source distribution, `doc` directory.

[3] John D. Ramsdell, Joshua D. Guttman, Moses D. Liskov, and Paul D. Rowe. *The CPSA Specification: A Reduction System for Searching for Shapes in Cryptographic Protocols.* The MITRE Corporation, 2009. In `http://hackage.haskell.org/package/cpsa` source distribution, `doc` directory.

[4] Alan Robinson and Andrei Voronkov. *Handbook of Automated Reasoning.* The MIT Press, 2001.

[5] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1), 1999.

A Unification and Matching in Haskell

```
-- Unification and matching in a simple Diffie-Hellman algebra

module SimpleDiffieHellman where

-- Equational unification and matching in an algebra with the
-- following equation is used to analyze protocols that make use of
-- the Diffie-Hellman problem.
--
--      exp(exp(x, y), z) = exp(exp(x, z), y)
--
-- The module shows how to perform the unification and matching.

import Char(isDigit, isAlpha)

-- TERMS

-- Variables are just integers so that it is easy to freshly generate
-- them.

type Var = Int

data Term                       -- A term is
    = V Var                     -- a variable, or a
    | F String [Term]           -- function symbol and a list of terms

-- Equality modulo the equation: exp(exp(x, y), z) = exp(exp(x, z), y).
instance Eq Term where
    (V x) == (V y) = x == y
    (F "exp" [F "exp" [x, y], z]) == (F "exp" [F "exp" [x', y'], z']) =
         x == x' && y == y' && z == z' ||
         x == x' && y == z' && z == y'
    (F sym ts) == (F sym' ts') = sym == sym' && ts == ts'
    _ == _ = False

-- SUBSTITUTIONS

-- A substitution is a map from variables to terms
type Subst = [(Var, Term)]

-- Apply a substitution to a term
subst :: Subst -> Term -> Term
subst s (V x) =
    case chase s (V x) of
      V y -> V y
      t -> subst s t
subst s (F sym ts) =
    F sym (map (subst s) ts)

-- A substitution may contain an equivalence class of variables.  The
-- chase function finds the canonical representitive of the
-- equivalence class.
chase :: Subst -> Term -> Term
chase s (V x) =
    case lookup x s of
      Nothing -> V x
      Just t -> chase s t
chase _ t = t

-- UNIFICATION

-- This is the entry point
unify :: Term -> Term -> [Subst]
unify t t' =
    unify0 t t' []

-- Chase variables to start unifying two terms
unify0 :: Term -> Term -> Subst -> [Subst]
unify0 t t' s =
    unify1 (chase s t) (chase s t') s

-- Unification by case analysis
unify1 :: Term -> Term -> Subst -> [Subst]
unify1 (V x) (V y) s            -- Unify two variables
```

```
    | x == y = [s]              -- Nothing to do
    | x < y = [(y, V x) : s]    -- Substitute larger variable
    | otherwise = [(x, V y) : s] -- in preference to a smaller one
unify1 (V x) t s
    | occurs x t = []           -- Fail when x is in t
    | otherwise = [(x, t) : s]
unify1 t t'@(V _) s =
    unify1 t' t s
-- Unify using the Diffie-Hellman equation.
-- To make unification tractable, one makes use of the equation
-- exp(exp(gen(), x), y) = exp(exp(gen(), y), x).
unify1 (F "exp" ts@[u, v]) (F "exp" ts'@[u', v']) s =
    unifyList ts ts' s ++     -- Ordinary unification
    -- Add an instances of the equation, and unify on both sides
    do
      s' <- unifyList ts left s
      unifyList ts' right s'
    where
      -- Generate a fresh variable by looking at the variables in use
      var = nextVar ([u, v,  u', v'] ++ terms s) -- Include  substitution
      var' = var + 1                -- Generate another variable
      left = [F "exp" [F "gen" [], V var], V var'] -- One side of equation
      right = [F "exp" [F "gen" [], V var'], V var] -- And the other
unify1 (F sym ts) (F sym' ts') s -- Unify ordinary compound terms
    | sym /= sym' = []          -- Fail on symbol clash
    | otherwise = unifyList ts ts' s

unifyList :: [Term] -> [Term] -> Subst -> [Subst]
unifyList [] [] s = [s]
unifyList (t:ts) (t':ts') s =
    do
      s <- unify0 t t' s
      unifyList ts ts' s
unifyList _ _ _ = []

-- Find next unused variable in a list of terms
nextVar :: [Term] -> Var
nextVar [] = 0
nextVar ts =
    maximum (map nextVariable ts)
    where
      nextVariable (V x) = x + 1
      nextVariable (F _ ts) = nextVar ts

-- Returns the terms in a substitution.
```

```
terms :: Subst -> [Term]
terms s =
    [ t' |
      (x, t) <- s,
      t' <- [V x, t] ]

-- Does variable x occur in term t?
occurs :: Var -> Term -> Bool
occurs x (V y) = x == y
occurs x (F _ ts) = any (occurs x) ts

-- MATCHING

-- This is the entry point
match :: Term -> Term -> [Subst]
match t t' =
    match0 t t' []

-- Matching by case analysis
match0 :: Term -> Term -> Subst -> [Subst]
match0 (V x) t s =
    case lookup x s of
      Nothing -> [(x, t) : s]
      Just t' -> if t == t' then [s] else []
-- Match using the Diffie-Hellman equation
match0 (F "exp" [x, y]) (F "exp" [F "exp" [x', y'], z']) s =
    matchList [x, y] [F "exp" [x', y'], z'] s ++
    matchList [x, y] [F "exp" [x', z'], y'] s
match0 (F sym ts) (F sym' ts') s
    | sym /= sym' = []
    | otherwise = matchList ts ts' s
match0 _ _ _ = []

matchList :: [Term] -> [Term] -> Subst -> [Subst]
matchList [] [] s = [s]
matchList (t:ts) (t':ts') s =
    do
      s <- match0 t t' s
      matchList ts ts' s
matchList _ _ _ = []

-- TERM ORDERING

instance Ord Term where
    compare (V x) (V y) = compare x y
    compare (F "exp" [F "exp" [x, y], z])
            (F "exp" [F "exp" [x', y'], z']) =
        case (compare y z, compare y' z') of
          (GT, GT) -> compare [F "exp" [x, z], y] [F "exp" [x', z'], y']
          (GT, _) -> compare [F "exp" [x, z], y] [F "exp" [x', y'], z']
          (_, GT) -> compare [F "exp" [x, y], z] [F "exp" [x', z'], y']
          _ -> compare [F "exp" [x, y], z] [F "exp" [x', y'], z']
    compare (F sym ts) (F sym' ts') =
        case compare sym sym' of
          EQ -> compare ts ts'
          c -> c
    compare (V _) (F _ _) = LT
    compare (F _ _) (V _) = GT

-- TERM INPUT

instance Read Term where
    readsPrec _ s =
        readTerm s
        where
          readTerm s =
              [(V $ read (c:cs), t) | (c:cs, t) <- lex s,
                                      isDigit c] ++
              [(F (c:cs) ts, v)     | (c:cs, t) <- lex s,
                                      isAlpha c,
                                      ("(", u) <- lex t,
                                      (ts, v) <- readArgs u]
          readArgs s =
              [([], t)              | (")", t) <- lex s] ++
              [(x:xs, u)            | (x, t) <- reads s,
                                      (xs, u) <- readRest t]
          readRest s =
              [([], t)              | (")", t) <- lex s] ++
              [(x:xs, v)            | (",", t) <- lex s,
                                      (x, u) <- reads t,
                                      (xs, v) <- readRest u]

-- TERM OUTPUT

instance Show Term where
    showsPrec _ (V x) =
        shows x
    showsPrec _ (F sym ts) =
        showString sym . showChar '(' . showArgs ts
        where
```

```
showArgs [] = showChar ')'
showArgs (x:xs) = shows x . showRest xs
showRest [] = showChar ')'
showRest (x:xs) = showChar ',' . shows x . showRest xs
```