Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

TESIS

para obtener el grado de Ingeniero en Computación presentada por

Adrián Gerardo SILVEIRA LAPENNE

diciembre, 2014

Prueba de Seguridad para el esquema de firma Full Domain Hash utilizando EasyCrypt

---

Directores: **Dr. Alfredo Viola** y **Dr. Gustavo Betarte**
Supervisores: **Dr. Gilles Barthe** y **Dr. Santiago Zanella Béguelin**

Tribunal
Dr. Alberto Pardo, **Dr. Carlos Luna**, **Ing. Juan Diego Campo**

Agradecimientos. A mis tutores Alfredo Viola y Gustavo Betarte por su paciencia, apoyo y aportes. A Gilles Barthe y Santiago Zanella por su invitación a trabajar junto a ellos en IMDEA Software y sus discusiones útiles. A mis padres, mi hermano y Stella por su constante apoyo en esta tarea.

Resumen

EasyCrypt es una herramienta automatizada que permite elaborar pruebas de seguridad representadas como una secuencia de juegos. La misma es una posible candidata a ser adoptada cómo estándar por la comunidad criptográfica para asistir en la escritura y verificación de las pruebas. Desde el rol de criptógrafo, en este trabajo se presenta la prueba de la cota original del esquema de firma Full Domain Hash utilizando la herramienta, la cual constituye un aporte que permite evaluar la misma por parte de la comunidad.

Índice general

1 Introducción

Hoy en día, con el incremento de las comunicaciones y el avance de la computación, se ha vuelto cada vez más necesario proteger la información que se envía a través de canales inseguros. Aspectos tales como la privacidad, secrecía o autenticidad de la información se han vuelto críticos. El primer desafío en esta dirección es definir formalmente (o sea de manera precisa y usando un lenguaje matemático) estos conceptos.

La criptografía, en su sentido más amplio, usa herramientas matemáticas y computacionales con el objetivo de poder "controlar" la información. En este sentido es muy importante dar definiciones precisas de "seguridad" y definir modelos matemáticos apropiados que capten de la manera más rigurosa posible estas definiciones. En particular, el área de la seguridad demostrable ("provable security") se encarga de modelar y analizar distintos protocolos con ciertos objetivos de seguridad que un adversario intentará quebrar. Por ejemplo, en un protocolo de firma digital, un posible ataque de un adversario, consiste en hallar una firma válida para un mensaje que no haya sido firmado previamente.
Muchos protocolos son diseñados a partir de problemas que son difíciles de resolver y para los cuales no se conocen algoritmos eficientes (en tiempo polinomial en el tamaño del problema). Dos ejemplos concretos son el problema de factorización y el del logaritmo discreto en ciertos grupos. En este sentido, estos problemas se consideran "primitivas seguras"del protocolo. Es importante notar que el hecho de que un problema sea difícil no significa que toda instancia del problema sea difícil de resolver. Por ejemplo dado dos números primos aleatorios de 5 bits (eg: 29 y 31) es trivial factorizar su producto sin conocer ninguno de los factores. Por otro lado, si se toman dos números primos aleatorios de 2048 bits, no se conocen algoritmos polinomiales tales que dado el número de 4096 bits producto de estos dos primos lo factorice.

La idea detrás del área de seguridad demostrable es probar que, bajo ciertas hipótesis, de existir un ataque que quiebre el protocolo de manera eficiente, dicho ataque puede ser usado como parte de un algoritmo eficiente para resolver estas

primitivas. Dado que no se conocen algoritmos eficientes para resolver estas primitivas, esta reducción es un indicador de que, bajo estas hipótesis, es altamente improbable que el protocolo pueda ser quebrado de manera eficiente.

Con el objetivo de garantizar la seguridad de dichos protocolos, el criptógrafo debe demostrar y probar la correctitud de las reducciones. Un enfoque para estructurar las pruebas es la técnica basada en juegos. En tal sentido, las nociones de seguridad se estructuran como programas llamados juegos y la reducción es una secuencia de sucesivas transformaciones entre dichos juegos.
Para la comunidad criptográfica, las pruebas se están volviendo cada vez más complejas y sería deseable contar con una herramienta automatizada que ayude en el proceso de escribir y verificar las mismas. En esta dirección, Halevi [Hal05] explicita las características necesarias para que dicha herramienta tenga éxito en la adopción por parte de la comunidad.
Las herramientas CryptoVerif [Bla06] y CertiCrypt [BGZ09; BGHZ09; Zan10; BGZB10] se han presentado como aproximaciones a los objetivos planteados por Halevi y han demostrado que es factible la construcción de dicha herramienta. Sin embargo no han tenido éxito en su adopción generalizada. Por otro lado, EasyCrypt [BGHZB11] se presenta como la herramienta sucesora de CertiCrypt y una posible candidata a ser adoptada como estándar por la comunidad, en tanto logre cumplir con dichos objetivos.
Con el fin de mostrar la utilidad de la herramienta EasyCrypt, es necesario demostrar y analizar pruebas de seguridad de protocolos ampliamente utilizados haciendo uso de la misma. En este trabajo se muestra la prueba de la cota original del esquema de firma *Full Domain Hash* en EasyCrypt. Además se mencionan las pruebas de otros esquemas.

El informe se organiza de la siguiente manera:

- En el capítulo 2 se presentan las principales nociones matemáticas y criptográficas que dan marco al esquema de prueba.
- En el capítulo 3 se presenta el enfoque de las herramientas automatizadas que son utilizadas por el criptógrafo para desarrollar y verificar sus pruebas. En particular, se describe los principales conceptos de la herramienta EasyCrypt.
- En el capítulo 4 se explica el esquema de firma digital *Full Domain Hash*.
- En el capítulo 5 se muestra el razonamiento y la formalización de la prueba en la herramienta EasyCrypt.
- En el capítulo 6 se discute el uso de la herramienta para la formalización y verificación de la prueba.
- En el capítulo 7 se concluye sobre la importancia de la verificación automatizada de las pruebas de seguridad y el uso de la herramienta EasyCrypt en esta dirección.

2
Nociones Preliminares

2.1. Notación y Definiciones

2.1.1. Función Único-Sentido ("one way function")

Informalmente, una función único-sentido es una función con la propiedad de que se puede computar la imágen de cualquier elemento en un tiempo polinomial en el tamaño de la entrada. Sin embargo es difícil, en el sentido de la teoría de la complejidad, encontrar una preimágen de un elemento elegido de forma aleatoria y uniforme en su rango. No se sabe si existen funciones único-sentido, pero existen candidatos potenciales. Un ejemplo consiste en factorizar el producto de dos primos grandes p y q. El producto $n = pq$ se puede calcular de manera eficiente, pero en el caso general, dado n, no se conocen algoritmos polinomiales para hallar p y q.

2.1.2. Familia de Permutaciones trampa ("trapdoor")

Una familia de permutaciones trampa es una tripleta de algoritmos (KG,f,f^{-1}) donde:

1. Dado un valor del parámetro de seguridad η que indica el tamaño de las claves, el generador $KG(\eta)$ selecciona aleatoriamente un par de claves (pk,sk).
2. Dado el par de claves (pk,sk) y un dominio D, $f_{pk} : D \to D$ es una permutación y $f_{sk}^{-1} : D \to D$ es su inversa.

2.1.3. Esquema de Firma Digital

Un esquema de firma digital consiste en una tripleta de algoritmos ejecutados en tiempo polinomial en el largo de la entrada:

1. Un algoritmo probabilístico generador de claves KG, siendo η el parámetro de seguridad que indica el tamaño de las claves. $KG(\eta)$ produce un par de claves pública y privada: (pk,sk) de tamaño η.
2. Un algoritmo probabilístico de firma $Firmar$. Para firmar un mensaje m bajo la clave secreta sk, se computa x=$Firmar_{sk}(m)$.
3. Un algoritmo determinístico de verificación $Verificar$. Dada la clave pública pk, un mensaje m y una firma candidata x; $Verificar_{pk}(m,x)$ retorna "verdadero" si la firma es válida o "falso" en caso contrario. Se debe cumplir que, para cada par de clave (p_k,s_k) que pueden ser salidas del algoritmo generador de claves y cada mensaje m, $Verificar_{pk}(m,Firmar(sk,m))$ sea verdadera.

2.1.4. Oráculo Aleatorio ("Random Oracle")

Los esquemas de seguridad por lo general utilizan funciones de Hash, originalmente motivadas por la necesidad de firmar mensajes largos con una firma corta. Cuando se analiza un ataque del adversario al esquema se debe pensar en un ataque genérico, independiente de la actual implementación que tengan algunos objetos.
En el modelo de oráculo aleatorio [BR93; FS87] las funciones de hash son representadas como funciones aleatorias uniformemente distribuidas. Esto significa que, dado un mensaje de entrada, el *oráculo de hash* retorna un valor aleatorio y uniforme en su rango de salida. Si la misma consulta se realiza dos o más veces, se retorna el mismo valor siempre.
Más especificamente dado un espacio de mensajes M y un espacio de valores de hash S, se tiene una función $h : M \to S$ tal que para todo $m \in M$ se cumple que $Pr[h(m)] = j = \frac{1}{|S|}$ para todo $j \in S$. Intuitivamente modela un peor caso en el sentido de que distribuciones sesgadas permiten que un adversario tome ventaja de este sesgo.
Por otro lado, el adversario también tiene acceso a un oráculo de firma. Este oráculo es un procedimiento de "caja negra" que permite al adversario firmar los mensajes que consulta sin conocer su implementación. Además, los oráculos son públicos: tanto los adversarios como los algortimos que componen el esquema tienen acceso a ellos.

2.1.5. Ataque Adaptativo de Texto Plano Elegido

Goldwasser, Micali y Rivest [GMR88] formalizaron el concepto de seguridad asintótica de un esquema de firma y luego Bellare y Rogaway [BR93] la extendieron y tomaron en cuenta la utilización de funciones de hash ideales.

Se considera que Alicia, con su clave privada sk_A firma documentos. Por otro lado, Oscar, un adversario realiza un *ataque adaptativo de texto plano elegido* cuando en posesión de la clave pública pk_A de Alicia intenta falsificar firmas. Esto es generar un par (m,x) talque que $Verificar_{pk_A}(m,x)$ es verdadero.

Oscar, a su elección le puede pedir a Alicia que firme un conjunto de mensajes $\{m_1, m_2, ..., m_k\}$. Con esta información desea generar el par (m,x) buscado, donde $m \notin \{m_1, m_2, ..., m_k\}$.

Es adaptativo en el sentido que puede ir modificando sus consultas al ir observando las firmas resultantes de las consultas anteriores. Oscar tiene éxito si logra devolver un par $(mensaje, firma)$ tal que $Verificar_{pk}(mensaje, firma)$ sea verdadero. El *mensaje* no pudo haber sido consultado previamente por su firma (es un mensaje fresco). En otras palabras el atacante logra falsificar una firma (el algoritmo de verificación verifica una firma no realizada por el dueño de la clave privada).

En la siguiente definición se presenta la noción de seguridad frente a un ataque adaptativo de texto plano elegido, en el modelo de Oráculo Aleatorio para cierta función de hash [BR93].

Definición 2.1 (Seguridad de un Ataque Adaptativo de Texto Plano Elegido en el modelo de Oráculo Aleatorio). *Se dice que un esquema de firma es (t,q_H,q_S,ϵ)-seguro contra un ataque adaptativo de texto plano elegido cuando un adversario que conoce la clave pública del firmante, limitado a ejecutar en a lo sumo un tiempo t, haciendo a lo sumo q_H consultas al oráculo de hash y q_S consultas de firma, tiene éxito en falsificar una firma para un mensaje fresco con probabiliad de a lo sumo ϵ.*

2.1.6. RSA: Rivest, Shamir y Adleman

RSA [RSA78] es un sistema criptográfico de clave pública cuya fortaleza de seguridad radica en la dificultad de la factorización de números enteros.
Se define Z_N como el conjunto finito (de exactamente N elementos) formado por las N clases de equivalencia $[0], ..., [N-1]$ en que resulta dividido el conjunto de los enteros Z por la relación de congruencia modulo N. Se define Z_N^* como el subconjunto de Z_N formado por aquellos elementos que son invertibles. Por ejemplo, si $N = 18$, $Z_{18}^* = \{1, 5, 7, 11, 13, 17\}$.

Definición 2.2 (Criptosistema RSA). *Sea $N = pq$ donde p y q son números primos y sean P y C los conjuntos de texto claros y texto cifrados respectivamente, tal que $P = C = Z_N$. Se define: $K = \{(N, p, q, e, d) : ed \equiv 1 mod \phi(N)\}$ donde ϕ es la función de Euler (la cantidad de numeros entre 1 y N que son primos relativos con N).*

Para $K = (n, p, q, e, d)$ se definen las funciones de encriptado y desencriptado como sigue:

$e_K(x) = x^e mod(N)$ *y* $d_K(y) = y^d mod(N)$ *con* $x, y \in Z_N$

La clave pública se compone por los valores N y e, y los valores p, q, d forman la clave privada.

La seguridad del criptosistema RSA está basada en que la función de encriptado $e_K(x)$ es una función único sentido. Para un oponente, no se conoce una manera computacionalmente eficiente de que pueda desencriptar un texto cifrado sin conocer el secreto d. Esto es debido a que no se conocen algoritmos polinomiales para factorizar números muy grandes. El secreto que permite desencriptar un texto cifrado es el conocimiento de la factorización de $N = pq$, dado que se puede computar $\phi(N) = (p-1)(q-1)$ y luego calcular el exponente de desencriptado d utilizando el algoritmo extendido de Euclides. Por esto, se dice que d es la "trampa" de la función e_K.

2.2. Seguridad Demostrable

La noción de *seguridad demostrable* emergió en 1982 con el trabajo de Shafi Goldwasser y Silvio Micali [GM82]. El enfoque consiste en construir los protocolos criptográficos en base a primitivas atómicas que tienen ciertas propiedades de seguridad deseables.
El criptógrafo diseña los protocolos bajo la premisa de que las primitivas atómicas son seguras. Sin embargo, aún cuando estas primitivas tengan propiedades de seguridad buenas el protocolo puede ser inseguro. Un objetivo de su construcción es que pueda mantener inherente la fortaleza de la primitiva y no perderla.
Por ejemplo se demuestra que el encriptado $one-time\ pad$ ($c = m \oplus e$, donde e es elegido uniformemente) admite secrecía perfecta (tiene maxima seguridad). Sin embargo si se usa la misma clave e para computar $c' = m' \oplus e$, se cumple que $c' \oplus c = m' \oplus m$. Esto lo hace muy débil a ataques de texto claro elegido. El "error de protocolo" consiste en usar dos veces la misma clave (de ahí el nombre $one-time\ pad$).
Para probar la seguridad de un protocolo se utiliza una prueba por reducción que muestra que la única manera de vulnerar el protocolo es vulnerar la primitiva atómica que utiliza. Desde este punto de vista, no es necesario criptoanalizar el protocolo directamente, dado que, si se encontrara una debilidad en él, también significaría una debilidad en la primitiva atómica. Por ello, si se realiza criptoanálisis se debe focalizar en la última o en ataques por fuera del modelo de seguridad en cuestión. En la literatura también se puede encontrar esta noción con el nombre de *seguridad reducible*.
A modo de ejemplo, se podría suponer que RSA es una función en un único-sentido y el objetivo de seguridad es vulnerar el protocolo P que se basa en dicha propiedad. Sea A el adversario que se ha diseñado para esto. La reducción desde RSA hacia la seguridad del protocolo, consiste en una transformación que toma el adversario A y lo convierte en un adversario A' que utiliza al primero para vulnerar RSA. Su objetivo es encontrar la preimagen de un elemento elegido de forma aleatoria y uniforme en el rango de la función de RSA. Con este razonamiento, se obtiene que

si se cree que no es posible vulnerar RSA, entonces no existirá tal adversario A y el protocolo P es seguro.

2.3. La Técnica basada en Juegos

La técnica basada en juegos para analizar la seguridad de las construcciones criptográficas fue inicialmente propuesta por Kilian y Rogaway (1996) [KR96] para analizar la seguridad del algoritmo DESX. Sucesivamente fue utilizada por Rogaway junto con otros investigadores para desarrollar sus pruebas y fue posteriormente formalizada en conjunto con Bellare [BR04]. De forma independiente y contemporánea, V. Shoup [Sho04] publicó un trabajo donde explica la técnica desde un punto de vista más conceptual que formal.

2.3.1. Juegos

Un juego es formalizado como una tupla de programas escritos en algún lenguaje con variables locales y globales. El juego es ejecutado con un adversario que es capaz de llamar a los programas que se le proveen y de esta forma dicha interacción captura el comportamiento del adversario en el ataque al criptosistema.

Con este enfoque, el esquema general de una prueba de seguridad consiste en construir una secuencia de juegos: G_0, G_1, ..., G_n. El juego G_0 representa el ataque del adversario al criptosistema y el juego G_n representa el ataque a la primitiva criptográfica objetivo. Cada juego se obtiene aplicando transformaciones al anterior, de forma que las probabilidades de éxito del atacante en cada juego son parecidas. De esta manera, se obtiene que la probabilidad de éxito del atacante en el juego G_0 es cercana a la probabilidad objetivo logrando la prueba de seguridad deseada.

Bellare y Rogaway justifican el paradigma por varias razones. Primeramente creen que el enfoque es más fácilmente verificable y es menos propenso a errores que aquellos basados en un lenguaje probabilístico más convencional. Segundo, es ampliamente aplicable en varios modelos criptográficos, entre ellos el modelo de oráculo aleatorio visto en la sección 2.1.4. Además se puede utilizar en los esquemas simétricos y asimétricos. Tercero, es fácilmente aplicable dado que no es complejo aprenderlo.

Por otro lado, el hecho de que sea más facilmente verificable depende de las transformaciones que se hagan entre los juegos. Es recomendable realizar transformaciones pequeñas que esten justificadas por reglas probadas formalmente y no saltos grandes entre juegos adyacentes.

2.3.2. Transformaciones y Técnicas

A continuación, se presenta los fundamentos de las principales transformaciones y técnicas que se utilizaron en este trabajo para realizar las pruebas objetivo.

Lema Fundamental

Las siguientes definiciones son necesarias para la formulación del lema:

Definición 2.3 (Bandera de Falla). *Una variable booleana "bad" en un juego G es llamada $bandera\ de\ falla$, si es inicializada en $falso$ y eventualmente puede cambiar su valor a $verdadero$ pero nunca puede revertir su valor a $falso$.*

Definición 2.4 (Idéntico Hasta Levantar la Bandera). *Se dice que dos juegos que tienen una bandera de falla cada uno son idénticos hasta levantar la bandera, si son sintácticamente idénticos hasta que a la $bandera$ se le asigna el valor $verdadero$.*

Los juegos G_A y G_B de las figuras 2.1 y 2.2 son ejemplos de juegos idénticos hasta levantar la bandera. En ambos juegos la bandera de falla bad se inicializa $falso$ y son sintácticamente iguales hasta que se le asigna el valor $verdadero$ a la misma. El oráculo aleatorio O retorna para los mensajes m consultados, un valor y elegido en forma aleatoria y uniforme en el rango T. El conjunto L almacena los valores de y para los mensajes m consultados. De esta forma, para idénticas consultas de mensajes, el oráculo O devuelve el mismo valor.

Figura 2.1. Juego G_A

```
Game G_A ≜
  L ← [ ];
  bad ← false
O(m) ≜
  if m ∉ dom(L) then
    y ←$ T;
    if y ∈ ran(L) then
      bad ← true;
    L ← (m, y) :: L
  return L(m)
```

Figura 2.2. Juego G_B

```
Game G_B ≜
  L ← [ ];
  bad ← false
O(m) ≜
  if m ∉ dom(L) then
    y ←$ T;
    if y ∈ ran(L) then
      bad ← true;
      y ←$ T − ran(L);
    L ← (m, y) :: L
  return L(m)
```

El lema fundamental 2.5 es el lema central que justifica la técnica basada en juegos y su prueba se puede encontrar en [BR04]. El lema acota superiormente la

diferencia de probabilidades de éxito que tiene un adversario en cada juego con el valor de la probabilidad de que se asigne *verdadero* a la bandera de falla.

En los juegos G_A y G_B el evento de falla modela la ocurrencia de una colisión en el oráculo aleatorio O. En otras palabras, para la consulta del mensaje m, el oráculo O ya devolvió el valor y para un mensaje m' consultado anteriormente con $m' \neq m$.

Lema 2.5 (Lema Fundamental). *Sean G y H dos juegos idénticos hasta levantar la bandera y sea A un adversario. Los juegos G y H devuelven 1 si el adversario A tuvo éxito en averiguar el desafío; en otras palabras, gana el juego. En caso contrario, los juegos devuelven 0. Se denota $\Pr[G_A \to 1]$ como la probabilidad que devuelva 1 el juego G ejecutando al adversario A. Luego,*

$$\Pr[G_A \to 1] - \Pr[H_A \to 1] \leq \Pr[G_A \textit{levanta-bandera}]$$

Luego de la bandera de falla, nada importa

La proposición 2.6 permite dado dos juegos idénticos hasta levantar la bandera, modificar lo que sucede luego de que la bandera *bad* es asignada a *verdadero*. En otras palabras, se puede decir que cualquier modificación luego de levantar la bandera es conservativa en tanto que no modifica la probabilidad del evento de falla en los juegos.
En el caso del juego G_B, se muestra que luego de levantar la bandera, se vuelve a elegir y de forma que no esté en el rango del conjunto L.

Proposición 2.6 (Luego de la bandera de falla, nada importa). *Sean G y H dos juegos idénticos hasta levantar la bandera y sea A un adversario. Luego,*

$$\Pr[G_A \textit{ levanta-bandera}] = \Pr[H_A \textit{ levanta-bandera}]$$

Elección Perezosa y Elección Inmediata

Se dice que un juego realiza elecciones inmediatas (*eager sampling*), cuando realiza las elecciones aleatorias en la inicialización. Por el contrario, se dice que un juego realiza elecciones perezosas (*lazy sampling*) cuando las elecciones aleatorias se retardan hasta que deben ser utilizadas.
Las juegos de las figuras 2.3 y 2.4 son ejemplos de elecciones perezosas y elecciones inmediatas respectivamente. El oráculo aleatorio O retorna para los mensajes m consultados, un valor y elegido en forma aleatoria y uniforme en el rango T. En el juego G_{lazy} se retardan las elecciones aleatorias de y hasta el momento que los mensajes son consultados. En el juego G_{eager} se selecciona aleatoriamente el valor y' en la inicialización. Luego, cuando se consulta al oráculo aleatorio O para cierto mensaje m' se retorna el valor y' escogido anteriormente.

Figura 2.3. Juego G_{lazy}

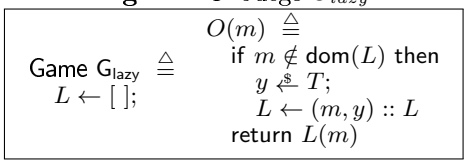
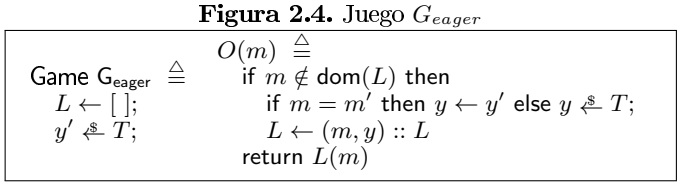


Figura 2.4. Juego G_{eager}



Las pruebas basadas en juegos comunmente incluyen transformaciones entre estos juegos. Estas transformaciones son de movimiento de código inter-procedural, dado que las elecciones aleatorias se mueven desde un oráculo al procedimiento principal del juego (*elecciones inmediatas*) ó contrariamente, desde el procedimiento principal al oráculo (*elecciones perezosas*).

El principio de *Elección Perezosa* asegura que estas transformaciones entre juegos preserva la semántica. En otras palabras, sea A un adversario, se puede afirmar que los juegos G_{lazy} y G_{eager} son equivalentes:

$$\Pr[G_{lazyA} \to 1] = \Pr[G_{eagerA} \to 1]$$

La prueba de esta proposición se puede encontrar en [BR04].

3
Herramientas Automatizadas para la Formalización

Las pruebas criptográficas se han vuelto cada vez más complejas y dificilmente verificables. Es por ello que la comunidad criptográfica ha argumentado en favor de desarrollar herramientas que permitan ayudar en el proceso de escribir y verificar sus pruebas, siendo de esta forma menos susceptibles a errores. En la sección 2.3 se describió cómo la técnica basada en juegos permite expresar las pruebas como secuencias de juegos. Cada juego se puede expresar como un programa. De esta manera se convierten las sucesivas transformaciones entre los juegos en transformaciones entre programas y pueden ser justificados rigurosamente mediante su semántica. Sin embargo, mientras que estas técnicas contribuyen a la rigurosa formalización de las pruebas y son más fáciles de verificar, ellas se basan en teorías de equivalencia de programas que tienen inherente principios de verificación, razonamiento algebraico, complejidad y probabilidad. Es por ello, que se vuelve necesario realizar la verificación formal de las pruebas para comprobar la correctitud de las mismas.

Shai Halevi [Hal05] argumentó que es necesario crear una herramienta que pueda ser utilizada por el criptógrafo para desarrollar y verificar sus pruebas de forma automática. Su principal motivación es que, como comunidad se generan más pruebas de seguridad de las que son rigurosamente verificadas; esto se debe básicamente a que son cada vez más complejas y largas. Las pruebas criptográficas tienen una parte creativa (por ejemplo, describir la simulación de la reducción) y una parte formal (por ejemplo, chequear que la reducción es correcta). Esta última parte comunmente es más dificil de verificar y es allí donde es necesario obtener una ayuda automatizada.

Por otro lado, Bellare y Rogaway [BR06] están de acuerdo con el enfoque propuesto por Halevi y explicitan que una de sus motivaciones para focalizarse en las pruebas basadas en juegos, es justamente la creación de dichas herramientas automatizadas.

3.1. Herramientas

En la literatura se pueden encontrar básicamente dos enfoques de herramientas que utilizan la técnica basada en juegos:

- CryptoVerif [Bla06] fue la primer herramienta en soportar pruebas de seguridad basadas en juegos. Los juegos son modelados como procesos y las transiciones son justificadas por técnicas algebraicas. Dicha herramienta fue aplicada exitosamente para probar protocolos y primitivas, como por ejemplo, FDH [BP06].
- CertiCrypt [BGZ09; BGHZ09; Zan10; BGZB10] es una herramienta que fue presentada como primer paso en dirección a los objetivos propuestos por Halevi y consiste en un framework que permite la construcción y verificación automática de pruebas basadas en juegos.
  CertiCrypt es un framework construido para probar la seguridad de pruebas criptográficas en el asistente de pruebas Coq y su base radica en varias áreas cómo son la probabilidad, complejidad, álgebra y semántica de los lenguajes de programas. La herramienta ofrece técnicas para razonar acerca de la equivalencia de programas probabilísticos incluyendo la lógica relacional de Hoare, transformaciones de programas y varias técnicas de razonamiento basadas en juegos como son los eventos de falla. A su vez, su utilidad está demostrada con la presentación de pruebas de seguridad de varios criptosistemas conocidos, entre ellos del esquema de firma FDH [ZGBO09].

La construcción y aplicación de ambas herramientas a varias pruebas criptográficas conocidas muestran que es factible la realización de los objetivos que proponía Halevi en su trabajo. Sin embargo, dichas herramientas no fueron ampliamente adoptadas por la comunidad criptográfica. Barthe et al. [BGLZB11] atribuyen esto a que es necesario por parte del criptógrafo experiencia y esfuerzo para construir pruebas basadas en la verificación automática debido a la forma en que se presentan tales herramientas. En tal sentido, fallan en parte con los objetivos de Halevi.
Motivados por ello, los autores presentan EasyCrypt cómo sucesor de CertyCrypt y argumentan que es significativamente más fácil de utilizar que las herramientas previamente mencionadas. De esta forma se marca un paso importante para la adopción por parte de los criptografos en la construcción de pruebas de seguridad ayudados por computadoras y la herramienta se alínea aún más con los objetivos propuestos por Halevi.

3.2. EasyCrypt

EasyCrypt [BGHZB11] es una herramienta automatizada para elaborar pruebas de seguridad de sistemas criptográficos a partir de bosquejos de pruebas, es decir, representaciones formales de la esencia de la prueba cómo secuencia de juegos. Su principal objetivo es hacer accesible la verificación de la seguridad a los criptógrafos

requiriendo limitados conocimientos en métodos formales. La herramienta se presenta cómo un posible candidato para ser adoptado cómo estándar en la comunidad e ilustra su aplicación en las pruebas de seguridad de los criptosistemas Cramer-Shoup y Hashed El Gamal [BGHZB11].

A continuación se pretende describir una visión global de las características de la herramienta utilizada en este trabajo para así brindar una noción de los fundamentos en los que se basa la misma. Si el lector desea profundizar aún más se recomienda la lectura de [BGHZB11].
Se insiste en que no es necesario que el criptógrafo deba tener un conocimiento profundo de las nociones que aquí se describen para poder construir y verificar la correctitud de su prueba utilizando la herramienta. Un argumento de ello es el presente trabajo dado que al momento de realizarlo era escasa la documentación que poseía la herramienta y ello no fue impedimento para la construcción de la prueba.

3.2.1. Lenguaje de Programación

Los juegos son modelados como programas en pWhile, un lenguaje fuertemente tipado, probabilístico, procedural e imperativo. Los principales tipos que se usaron en este trabajo incluyen booleanos, enteros, listas y mapas. Además el usuario puede definir operadores y estos deben ser axiomatizados de forma de lograr la semántica deseada o inclusive re-definirlos en función de otros. Los comandos del lenguaje son definidos por la siguiente gramática:

$I ::= V \leftarrow E$	asignación
$\mid V \xleftarrow{\$} DE;$	elección aleatoria
$\mid$ if E then C else C	condicional
$\mid V \leftarrow P(E, ..., E)$	llamada procedural
$C ::= skip$	nop
$\mid I; C$	secuencia

donde V es un conjunto de variables, P es un conjunto de procedimientos y DE es un conjunto de distribuciones de expresiones. La única característica que no es estándar del lenguaje son las asignaciones aleatorias. Para este trabajo, las distribuciones de expresiones son restringidas a distribuciones uniformes en el dominio en cuestión. Esto significa, que se utilizará una elección aleatoria y uniforme en un conjunto de naturales. Si el lector está interesado en una descripción más detallada de como la semántica de pWhile ha sido formalizada en el asistente de pruebas Coq se puede referir a [BGLZB11].
Los programas son representados como colección de procedimientos y los **adversarios** son representandos cómo procedimientos abstractos cuya definición no se especifica. En este trabajo los **oráculos** aleatorios son representados cómo procedimientos que mapean valores desde un dominio a un codominio de valores uniformes. Se debe ser consistente al implementarlos para que a iguales consultas sea respondido el mismo valor.

3.2.2. Logica Probabilística y Relacional de Hoare

El fundamento de la lógica de EasyCrypt se basa en la lógica probabilística y relacional de Hoare (pRHL: *probabilistic Relational Hoare Logic*) cuyos juicios son cuadruplas de la forma:

$$\vdash c_1 \sim c_2 : \psi \rightarrow \phi$$

donde c_1, c_2 son programas y ψ, ϕ son formulas relacionales de primer orden. Las formulas relacionales son definidas por la gramática:

$$\psi, \phi ::= e \mid \neg\phi \mid \psi \wedge \phi \mid \psi \vee \phi \mid \psi \rightarrow \phi \mid \forall x.\phi \mid \exists x.\phi$$

donde e puede ser una expresión booleana sobre variables logicas y variables de programas etiquetados con $\langle 1 \rangle$ ó $\langle 2 \rangle$ dependiendo a que juego pertenezcan. La única restricción es que las variables lógicas no deben ocurrir libres. Una fórmula relacional se interpreta cómo una relación entre las memorias de los programas.
Un juicio pRHL de la forma $\vdash G_1 \sim G_2 : \psi \rightarrow \phi$ relaciona la evaluación de un programa G_1 con la evaluación de un programa G_2 en relación con una precondición ψ y una poscondición ϕ, ambas definidas cómo relaciones de estados determinísticos. En el caso de los programas determinísticos, este juicio es válido cuando para cualquier par de memorias m_1, m_2 que satisfacen $m_1\ \psi\ m_2$, si las evaluaciones de G_1 en m_1 y G_2 en m_2 finalizan con memorias finales m_1' y m_2' respectivamente, luego $m_1'\ \phi\ m_2'$ vale. El lector puede referirse a [BGZ09; JYL01] por una definición más precisa.

En EasyCrypt, un juicio pRHL es introducido con juicios de la forma:

$$\text{equiv F} : G_1.f_1 \sim G_2.f_2 : Pre \rightarrow Pos$$

donde F es el identificador de un juicio, G_1 y G_2 son juegos, f_1 y f_2 son identificadores de procedimientos en los juegos G_1 y G_2 respectivamente.
La precondición Pre y poscondición Pos son formulas relacionales. La precondición es una relación entre los parámetros y variables globales de los procedimientos. La poscondición es una relación entre las variables globales y una variable especial llamada res. Esta variable representa el valor de retorno del procedimiento. Más precisamente $res\langle 1 \rangle$ es el valor de retorno del procedimiento izquierdo $(G_1.f_1)$ y $res\langle 2 \rangle$ es el valor de retorno del procedimiento derecho $(G_2.f_2)$.
Por conveniencia, EasyCrypt permite incluir en las precondiciones y poscondiciones subformulas de la forma: $x_1, ..., x_n$ donde x_i son variables de los procedimientos. Esto significa que los valores de las variables $x_1, ..., x_n$ deben coincidir en las memorias de los juegos izquierdo y derecho. Esta escritura es equivalente y más corta a escribir $x_1\langle 1 \rangle = x_1\langle 2 \rangle, ..., x_n\langle 1 \rangle = x_n\langle 2 \rangle$

3.2.3. Razonamiento Automatizado

La herramienta combina la aplicación de la precondición más débil con tácticas interactivas, con el objetivo de generar condiciones de verificación cuya validez es chequeada utilizando probadores automáticos y cómo último recurso se pueden utilizar asistentes de prueba. Los probadores automáticos utilizan reglas de inferencia lógica para demostrar la veracidad de los juicios. En la prueba presentada se utilizaron los probadores $alt-ergo$ [Bob] y $simplify$ [DNS05].

Tácticas

La herramienta implementa un lenguaje básico de tácticas para probar la validez de los juicios y las transformaciones de los programas. Las tácticas pueden ser aplicadas en ambos lados de los juegos ó en un lado solo, operando de esta forma en un solo juego del juicio. A continuación se menciona la semántica de las tácticas que se utilizan en este trabajo:

- $derandomize$: Dado un programa c se computa un conjunto de sentencias aleatorias s y un programa determinístico c' talque c es semánticamente equivalente a $(s; c')$.
- rnd: Dada una elección aleatoria sobre la misma variable, sustituye en la postcondición cada ocurrencia de ella por una cuantificación universal.
- $rnd\ x\ y$: Dada una elección aleatoria sobre la misma variable a, sustituye en la postcondición cada ocurrencia de ella por una cuantificación universal pero sustituyendo la ocurrencia de a por x y por y según se trate del primer o segundo juego respectivamente.
  La táctica anterior es un caso particular de esta: $rnd\ a\ a$.
- $pop, push, swap$: Permiten mover sentencias en el juego siempre que sea posible.
- $inline$: Sustituye la llamada de un procedimiento por su definición.
- $case\ inv$: La táctica realiza un análisis de casos sobre el valor de una expresión booleana (predicado) inv en las memorias iniciales. Obliga a realizar primeramente la prueba donde se agrega inv a la precondición y luego la prueba donde se agrega el negado $\neg inv$.
- $app\ x\ y\ inv$: Sean ψ y ϕ las pre y post condiciones respectivamente. Sea el juego $G_1 = A_1 X B_1$ donde X es la línea de código número x y A_1 y B_1 son el código que esta antes y después del corte. Análogamente se razona con el juego $G_2 = A_2 Y B_2$ para el corte en la línea número y. Entonces, la táctica obliga a realizar dos pruebas. La primera con pre-condición ψ y postcondición el invariante inv y el prefijo de ambos juegos: $A_1 X$ y $A_2 X$. Luego la siguiente prueba se debe realizar con precondición inv y postcondición ϕ y el sufijo de ambos juegos: B_1 y B_2. La utilidad de la táctica radica en que el criptógrafo intuye que sabe probar el juicio "pasando" por el invariante "intermedio" inv y "cortando" los juegos para ello.
- $trivial, auto$

Razonando con eventos de falla

Las pruebas criptográficas usualmente hacen uso de que dos juegos G_1 y G_2 se comportan de forma idéntica hasta levantar la bandera y se utiliza una bandera booleana *bad* para marcar el evento de falla F de la simulación. La transición entre estos dos juegos está justificadas por el lema fundamental presentado en la sección 2.3.2. El lema permite acotar la diferencia entre las probabilidades de ocurrencia de cualquier evento en G_1 y G_2 por la probabilidad de ocurrencia del evento de falla F en cualquiera de los dos juegos. Generalmente, en las pruebas, el evento de falla ocurre en el oráculo de firma de las consultas que realiza el adversario y es utilizado para acotar la probabilidad de ocurrencia del evento de falla de la simulación.

Selección Perezosa

Otras técnicas comúnmente utilizadas en las pruebas son las de selección inmediata y perezosa como se definieron en la sección 2.3.2. Estas técnicas son implementadas en EasyCrypt y para su definición el lector puede referirse a [BR06; BGZ09].

3.2.4. Extracción de pruebas en Coq

EasyCrypt genera a partir de la prueba un archivo Coq [The10] que puede ser verificado utilizando su chequeo de tipos, de esta forma a partir del bosquejo de prueba se produce una prueba independiente verificable. Además, si algún paso de la prueba no es posible realizarlo con los métodos automatizados, es posible, a partir del archivo de salida, completar la prueba con el asistente.

4
Full Domain Hash

Full Domain Hash (de aquí en más FDH) es un esquema de firma digital basado en la familia de permutaciones trampa RSA 2.1.2 propuesto por Bellare y Rogaway [BR93; BR96]. Este esquema sigue el paradigma $hash - then - decrypt$ que es ampliamente utilizado para firmar documentos y consiste básicamente en primero computar el valor de hash del mensaje m, $y = Hash(m)$ y luego computar su firma como $x = f_{sk}^{-1}(y)$. Para verificar que una firma x es válida para el mensaje m, primeramente se computa $f_{pk}(x)$ y se chequea si es igual a $Hash(m)$. Este paradigma es la base de varios estándares de firma [JK03]. Un requerimiento necesario es que la función de Hash sea resistente a las colisiones pero no es suficiente para obtener una prueba de seguridad. Se debe tener en cuenta que aún cuando se asuma que la función de RSA sea una función de único-sentido y la función de Hash sea ideal, el esquema de firma puede ser inseguro en el sentido que no sea resistente a ciertos ataques.

4.1. FDH-RSA

El esquema de firma *FDH-RSA* está basado en el algoritmo RSA presentado en la sección 2.1.6. En dicho esquema para firmar un mensaje m primeramente se computa el valor de hash utilizando la función de hash cuyo rango es el dominio completo de la función trampa de único-sentido RSA: $H : \{0,1\}^* \rightarrow Z_N^*$ y luego se computa la firma cómo $\sigma = H(m)^d mod N$. El algoritmo de verificación simplemente verifica si la siguiente igualdad es cierta, $H(m) = \sigma^e mod N$. El esquema de firma se muestra en la figura 4.1.

FDH-RSA posee seguridad demostrable [BR93; BR96], es decir, que se muestra una prueba donde se reduce el problema de quebrar la seguridad del esquema FDH al problema de invertir la función de RSA en el sentido que fue presentado en las secciones 2.1.5 y 2.2.

Figura 4.1. Esquema de firma FDH-RSA

$SignFDH_{N,d}(m) \triangleq$
$y \leftarrow H(m)$
return $y^d \bmod N$

$VerifyFDH_{N,e}(m,x) \triangleq$
$y \leftarrow x^e \bmod N;$
$y' \leftarrow H(m);$
return $y = y'$

4.2. Generalización de FDH

FDH ha sido históricamente presentado basado en la familia de permutaciones trampa RSA. Sin embargo la misma construcción y su prueba de seguridad son válidas para cualquier familia de pemutaciones trampa. En este trabajo, se considera una familia genérica de permutaciones trampa de modo que se pueda abstraer de dicha instanciación y probar los resultados de seguridad independientemente de ella. De esta forma, el criptógrafo podrá instanciar dicha permutación con la permutación trampa que desee y lograr la cota de seguridad probada en este trabajo.

4.2.1. Definición

En la definición 4.1 se define el esquema de seguridad FDH como se describe a continuación. El algoritmo de generación de claves solo ejecuta el algoritmo de generación de claves de la permutación trampa f subyacente obteniendo la clave pública pk y la clave secreta sk, que son utilizadas en el algoritmo de verificación y de firma respectivamente. Dado un mensaje $m \in \{0,1\}^*$, la firma se computa simplemente cómo la preimagen bajo f de su valor de hash: $f^{-1}(sk, H(m))$. Por otro lado, si se tiene un mensaje m y su firma σ, se verifica si el valor de hash de m coincide con la imágen de σ bajo f: $f(pk, \sigma) = H(m)$.

Definición 4.1 (Full Domain Hash). *Sea (KG, f, f^{-1}) una familia de permutaciones trampa único-sentido en un grupo cíclico G_n y sea H una familia de funciones de hash con dominio bitstrings de largo arbitrario y con codominio las permutaciones en G_n. El esquema de firma digital FDH se compone de la siguiente tripla de algoritmos:*

$$KG(n) = (pk, sk) \leftarrow KG_f(n); return(pk, sk)$$

$$Sign(sk, m) = return(f^{-1}(sk, H(m)))$$

$$Verify(pk, m, \sigma) = return(f(pk, \sigma) = H(m))$$

4.2.2. FDH cumple con la seguridad demostrable

El esquema FDH puede ser probado seguro en el modelo de oráculo aleatorio contra un ataque existencial de falsificación bajo el modo adaptativo de texto plano elegido ofreciendo una reducción según lo presentado en la sección 2.1.5.

Dicha prueba de seguridad [BR93; BR96] se puede enunciar de la siguiente forma.

Teorema 4.2 (Seguridad Demostrable de FDH). *Sea A un adversario efectuando un ataque adaptativo de texto plano elegido contra el esquema de FDH que hace a lo sumo q_H consultas al oráculo de hash H y q_S consultas al oráculo de firma $Sign$. Se asume, sin pérdida de generalidad, que el adversario A nunca repite las consultas al oráculo y que si hace una consulta de firma del mensaje m, previamente consultó su valor de hash. Además, se supone que el adversario A tiene éxito con probabilidad ϵ. Entonces, se puede construir un inversor I para la familia subyacente de permutaciones trampa y probar que tiene éxito con probabilidad ϵ' no despreciable:*

$$\epsilon'(k) = \Pr[(f, f^{-1}) \leftarrow G_*(1^k); y \xleftarrow{\$} \{0,1\}^k : I(f, y) = f^{-1}(y)]$$

donde:

- *k es el parámetro de seguridad y representa el largo de los mensajes en el dominio de la permutación f.*
- *G_* es un generador de permutaciones trampa.*

Es decir que dada la permutación trampa f y un elemento y elegido de forma aleatoria y uniforme en el codominio de f, el desafío del inversor I es retornar una preimágen x de y bajo la permutación f. El inversor I tiene éxito, cuando x coincide con una preimágen de y por f. En [BR93] se demuestra que dicha probabilidad $\epsilon'(k)$ de éxito está acotada inferiormente por un factor de q^{-1} siendo $q = q_H + q_S$. Dado que q es polinomial en k, dicha probabilidad es no despreciable lo que contradice el hecho de que G_* es un generador de las permutaciones trampa.
Se puede decir que esta seguridad asintótica es deseable pero su aplicación en la práctica es limitada, dado que no establece cómo elegir los parámetros de seguridad para alcanzar cierto nivel de seguridad para un largo *fijo* n del mensaje. Para ello, un resultado más útil es calcular la seguridad exacta del esquema, es decir, una cota que cuantifique la brecha entre la seguridad del esquema y la dificultad de invertir la permutación único-sentido para un largo del mensaje n fijo.

4.2.3. Seguridad Exacta de FDH

En la sección 4.2.2 se enunció el teorema presentado en 4.2 que reduce el problema de invertir la permutación trampa al de quebrar la seguridad del esquema FDH. Sin embargo, cuando se implementa el esquema se deben elegir adecuadamente los parámetros para obtener una cota concreta basada en los

recursos y la probabilidad de éxito del ataque. Para ello es necesario cuantificar la correspondencia entre el problema de invertir la subyacente permutación trampa y el problema de encontrar la firma de un mensaje fresco según el esquema.
El siguiente teorema muestra el cálculo de la seguridad exacta de FDH que se obtiene directamente de observar la reducción descripta en la sub-sección anterior [BR93].

Teorema 4.3 (Seguridad Exacta de FDH). *Si la permutación trampa subyacente es (t', ϵ')-segura, luego FDH es (t,q_H,q_S,ϵ)-seguro contra un ataque adaptativo de texto plano elegido donde:*

$$t = t' - (q_H + q_S)T_f$$

$$\epsilon = q\epsilon' + 2^{-k}$$

donde T_f es una cota superior del tiempo que se necesita para computar la imágen de un elemento bajo f.

4.2.4. Cota Original

En la sección 5 se presenta una prueba basada en juegos del resultado original presentado por Bellare y Rogaway [BR93] pero aplicado al esquema FDH-generalizado presentado en la sección 4.2.1. El teorema se formaliza de la siguiente manera:

Teorema 4.4 (Cota Original). *Sea A un adversario montando un ataque existencial y adaptativo de texto plano elegido contra el esquema de firma FDH que hace a lo sumo q_H consultas al oráculo de hash y a lo sumo q_S consultas al oráculo de firma. Se supone que A tiene éxito en falsificar una firma para un mensaje fresco con probabilidad ϵ y en un tiempo t. Entonces, existe un inversor I que halla una preimagen de un elemento elegido de forma aleatoria y uniforme en el rango de f con probabilidad ϵ' dentro de un tiempo t' tal que:*

$$\epsilon' \geq (q_H + q_S + 1)^{-1}\epsilon$$

$$t' \leq t + (q_H + q_S)O(T_f)$$

donde T_f es una cota superior del tiempo requerido para computar la imágen de un elemento del grupo bajo la permutación f.

Se observa que la cota de seguridad depende del número de consultas tanto al oráculo de hash como al oráculo de firma. En la sección 5.1 se muestra la prueba de dicho teorema usando EasyCrypt.

5
Verificación Formal de FDH en EasyCrypt

En esta sección se presenta el razonamiento y la formalización de la prueba de la cota original de FDH enunciada en el teórema 4.4 en EasyCrypt.

5.1. Idea Principal

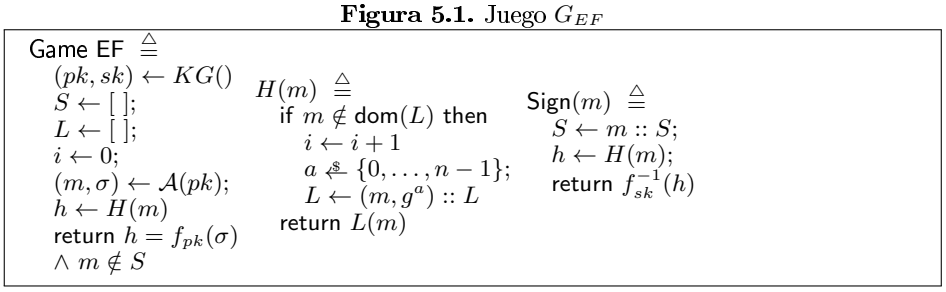
Se considera un adversario A que realiza un ataque *existencial* y *adaptativo de texto plano elegido* contra el esquema de firma FDH pudiendo realizar a lo sumo q_H y q_S consultas a los oráculos de hash y de firma respectivamente. El adversario A tiene éxito si logra hallar un par (m, σ) tal que el algoritmo de verificación del esquema valida que $H(m) = f_{pk}(\sigma)$. A su vez, el mensaje m debe ser fresco, es decir, que el adversario nunca consultó al oráculo de firma con el mensaje m.
El adversario A es una caja negra, en el sentido que no se conoce como implementa el ataque, simplemente tiene acceso al oráculo de hash que es implementado como un oráculo aleatorio y al oráculo de firma para poder realizar las consultas.

El juego de la figura 5.1 captura la semántica antes descripta. Al principio del juego se ejecuta el algoritmo de generación de claves KG_f y se crean dos listas, L y S, que almacenan el valor de hash y la firma de los mensajes consultados respectivamente. Con el objetivo de ser consistente al simular el comportamiento aleatorio del oráculo de hash, a idénticas consultas de mensajes se devuelve el mismo valor de hash.

Entonces, la probabilidad de éxito del adversario se puede expresar como la probabilidad de que ocurra la conjunción entre los siguientes puntos:

- El algoritmo de verificación de FDH devuelve verdadero para la firma σ y el mensaje m hallado por el adversario. Esto quiere decir, que se cumple que $f_{pk}(\sigma) = H(m)$
- El adversario nunca consultó al oráculo de firma con el mensaje m.

Figura 5.1. Juego G_{EF}



Esto se expresa de la siguiente manera:

$$\epsilon = \Pr[G_{EF} : h = f_{pk}(\sigma) \wedge m \notin S]$$

El objetivo de la construcción de la prueba es mostrar que se reduce la seguridad del esquema a la dificultad de invertir la permutación único-sentido f subyacente. Esto significa, que si existe el adversario A con la probabilidad de éxito antes mencionada, entonces es posible construir un inversor I que halla la preimágen de un elemento y elegido de forma aleatoria y uniforme en el rango de f con probabilidad ϵ'. Sea x la respuesta de I. Como f es una permutación trampa con clave secreta sk, el adversario tiene éxtio si el valor de $f_{sk}^{-1}(y)$ es igual a la preimágen x hallada. Sea G_{OW} el juego que captura la semántica del ataque del inversor I, entonces esta probabilidad se puede expresar como:

$$\epsilon' = \Pr[G_{OW} : x = f_{sk}^{-1}(y)]$$

Con el objetivo de poder construir el juego G_{OW} se deben tener en cuenta los siguientes puntos:

1. Al tratarse de una prueba por reducción, el inversor I hace uso del adversario A, del cual se conoce su probabilidad de éxito, para llevar a cabo su ataque. Para ello, el inversor I intercepta las consultas que hace A a los oráculos de hash y firma para implementarlos a su conveniencia. El inversor I debe tener en cuenta las siguientes restricciones a la hora de modelar dichos oráculos. Primeramente la implementación del oráculo de hash debe ser aleatorio dado que es sobre ese modelo que el adversario A sabe efectuar su ataque. Luego, es evidente, que el inversor I no conoce la clave secreta sk para firmar, por lo que debe simular el oráculo de firma de forma que sea consistente con el oráculo de hash y el algoritmo de verificación del esquema.

2. El éxito del inversor I depende del éxito del adversario A. El adversario A tiene éxito en el juego G_{EF} si se cumple que $Hash(m) = f_{pk}(\sigma)$. En otras palabras, la igualdad anterior sostiene que σ es la preimágen de $Hash(m)$ por f. Entonces,

si el inversor I logra que el valor de $Hash(m)$ conincida con su desafío y, la preimágen buscada es en definitiva la firma σ devuelta por el adversario A.

Con lo antes descripto en mente, bastaría con que el oráculo de hash devuelva para el mensaje m el desafío y para la consulta de verificación del juego. El adversario A puede realizar a lo sumo $q_H + q_S$ consultas al oráculo de hash, dado que cada consulta al oráculo de firma implica una consulta al oráculo de hash y se debe sumarle una consulta adicional por la verificación del juego. Para implementar esto, el oráculo de hash tiene un contador i de consultas. Entonces, el inversor I debe adivinar qué número de consulta j al oráculo de hash es de verificación, es decir, en qué número de consulta se pregunta por el hash del mensaje cuya firma el adversario falsifica. Esto lo logra con probabilidad $(q_H + q_S + 1)^{-1}$.

El juego de la figura 5.2 captura la semántica antes descripta. Sea $q = q_H + q_S$. Los conjuntos L y P, almacenan el valor de hash y firma para los mensajes m consultados respectivamente.

Figura 5.2. Juego G_{OW}

```
Game G_OW ≜
  (pk, sk) ← KG()
  a' ←$ {0, ..., n − 1};
  x ← Inv(pk);
  return x = f_sk^{-1}(g^{a'})

Inv(pk) ≜
  L ← [ ];
  P ← [ ];
  i ← 0;
  j ←$ {0, ..., q};
  (m, σ) ← A(pk);
  return σ

H(m) ≜
  if m ∉ dom(L) then
    if i = j then
      h ← g^{a'}
    else
      a ←$ [0..n − 1];
      h ← f_pk(g^a)
      P ← (m, g^a) :: P;
    i ← i + 1
    L ← (m, h) :: L
  return L(m)

Sign(m) ≜
  S ← m :: S;
  h ← H(m);
  return P[m]
```

En suma, el éxito del inversor I depende tanto del éxito del adversario A como de que la simulación sea exitosa, es decir, al menos I tendrá éxito cuando ocurran estos dos hechos que son independientes. Entonces, de forma natural se desprende el teorema 4.4 de la cota original a probar:

$$\Pr[G_{OW} : x = f_{sk}^{-1}(y)] \geq (q_H + q_S + 1)^{-1} \Pr[G_{EF} : h = f_{pk}(\sigma) \wedge m \notin S]$$

5.2. La Prueba

En esta sección se describe la secuencia de juegos que define la prueba de seguridad y las transformaciones semánticas que se realizan entre los mismos. Para

cada transformación se justifican los cambios realizados en los juegos y se mencionan algunas de las tácticas más destacadas que se utilizaron en EasyCrypt. Además, dado que cada prueba requiere de varios invariantes a probar, que solo toman sentido cuando se la construye observando detalladamente las precondiciones y poscondiciones, se hará mención solo a aquellos que resulten ilustrativos para el resultado que se quiera probar. Es por ello que si el lector desea profundizar observando la totalidad y composición de las distintas tácticas utilizadas así como de los invariantes involucrados, puede referirse a la sección 8 que contiene el archivo completo de la prueba.

5.2.1. Declaraciones

Tipos

En la prueba se definen los tipos abstractos: grupo, mensaje y el par de claves pública y privada. Se denomina g al elemento generador del grupo de orden n. Sea q el número máximo de consultas al oráculo de hash, ie, $q = q_H + q_S$.

Operadores

Se definen los operadores de exponenciación y logaritmo discreto en base g según:

$$(\wedge) : group \times int \to group$$

$$dlg : group \to int$$

Además, se define el siguiente operador para retornar las claves:

$$pkey : key\_pair \to public\_key$$

$$skey : key\_pair \to secret\_key$$

Se define la función de permutación trampa único-sentido según:

$$f : public\_key \times group \to group$$

$$f^{-1} : secret\_key \times group \to group$$

Axiomas

En este punto los operadores y constantes son totalmente abstractos, aún no se ha especificado su semántica. Para ello, se definen los siguientes axiomas:

- *q es positivo:*
  q_nat: $q \geq 0$

- *la permutación f es consistente con el par de claves:*
  f_inv_left: $\forall(x : group, kp : key\_pair).\{x = f^{-1}(skey(kp), f(pkey(kp), x))\}$
  f_inv_right: $\forall(x : group, kp : key\_pair).\{x = f(pkey(kp), f^{-1}(skey(kp), x))\}$

- *exponenciación y logaritmo:*
  dlg_pow: $\forall(x : group).\{g^{dlg(x)} = x\}$
  pow_dlf: $\forall(a : int).\{dlg(g^a) = a \bmod n$
  dlg_bound: $\forall(x : group)\{0 \leq dlg(x)\} \wedge \{dlg(x) \leq n - 1\}$
  mod_small: $\forall(a : int, n : int).\{0 \leq a\} \Rightarrow \{a < n\} \Rightarrow \{a\ mod\ n\}$

Adversarios

El adversario A recibe una clave pública y retorna el par $(mensaje, firma)$ luego de efectuar su ataque. Su especificación es:

adversary A(pk: public_key) : message * group
{ message $\rightarrow$ $group$; $message \rightarrow group$}

Entre llaves se especifica el tipo de oráculos a los que puede tener acceso el adversario, en este caso, al oráculo de hash y firma respectivamente. Se hace notar que el código del adversario es desconocido.

El procedimiento KG que no tiene entrada y retorna una variable de tipo key_pair, representa el algoritmo de generación de claves del esquema. Si bien se declara como un adversario y evidentemente no lo es, se representa de esta forma pues en el momento de realizar la prueba era el mecanismo que proporcionaba EasyCrypt para abstraerse de la implementación del procedimiento. Hoy en día, la herramienta proporciona un operador probabilístico para ello.

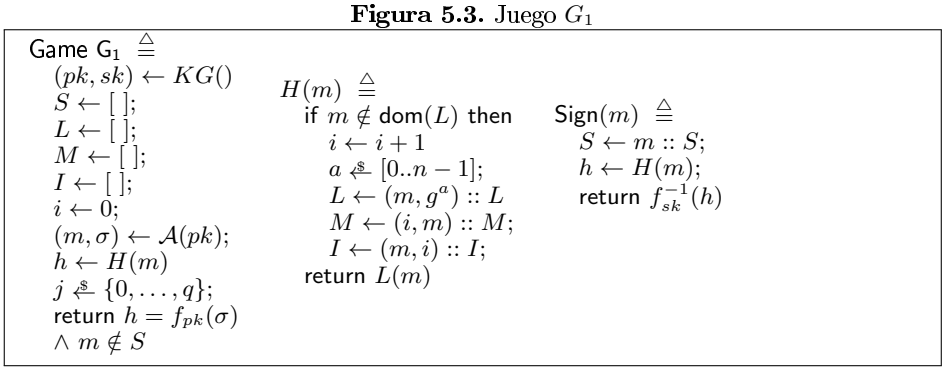
adversary: KG() : key_pair { }

5.2.2. Secuencia de Juegos

La prueba consiste en una secuencia de juegos comenzando con el juego G_{EF} que representa el ataque existencial y adaptativo de texto plano elegido y terminando con el juego G_{OW} que representa el ataque del inversor a la permutación trampa único-sentido subyacente. En cada transformación entre juegos se busca que la cantidad de cambios no repercuta en la dificultad de la justificación formal de las reglas.

En el juego G_1 de la figura 5.3 se introduce la elección de j que será utilizada más adelante por el inversor I. La elección se realiza de forma aleatoria y uniforme entre todas las posibles consultas al oráculo de hash. La sentencia se introduce luego de la llamada al adversario A, de forma que su independencia con la elección de j es evidente.

Figura 5.3. Juego G_1



Además, se mantienen los conjuntos M, I, que almacenan los mensajes que fueron consultados a H junto con su índice.

Entonces, se prueba que:

$$\Pr[G_{EF} : res \wedge i \leq q] = \Pr[G_1 : res \wedge i \leq q]$$

Para probar dicha igualdad de probabilidad es necesario probar la equivalencia con respecto a las variables i y res y de las variables involucradas en el resultado. Esto se expresa cómo:

$$G_{EF} \sim G_1 : = \{pk, sk, L, S, i, res\}$$

Se utilizan las siguiente tácticas para probar dicha equivalencia. Primeramente se sustituye H por su definición ($inline$) y la elección aleatoria de j se sube lo más posible en el juego sustituyendola por una asignación determinística ($derandomize$). Luego, se calcula la precondición más debil (wp), la cual sube la elección aleatoria de a, quedando para resolver la llamada al adversario para el cuál se prueba el invariante en las variables. La táctica rnd remueve la asignación aleatoria que se originó a partir de a, agregando en todas sus ocurrencias un cuantificador universal en la poscondición. De forma similar se resuelve la asignación aleatoria originada a partir de j en el juego G_2. Finalmente, se prueba la equivalencia con la táctica $trivial$.

Trabajando sobre el juego G_1 se demuestra la probabilidad de que la elección de j coincida con la consulta de verificación al oráculo de hash. Se prueba el siguiente invariante sobre el juego G_1 en conjunto con invariantes sobre las variables globales y la semántica entre los conjuntos M y I.

$$G_1 \sim G_1 : \{res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]\}_{\langle 1 \rangle} \iff$$
$$\{res \wedge i \leq q \wedge j = I[m] \wedge 0 \leq I[m] \geq q\}_{\langle 2 \rangle}$$

haciendo posible demostrar que:

$$\Pr[G_1 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]] =$$
$$\Pr[G_1 : res \wedge i \leq q \wedge j = I[m] \wedge 0 \leq I[m] \geq q]$$

El hecho de que j sea el índice de la consulta de verificación es igual al hecho de que $\{j = I[m] \wedge 0 \leq I[m] \geq q\}$, por lo que dicha probabilidad es de $(q_H + q_S + 1)^{-1}$ y es evidentemente independiente del éxito del ataque. Sea $q = q_H + q_S + 1$. Entonces, se cumple que:

$$\Pr[G_1 : res \wedge i] * q^{-1} \leq \Pr[G_1 : res \wedge i \leq q \wedge j = I[m] \wedge 0 \leq I[m] \geq q]$$

Este resultado es admitido en el extracto de la prueba, dado que *EasyCrypt* aún no tenía implementado el computo de probabilidades cuando existe independencia de eventos. Esto significa que el juicio se asume verdadero sin realizar una prueba, pero como se explicó anteriormente es correcto realizar dicha asunción.

Las transformaciones en los siguientes juegos G_2 y G_3 tendrán por objetivo poder aplicar la táctica de *eager sampling* (elección inmediata), para así poder introducir el desafío del inversor I en la j-ésima llamada al oráculo de hash.
En el juego G_2 de la figura 5.4 se mueve la asignación aleatoria de j para un lugar anterior a la llamada del adversario A. Además, luego de la llamada del adversario se realiza la re-elección del exponente a': si $i \leq j$, significa que no se realizó la j-ésima consulta al oráculo de hash y entonces es posible seleccionar nuevamente de forma aleatoria y uniforme el exponente; de lo contrario, a' se recupera computando el logaritmo discreto del valor de hash correspondiente a j, es decir, $L[M[j]]$.

Figura 5.4. Juego G_2

Game G_2 $\triangleq$
$(pk, sk) \leftarrow KG()$
$S \leftarrow [\,]$;
$L \leftarrow [\,]$;
$M \leftarrow [\,]$;
$i \leftarrow 0$;
$j \xleftarrow{\$} \{0, \ldots, q\}$;
$(m, \sigma) \leftarrow \mathcal{A}(pk)$;
$h \leftarrow H(m)$
if $i \leq j$then
  $a' \xleftarrow{\$} \{0, \ldots, n-1\}$;
else
  $a' \leftarrow dlg(L[M[j]])$
return $h = f_{pk}(\sigma)$
$\wedge\ m \notin S$

$H(m)$ $\triangleq$
  if $m \notin$ dom$(L) \wedge$
  $(i \leq j \ ||$
  $M[j] \in$ dom$(L))$ then
    $i \leftarrow i + 1$
    $a \xleftarrow{\$} [0..n-1]$;
    $L \leftarrow (m, g^a) :: L$
    $M \leftarrow (i, m) :: M$;
  return $L(m)$

Sign(m) $\triangleq$
  $S \leftarrow m :: S$;
  $h \leftarrow H(m)$;
  return $f_{sk}^{-1}(h)$

Se prueba la siguiente equivalencia entre los juegos:

$$G_1 \sim G_2 : = \{res, pk, sk, L, S, M, i, j, m\} \wedge \{j < i\} \Rightarrow \{j \in dom(M)\}_{\langle 2 \rangle}$$

y a partir de ella se deduce que:

$$\Pr[G_1 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]] =$$
$$\Pr[G_2 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]]$$

El juego G_2 representa el juego *perezoso* ("lazy") dado que se retarda la elección aleatoria del exponente a hasta que es utilizada en el oráculo de hash. Por el contrario, en el juego G_3 de la figura 5.5 se elige, antes de la llamada al adversario, el exponente desafío a' que se desea devolver cómo valor de hash en la j-ésima consulta. De este modo, G_3 es el juego *inmediato* ("eager").

Figura 5.5. Juego G_3

```
Game G3 ≜
  (pk, sk) ← KG()
  S ← [ ];
  L ← [ ];
  M ← [ ];
  i ← 0;
  j ←$ {0, ..., q};
  if i ≤ j then
    a' ←$ {0, ..., n − 1};
  else
    a' ← dlg(L[M[j]])
  (m, σ) ← A(pk);
  h ← H(m)
  return h = f_pk(σ)
  ∧ m ∉ S

H(m) ≜
  if m ∉ dom(L) ∧
  (i ≤ j ||
  M[j] ∈ dom(L)) then
    if i = j then
      h ← g^a'
    else
      a ←$ [0..n − 1];
      h ← g^a
    i ← i + 1
    L ← (m, h) :: L
    M ← (i, m) :: M;
  return L(m)

Sign(m) ≜
  S ← m :: S;
  h ← H(m);
  return f_sk^-1(h)
```

Entre los juegos G_2 y G_3 se aplica la transformación de *eager sampling* (elección inmediata) que preserva la semántica. Aplicando dicha táctica se prueba la siguiente equivalencia:

$$G_2 \sim G_3 : = \{res, pk, sk, L, S, M, i, j\}$$

y luego es posible probar que:

$$\Pr[G_2 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]] =$$
$$\Pr[G_3 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]]$$

La simulación fallará cuando el adversario A consulte por la firma de un mensaje que corresponde con la j-ésima consulta al oráculo de hash, esto es, $i = j$. En este

caso, el algoritmo de verificación del esquema para los valores de hash y de firma devueltos responderá que no es un par válido. Entonces, en el juego G_4 de la figura 5.6 se introduce la bandera *bad* que tomará el valor *true* cuando en el oráculo de firma suceda esto.
Por otra parte, en el oráculo de hash, cuando $i \neq j$, en lugar de devolver cómo valor de hash g^a se devolverá $f_{pk}(g^a)$. De esta manera, en un juego sucesivo se podrá devolver g^a como la firma σ de un mensaje m y la simulación resultará exitosa dado que se verificará que: $f_{pk}(g^a) = H(m) = f_{pk}(\sigma) = f_{pk}(g^a)$. Es posible hacer este cambio en el oráculo de hash dado que al ser f una permutación, la imágen de un elemento elegido de forma aleatoria y uniforme en el dominio de f tendrá también distribución uniforme en el rango de f. Se introduce el conjunto P para almacenar dichos valores y luego devolverlos en el oráculo de firma.

Figura 5.6. Juego G_4

```
Game G4 ≜
  (pk, sk) ← KG()
  S ← [ ];
  L ← [ ];
  M ← [ ];
  P ← [ ];
  i ← 0;
  bad ← false;
  j ←$ {0, ..., q};
  a' ←$ {0, ..., n − 1};
  (m, σ) ← A(pk);
  h ← H(m)
  return h = f_pk(σ)
      ∧ m ∉ S

H(m) ≜
  if m ∉ dom(L) then
    if i = j then
      h ← g^a'
    else
      a ←$ [0..n − 1];
      h ← f_pk(g^a)
      P ← (m, g^a) :: P;
    i ← i + 1
    L ← (m, h) :: L
    M ← (i, m) :: M;
  return L(m)

Sign(m) ≜
  S ← m :: S;
  h ← H(m);
  if j ∈ dom(M) ∧
   m = M[j] then
    bad ← true
  return f_sk^-1(h)
```

Para poder inferir la igualdad de probabilidades entre los juegos, se prueba la siguiente equivalencia:

$$G_3 \sim G_4 : = \{res, L, S, pk, sk, m, \sigma, M, j, i\}$$

En este caso fue necesario probar invariantes directamente sobre los propios oráculos para que luego sea posible inferir, utilizando tácticas, invariantes en el adversario. Por ejemplo, para resolver la asignación aleatoria de la variable a se le debe indicar a la táctica rnd cómo recuperar la variable relacionando ambos juegos. Dado que L es equivalente en ambos juegos, el valor de hash h es equivalente, por lo que se debe cumplir que: $g^a \equiv f_{pk}(g^a)$. Despejando del lado izquierdo se obtiene que $a = dlg(f_{pk}(g^a))$ y haciendo lo mismo del lado derecho se obtiene que $a = dlg(f_{sk}^{-1}(g^a))$.
Entonces, se aplica la táctica $rnd\ dlg(f_{sk}^{-1}(g^a))\ dlg(f_{pk}(g^a))$ que sustituye en la poscondición cada ocurrencia de a de cada juego por la definición de la variable según se despejó del otro juego.

Luego, se puede probar que:

$$\Pr[G_3 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]] =$$
$$\Pr[G_4 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]]$$

Trabajando en el juego G_4, se prueba que si $M[j] \notin S$, es decir, no se realizó una consulta de firma con tal mensaje, entonces a la bandera bad nunca se le asignó $true$:

$$G_4 \sim G_4 : = \{res, m, S, M, j, bad\} \wedge \{j \in dom(M) \wedge M[j] \notin S\} \Rightarrow \{\neg bad\}$$

Entonces, $\neg bad$ es una poscondición del juego y es posible probar que:

$$\Pr[G_4 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]] =$$
$$\Pr[G_4 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j] \wedge \neg bad]$$

En el juego G_5 de la figura 5.7, se aplica una transformación conservativa, sustituyendo luego de la bandera bad el valor de retorno $P[m]$, dado que es allí donde falla la simulación.

Dado que bad es un evento de falla y los juegos son sintácticamente idénticos hasta que se asigna el valor $verdadero$ a la bandera bad, se aplica el lema fundamental 2.5 y se prueba que:

$$\Pr[G_4 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j] \wedge \neg bad] =$$
$$\Pr[G_5 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j] \wedge \neg bad]$$

Luego, por reglas de probabilidad se prueba que:

$$\Pr[G_5 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j] \wedge \neg bad] \leq$$
$$\Pr[G_5 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]]$$

En el juego G_6 de la figura 5.8 simplemente se sustituye en el oráculo de firma la sentencia condicional por el retorno del valor $P[m]$ y de esta forma se completa la forma deseada de dicho oráculo.

Figura 5.7. Juego G_5

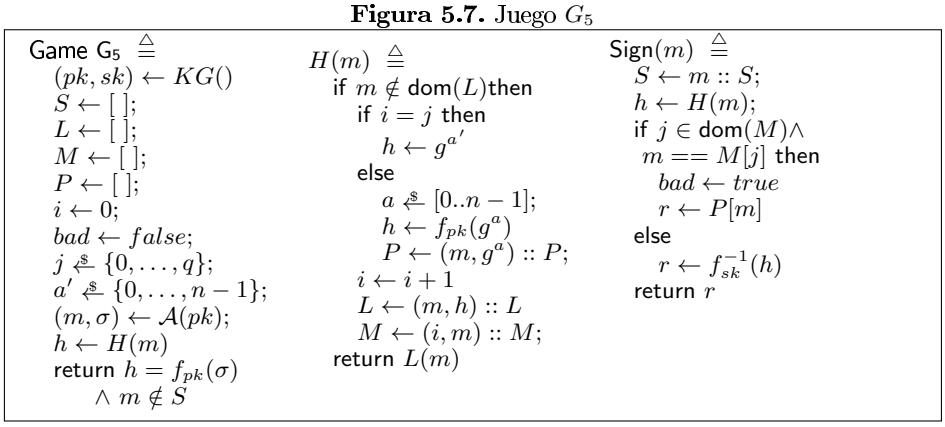
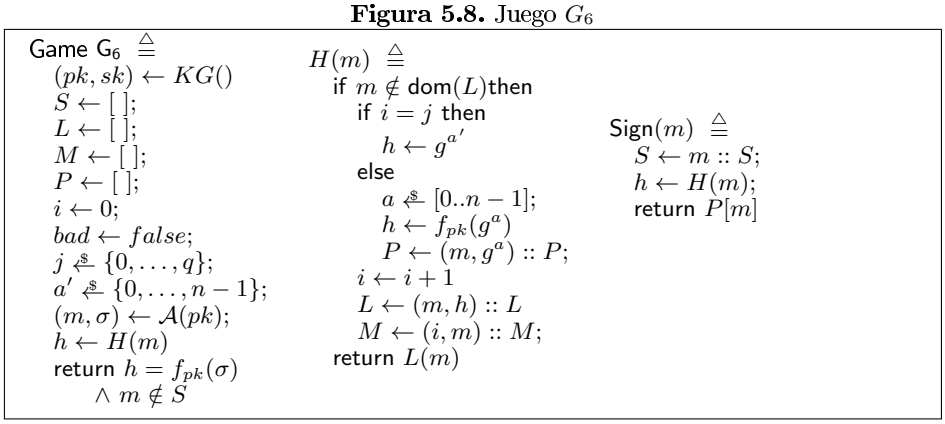


Figura 5.8. Juego G_6

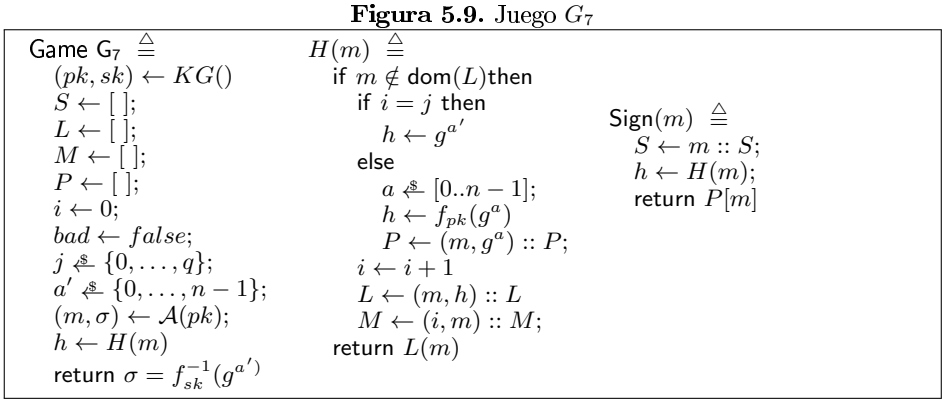


Se prueba la invarianza de todas las variables globales de los juegos y se deduce que:

$$\Pr[G_5 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]] =$$
$$\Pr[G_6 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]]$$

En el juego G_7 de la figura 5.9 el ataque tiene éxito si se adivina la preimágen del elemento desafío $g^{a'}$ representando de esta forma la semántica del ataque del inversor.

Figura 5.9. Juego G_7



Es necesario probar la siguiente equivalencia para luego demostrar la relación de probabilidad que existe entre los juegos.

$$G_6 \sim G_7 : = \{L, S, m, j, M, pk, sk, P, a', sigma, i\} \wedge$$
$$\forall(m').\{m \in dom(L)\} = \{m \in rng(M)\}_{\langle 1 \rangle} \wedge$$
$$\{j \in dom(M)\} \Rightarrow M[j] \in dom(L)\}_{\langle 1 \rangle} \wedge$$
$$\{L[M[j]] = g^{a'}\}_{\langle 1 \rangle} \wedge$$
$$\{res \wedge j \in dom(M) \wedge m = M[j]\}_{\langle 1 \rangle} \Rightarrow \{res\}_{\langle 2 \rangle}$$

Luego, se prueba que:

$$\Pr[G_6 : res \wedge i \leq q \wedge j \in dom(M) \wedge m = M[j]] \leq$$
$$\Pr[G_7 : res \wedge i \leq q]$$

Finalmente, el juego G_{OW} de la figura 5.10 representa el ataque del inversor I a la permutación trampa único-sentido subyacente f.

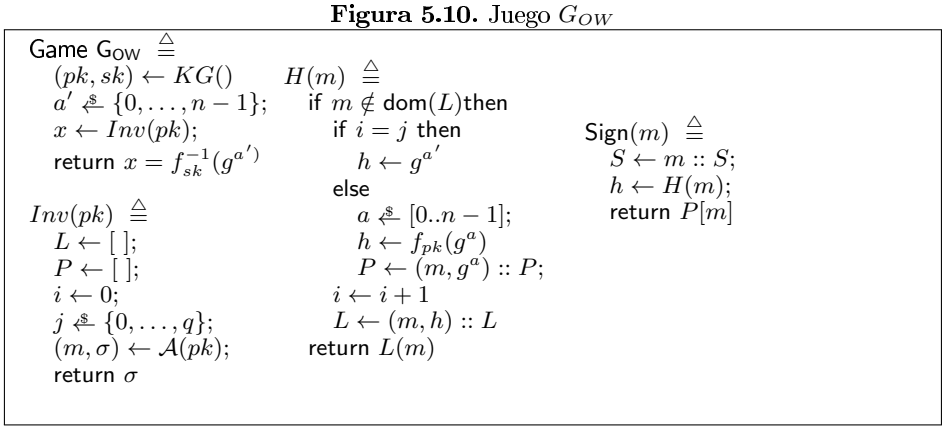
Se prueba la siguiente equivalencia:

$$G_7 \sim G_{OW} : = \{res, a', pk, sk, j\} \wedge \{\sigma\langle 1 \rangle = x\langle 2 \rangle\}$$

y se deduce que:

$$\Pr[G_7 : res \wedge i \leq q] \leq \Pr[G_{OW} : res]$$

Figura 5.10. Juego G_{OW}



Poniendo todos los resultados juntos y aplicando la propiedad de transitividad se tiene que:

$$\Pr[G_{EF} : res \wedge i \leq q] * q^{-1} = ... \leq \Pr[G_{OW} : res]$$

donde

- $q = q_H + q_S + 1$
- $i \leq q$ representa que el adversario puede hacer a lo sumo q consultas al oráculo de hash.

Entonces, se ha demostrado que:

$$\boxed{\Pr[\mathrm{G}_{EF} : res \wedge i \leq q] * q^{-1} \leq \Pr[G_{OW} : res]}$$

Se recuerda que el significado de la variable res en cada uno de los juegos es:

- res_{EF} es la evaluación de la expresión $h = f_{pk}(\sigma) \wedge m \notin S$.
- res_{OW} es la evaluación de la expresión $x = f_{sk}^{-1}(y)$

Entonces, si se sustituye la variable res por lo anterior, se ha demostrado que:

$$\boxed{\Pr[\mathrm{G}_{EF} : h = f_{pk}(\sigma) \wedge m \notin S \wedge i \leq q] * q^{-1} \leq \Pr[G_{OW} : x = f_{sk}^{-1}(y)]}$$

Lo que se quería demostrar.

6
Discusión y Trabajos Relacionados

6.1. La prueba en EasyCrypt

En la sección 5 se describió la prueba de seguridad de FDH utilizando la herramienta EasyCrypt. La reducción de la prueba se estructuró como una secuencia de juegos comenzando con el juego que representa el objetivo de seguridad y finalizando con el juego que representa la primitiva criptográfica. En tal sentido, la herramienta ofrece modelar los juegos como programas en el lenguaje pWhile lo cual permitió capturar la semántica deseada en cada juego de la prueba. Además, las transiciones lógicas entre los juegos son justificadas utilizando la lógica probabilística de Hoare (pRHL) y para ello se utiliza las distintas tácticas que ofrece la herramienta.
Como se discutió en la sección 3 uno de los problemas con que se enfrenta la comunidad criptográfica es que las pruebas de seguridad están siendo cada vez más difíciles de verificar por su complejidad, por ello la importancia de herramientas automatizadas. En tal sentido, EasyCrypt cumple con uno de sus principales objetivos permitiendo escribir y verificar pruebas criptográficas, favoreciendo a que estas sean menos susceptibles a errores. La herramienta permite justificar transformaciones entre los juegos, ofreciendo la utilización de tácticas y pudiendo chequear automáticamente que las transformaciones preservan las propiedades deseadas, como manejo de eventos de fallas y lema fundamental, entre otros.
Otro punto de importancia para el criptógrafo, a la hora de evaluar una herramienta como EasyCrypt, es el esfuerzo necesario para aprender a utilizarla y los conocimientos necesarios en métodos formales para construir la prueba basada en la verificación automática. En este aspecto, como se mencionó en la sección 3, la herramienta predecesora CertiCrypt no tuvo éxito en su adopción por parte de la comunidad debido a su alta curva de aprendizaje en su utilización. Sin embargo, la utilización de EasyCrypt no requiere un gran esfuerzo en la comprensión de las herramientas que ofrece para escribir la prueba y en el aprendizaje de las nociones de verificación para lograr chequear la correctitud de la misma. El artículo [BGHZB11]

que presentó EasyCrypt fue reconocido con el premio *Best Paper Award* en la 31^a conferencia internacional de criptografía en 2011.
Al momento de realizar la prueba, no se contaba con la documentación necesaria para entender el lenguaje y la semántica exacta de las tácticas. El aprendizaje se hizo en gran parte desarrollando la prueba y observando otras pruebas de seguridad desarrolladas en la herramienta. Este punto termina siendo una de sus fortalezas y claramente se evidencia en este trabajo que presenta una prueba de seguridad por parte del criptógrafo. Hoy en día, el equipo de desarrolladores de EasyCrypt ofrece mayor documentación con un manual de usuario de la herramienta [Bar].
Por otro lado, al realizar el trabajo se vió la necesidad de que se mejore en el desarrollo de la interfaz para permitir por ejemplo, resaltar sintaxis, compilar línea por línea y deshacer la aplicación de la última táctica. Este aspecto ha sido trabajado por parte del equipo de desarrolladores. Además, el equipo ha trabajado en ofrecer una salida en formato LaTeX de los juegos, lo cual permite al criptógrafo continuar con la prueba en papel si la herramienta no soporta los argumentos que necesita utilizar y le facilita su escritura en publicaciones.
Es importante destacar, que si bien la herramienta tenía ciertas limitaciones [BGHZB11] a la hora de realizar este trabajo, como por ejemplo, carecer de iteraciones en los juegos, la misma está en continuo desarrollo y estas funcionalidades se fueron agregando en la medida que se requierían. Por ejemplo, al desarrollar la prueba de seguridad del esquema de firma PFDH que se menciona en la sección 6.2, se necesitó utilizar una táctica para que tratara con las iteraciones y esto fue implementado por el equipo de desarrolladores de la herramienta.

La prueba de seguridad del esquema FDH presentada en este trabajo en EasyCrypt resulta más significativamente fácil de entender y rápida de compilar que la presentada en CertiCrypt [Zan10]. En efecto, la prueba desarrollada en EasyCrypt consta de 709 líneas mientrás que la prueba en CertiCrypt consta de 2146 líneas. En tal sentido y con los argumentos expuestos anteriormente, la herramienta cumple con uno de los principales objetivos explicitados por Halevi [Hal05] donde la utilidad de la herramienta dependerá de la adopción que tenga en la comunidad criptográfica y claramente, su adopción será mayor en tanto ofrezca mayor facilidad de comprensión sobre ella. Este punto fue la debilidad de CertiCrypt y es ampliamente superado por la herramienta en cuestión.
CryptoVerif [Bla06; BP06] es otra herramienta en la que se presentó una formalización de la prueba de FDH. La principal desventaja que tiene dicha prueba es que observando la salida de la herramiente es muy dificil para el criptógrafo poder recuperar la reducción hecha en la prueba, a diferencia de la presentada en EasyCrypt donde se sigue de cerca la idea de la reducción. Sin embargo, dicha herramienta fue la primera en ser construida y fue de sumo interés dado que mostró los beneficios de la verificación automatizada de las pruebas para la comunidad criptográfica.

6.2. Variantes de esquemas de firma

El esquema de firma probabilístico de FDH (PFDH) [Cor02] es una versión probabilística simple de FDH donde la firma de un mensaje tiene una componente aleatoria, permitiendo obtener distintas firmas de un mismo mensaje. La familia de permutaciones trampa de único-sentido (KG, f, f^{-1}) y la familia de funciones de hash H son las misma que FDH. El algoritmo de firma retorna en conjunto con la firma, la componente aleatoria r concatenada al mensaje para consultar su valor de hash, de modo que el algoritmo de verificación pueda utilizarla para comprobar la correctitud de la firma. El esquema de firma se compone de la siguiente tripla de algoritmos:

$$KG(n) = (pk, sk) \leftarrow KG_f(n); return(pk, sk)$$
$$Firmar(sk, m) = r \xleftarrow{\$} \{0,1\}^{k_0}, return(f^{-1}(sk, H(m||r)), r)$$
$$Verificar(pk, m, (\sigma, r)) = return(f(pk, \sigma) = H(m||r))$$

Formalmente el esquema es parametrizado con la longitud k_0 de la componente aletoria y se observa que FDH es un caso especial donde dicha componente vale cero. El teorema 6.1 muestra la seguridad del esquema [Zan10].

Teorema 6.1 (Seguridad de PFDH). *Sea A un adversario efectuando un ataque existencial y adaptativo de texto plano elegido contra el esquema de PFDH que hace a lo sumo $q_H(k)$ consultas al oráculo de hash H y $q_S \leq 2^{k_0}$ consultas al oráculo de firma $Sign$. Se supone que el adversario A tiene éxito con probabilidad $\epsilon(k)$ en encontrar la firma de un mensaje fresco dentro de un tiempo t. Entonces, se puede construir un inversor I para la subyacente familia de permutaciones trampa que encuentra la preimagen de un elemento elegido de forma aleatoria y uniforme en el rango de f con probabilidad ϵ' dentro de un tiempo t' donde:*

$$\epsilon' \geq (1 - \frac{1}{q_s})^{q_s}\epsilon \approx exp(-1)\epsilon$$
$$t' \leq t + (q_H + q_S)O(t_f) + q_h q_S O(k_0)$$

Esto significa que cuando $k_0 \geq log_2(q_S)$, la seguridad de PFDH está fuertemente relacionada con el problema de invertir la subyacente permutación. De esta forma es posible lograr una mejor reducción a costa de firmas más largas. Es importante destacar, que a diferencia de FDH, la cota de seguridad de este esquema depende de las consultas al algoritmo de firma y es independiente de las consultas a la función de hash. Desde un punto de vista práctico, la cantidad de consultas de hash está solamente limitada a la cantidad de tiempo y los recursos del atacante, mientrás que la cantidad de consultas de firma pueden ser limitadas por quien posea la clave privada, pudiendo cambiar la misma luego de que cierta cantidad de consultas han sido hechas.
A continuación de la prueba de FDH realizada en EasyCrypt, también se realizó la prueba de PFDH en la herramienta. Dicha prueba no se presenta en este trabajo

dado que está fuera del alcance del mismo quedando su discusión para futuras publicaciones. Sin embargo, es importante destacar que fue factible su desarrollo en la herramienta presentando una secuencia de juegos y transformaciones más complejas que las formalizadas para FDH, constituyendose como otro ejemplo más de su utilidad para la comunidad criptográfica.

Bellare y Rogaway [BR96] propusieron el esquema de firma PSS (Probabilistic Signature Scheme) que se diferencia de PFDH dado que la componente aleatoria no es transmitida de forma separada al firmar el mensaje, sino que se recupera al verificar la firma. Este esquema de firma es el incluido en el estándar PKCS#1 y preserva la seguridad de FDH pero logra esta última con una cota mejor de seguridad. La descripción del esquema y el análisis de su seguridad se puede ver en [BR96; Cor02].

7
Conclusiones y Trabajo A Futuro

En este trabajo se presenta la prueba de la cota original del esquema de firma FDH usando EasyCrypt. En la sección 3 se ha discutido la importancia de la verificación automatizada de las pruebas de seguridad para la comunidad criptográfica. La relevancia de este enfoque ha sido mayormente explicitado por Halevi [Hal05] que propone la creación de una herramienta para ayudar a escribir y verificar las pruebas. La factibilidad de la creación de dicha herramienta ya ha sido demostrada con la presentación en primer termino de CryptoVerif y luego de CertiCrypt como se vió en la sección 3. Sin embargo, como se discutió en la sección 6 dichas herramientas carecen de características relevantes en tanto cumplen parcialmente con los objetivos propuestos por Halevi. En tal sentido, se presenta la herramienta EasyCrypt y es una posible candidata a ser evaluada para adoptarla como estándar por la comunidad criptográfica.
EasyCrypt cumple con un importante objetivo para el criptógrafo en tanto no requiere de un elevado conocimiento en métodos formales y un elevado aprendizaje de utilización para lograr escribir las pruebas. Un punto que contribuirá satisfactoriamente en dicho camino será completar su documentación con más ejemplos de prueba y continuar ofreciendo funcionalidades como la salida en LaTeX de la secuencia de juegos. Dichos puntos ya han sido desarrollados por el equipo de EasyCrypt.
En la sección 5 se ha presentado la prueba de seguridad del esquema de FDH donde se formalizó la prueba de la cota original de un ataque existencial y adaptativo de texto plano elegido contra el esquema utilizando la herramienta. Este esquema es ampliamente utilizado y la prueba presentada en este trabajo agrega valor a la herramienta en tanto contribuye a mostrar su utilidad para el criptógrafo. A su vez, como se mencionó en la sección 6.2 también se demostró la prueba de seguridad contra un ataque existencial y adaptativo de texto plano elegido del esquema de firma PFDH. Esta prueba no ha sido mostrada en las otras dos herramientas mencionadas, lo cual constituye un aporte novedoso en la formalización de las pruebas utilizando una herramienta automatizada. Estos ejemplos, demuestran que

las pruebas en EasyCrypt son significativamente más faciles y rápidas de construir que en otras herramientas y además están estrechamente ligadas al razonamiento de la reducción que hace el criptógrafo al pensar en papel y lápiz.
Es de suma importancia para que la herramienta tenga éxito en constituirse como un estándar seguir formalizando pruebas de seguridad de esquemas de firma utilizados en la comunidad y así demostrar su utilidad. Un próximo paso, como se mencionó en la sección 6.2, es continuar con la demostración de la cota de seguridad contra un ataque existencial y adaptativo de texto plano elegido del esquema de firma PSS.

8
Apéndice

Archivo de Entrada para la prueba de seguridad de FDH

A continuación se muestra el archivo de entrada de EasyCrypt correspondiente con la prueba de seguridad de FDH descripta en la sección 5.

```
1
2  /** Abstract Types **/
3  type group;;
4  type message;;
5  type key_pair;;
6  type secret_key;;
7  type public_key;;
8
9
10 /** Constants **/
11 cnst n      : int;;      // group order
12 cnst g      : group;;    // group generator
13 cnst q      : int;;      // allotted queries to the hash oracle
14 cnst dummy : message;;
15
16
17 /** Type abbreviations **/
18 type signature = group;;
19
20
21 /** Operators **/
22 // Group
23 op (^)    : group, int → group         = group_pow;;
24 op dlg    : group → int                = dlg;;
25
26 // One-way permutation
27 op f      : public_key, group → group = f;;
28 op f_inv : secret_key, group → group = f_inv;;
29
30 // Keys
31 op pkey   : key_pair → public_key      = pkey;;
32 op skey   : key_pair → secret_key      = skey;;
33
34
35 /** Axioms **/
36 axiom q_nat : { 0 ≤ q };;
37
```

```
38 axiom f_inv_left : ∀ (x:group, kp:key_pair).
39   { x = f_inv(skey(kp), f(pkey(kp), x)) };;
40
41 axiom f_inv_right : ∀ (x:group, kp:key_pair).
42   { x = f(pkey(kp), f_inv(skey(kp), x)) };;
43
44 axiom dlg_pow : ∀ (x:group). { (g^dlg(x)) = x };;
45
46 axiom pow_dlg : ∀ (a:int). { dlg(g^a) = a%n };;
47
48 axiom dlg_bound : ∀ (x:group). { 0 ≤ dlg(x) } ∧ { dlg(x) ≤ n-1 };;
49
50 axiom mod_small : ∀ (a:int,n:int). { 0 ≤ a } ⇒ { a < n } ⇒ { a%n = a };;
51
52 axiom real_of_bool_mul_le_l : ∀ (a:bool, b:real).
53   { real_of_bool(a) * b ≤ b };;
54
55 axiom real_of_bool_mul_le_r : ∀ (a:bool, b:real).
56   { b * real_of_bool(a) ≤ b };;
57
58 axiom real_of_bool_and : ∀ (a:bool, b:bool).
59   { real_of_bool(a ∧ b) = real_of_bool(a) * real_of_bool(b) };;
60
61
62 /** Abstract procedures **/
63 adversary A(pk: public_key) : message * group
64 { message → group; message → group };;
65
66 adversary KG() : key_pair {};;
67
68 /** Game definitions **/
69
70 /*
71 ** Game EF:
72 **
73 ** This is the standard existential forgery experiment for FDH
74 */
75 game EF = {
76   var pk : public_key
77   var sk : secret_key
78   var L : (message, group) map
79   var S : message list
80   var i : int
81   var mm : message
82   var sigma : group
83
84   fun H(m:message) : group = {
85     var a : int = [0..n-1];
86     if ( ¬in_dom(m,L) ) {
87       i = i + 1;
88       L[m] = g^a;
89     };
90     return L[m];
91   }
92
93   fun Sign(m:message) : signature = {
94     var h : group;
95     S = m :: S;
96     h = H(m);
97     return f_inv(sk, h);
98    }
99
100  abs A = A {H, Sign}
101  abs KG = KG {}
102
103  fun Main() : bool = {
104    var kp : key_pair;
```

```
105      var h : group;
106      kp = KG();
107      pk = pkey(kp);
108      sk = skey(kp);
109      L = empty_map();
110      S = [];
111      i = 0;
112      (mm, sigma) = A(pk);
113      h = H(mm);
114      return (h = f(pk, sigma) ∧ ¬in(mm, S));
115   }
116  };;
117
118
119  game G1 = EF
120   var M : (int, message) map
121   var I : (message, int) map
122   var j : int
123
124   where H = {
125     var a : int = [0..n-1];
126     if ( ¬in_dom(m,L) ) {
127       L[m] = g^a;
128       M[i] = m;
129       I[m] = i;
130       i = i + 1;
131     };
132     return L[m];
133   }
134
135   and Main = {
136     var kp : key_pair;
137     var h : group;
138     kp = KG();
139     pk = pkey(kp);
140     sk = skey(kp);
141     L = empty_map();
142     M = empty_map();
143     I = empty_map();
144     S = [];
145     i = 0;
146     (mm, sigma) = A(pk);
147     h = H(mm);
148     j = [0..q];
149     return (h = f(pk, sigma) ∧ ¬in(mm,S));
150   };;
151
152  equiv EF_G1 : EF.Main ~ G1.Main: {true} ⟹ ={pk,sk,L,S,i,res};;
153  save;;
154  inline H; derandomize; wp; call inv ={pk,sk,L,S,i};;
155  wp; call inv {true}; pop⟨2⟩ 1; rnd; rnd⟨2⟩; trivial;;
156  save;;
157
158  claim PrEF_G1 : EF.Main[res ∧ i ≤ q] = G1.Main[res ∧ i ≤ q]
159  using EF_G1;;
160
161  equiv G1_G1 : G1.Main ~ G1.Main:
162    {true} ⟹
163    ={pk,sk,L,S,I,M,i,sigma,mm,res} ∧
164    (∀ (m:message). { in_dom(m, I⟨1⟩) = in_dom(m, L⟨1⟩) }) ∧
165    (∀ (m:message). { in_rng(m, M⟨1⟩) = in_dom(m, I⟨1⟩) }) ∧
166    ( {in_dom(mm⟨1⟩,I⟨1⟩)} ) ∧
167      (∀ (m:message, k:int). { in_dom(k,M⟨1⟩) ∧ m = M⟨1⟩[k] } ⇒
168      { in_dom(m,I⟨1⟩) ∧ I⟨1⟩[m] = k }) ∧
169    ( {res⟨1⟩ ∧ i⟨1⟩ ≤ q ∧ in_dom(j⟨1⟩,M⟨1⟩) ∧ (mm⟨1⟩ = M⟨1⟩[j⟨1⟩])} ⇔
170      {res⟨2⟩ ∧ i⟨2⟩ ≤ q ∧ I⟨2⟩[mm⟨2⟩] = j⟨2⟩ ∧ 0 ≤ I⟨2⟩[mm⟨2⟩] ∧
171       I⟨2⟩[mm⟨2⟩] ≤ q } ) ∧
```

```
172   { (res⟨1⟩ ∧ i⟨1⟩ ≤ q) = (res⟨2⟩ ∧ i⟨2⟩ ≤ q) };;
173 inline H; derandomize; wp;;
174 call inv
175   (={pk,sk,L,S,I,M,i} ∧
176    (∀ (m:message). { in_dom(m, I⟨1⟩) = in_dom(m, L⟨1⟩) }) ∧
177    (∀ (m:message). { in_rng(m, M⟨1⟩) = in_dom(m, I⟨1⟩) }) ∧
178    ∀ (m:message, k:int).
179      { in_dom(k,M⟨1⟩) ∧ m = M⟨1⟩[k]} ⇒ { in_dom(m,I⟨1⟩) ∧ (I⟨1⟩[m]=k)});;
180 wp; call inv {true}; repeat rnd; trivial;;
181 save;;
182
183 prover alt-ergo;;
184
185 claim PrG1_G1b :
186   G1.Main[res ∧ i ≤ q ∧ in_dom(j, M) ∧ mm = M[j]] =
187   G1.Main[res ∧ i ≤ q ∧ I[mm] = j ∧ 0 ≤ I[mm] ∧ I[mm] ≤ q]
188 using G1_G1;;
189
190 /* TODO */
191 claim PrG1_G1 : G1.Main[res ∧ i ≤ q] * 1%r / q%r ≤
192   G1.Main[res ∧ i ≤ q ∧ I[mm] = j ∧ 0 ≤ I[mm] ∧ I[mm] ≤ q]
193 using admit;;
194
195
196 game G2 = G1
197  var a' : int // challenge
198
199  where H = {
200    var a : int = [0..n-1];
201    if ( ¬in_dom(m, L) ∧ (i ≤ j ∨ in_dom(M[j], L)) ) {
202      L[m] = g^a;
203      M[i] = m;
204      i = i + 1;
205   };
206   return L[m];
207  }
208
209  and Main = {
210    var kp : key_pair;
211    var h : group;
212    kp = KG();
213    pk = pkey(kp);
214    sk = skey(kp);
215    L = empty_map();
216    M = empty_map();
217    I = empty_map();
218    S = [];
219    i = 0;
220    mm = dummy;
221    j = [0..q];
222    a' = 0;
223    sigma = g;
224    h = g;
225    (mm, sigma) = A(pk);
226    h = H(mm);
227    if (i ≤ j) { a' = [0..n-1]; } else { a' = dlg(L[M[j]]); }
228    return (h = f(pk, sigma) ∧ ¬in(mm, S) );
229  };;
230
231 equiv G1_G2 : G1.Main ~ G2.Main:
232  {true} ⟹
233  ={pk,sk,L,S,M,i,j,mm,res} ∧ ( { j⟨2⟩ < i⟨2⟩ } ⇒ { in_dom(j⟨2⟩, M⟨2⟩) } );;
234 pop⟨1⟩ 4;;
235 inline H; derandomize; wp;;
236 call inv
237   (={pk,sk,L,S,M,i,j} ∧
238    { i⟨2⟩ ≤ j⟨2⟩ ∨ in_dom(M⟨2⟩[j⟨2⟩], L⟨2⟩) } ∧
```

```
239     ( { j⟨2⟩ < i⟨2⟩ } ⇒ { in_dom(j⟨2⟩, M⟨2⟩) } ));;
240 wp; call inv {true}; pop⟨2⟩ 2;;
241 repeat rnd; trivial;;
242 save;;
243
244 claim PrG1_G2 :
245   G1.Main[res ∧ i ≤ q ∧ in_dom(j, M) ∧ mm = M[j]] =
246   G2.Main[res ∧ i ≤ q ∧ in_dom(j,M ) ∧ mm = M[j]]
247 using G1_G2;;
248
249 game G3 = G2
250  where H = {
251     var h : group;
252     var a : int = [0..n-1];
253     if ( ¬in_dom(m,L) ∧ (i ≤ j ∨ in_dom(M[j], L)) ) {
254       if (i = j) {
255         h = g^a';
256       } else {
257         h = g^a;
258       }
259       L[m] = h;
260       M[i] = m;
261       i = i + 1;
262     };
263     return L[m];
264  }
265
266  and Main = {
267     var kp : key_pair;
268     var h : group;
269     kp = KG();
270     pk = pkey(kp);
271     sk = skey(kp);
272     L = empty_map();
273     M = empty_map();
274     I = empty_map();
275     S = [];
276     i = 0;
277     mm = dummy;
278     j = [0..q];
279     a' = 0;
280     sigma = g;
281     h = g;
282     if (i ≤ j) { a' = [0..n-1]; } else { a' = dlg(L[M[j]]); }
283     (mm, sigma) = A(pk);
284     h = H(mm);
285     return (h = f(pk, sigma) ∧ ¬in(mm, S) );
286  };;
287
288 //checkproof;;
289
290 equiv auto G2_G3 : G2.Main ~ G3.Main : {true} ⟹ ={pk,sk,L,S,M,i,j,mm,res}
291 eager { if (i ≤ j) { a' = [0..n-1]; } else { a' = dlg(L[M[j]]); } };;
292
293
294
295 /* H */
296 derandomize; wp;;
297 case⟨1⟩: (i = j ∧ ¬in_dom(m,L) ∧ (i ≤ j ∨ in_dom(M[j],L)) );;
298 rnd; rnd; trivial;;
299 pop⟨2⟩ 1; rnd; rnd; trivial;;
300 save;;
301
302 /* Prefix */
303 app 1 1 ={kp};;
304 auto inv {true};;
305 wp; derandomize; wp;;
```

```
306 rnd; trivial;;
307
308 /* Suffix */
309 trivial;;
310 save;;
311
312 claim PrG2_G3 :
313   G2.Main[res ∧ i ≤ q ∧ in_dom(j, M) ∧ mm = M[j]] =
314   G3.Main[res ∧ i ≤ q ∧ in_dom(j, M) ∧ mm = M[j] ]
315 using G2_G3;;
316
317 game G4 = G3
318   var P : (message, group) map
319   var bad : bool
320
321   where H = {
322    var h : group;
323    var a : int;
324    if ( ¬in_dom(m, L) ) {
325      if (i = j) {
326        h = g^a';
327      }
328      else
329      {
330        a = [0..n-1];
331        h = f(pk, g^a);
332        P[m] = g^a;
333      }
334      L[m] = h;
335      M[i] = m;
336      i = i + 1;
337   };
338   return L[m];
339  }
340
341  and Sign = {
342    var h, r : group;
343    S = m :: S;
344    h = H(m);
345    if ( in_dom(j, M) ∧ (m = M[j]) ) {
346      bad = true;
347    };
348    r = f_inv(sk, h);
349    return r;
350  }
351
352  and Main = {
353    var kp : key_pair;
354    var h : group;
355    kp = KG();
356    pk = pkey(kp);
357    sk = skey(kp);
358    L = empty_map();
359    S = [];
360    M = empty_map();
361    P = empty_map();
362    bad = false;
363    i = 0;
364    mm = dummy;
365    j = [0..q];
366    a' = [0..n-1];
367    sigma = g;
368    h = g;
369    (mm, sigma) = A(pk);
370    h = H(mm);
371    return (h = f(pk, sigma) ∧ ¬in(mm, S));
372  };;
```

```
373
374 equiv H_G3_G4 : G3.H ~ G4.H :
375  ={m,pk,sk,L,S,M,i,j,a'} ∧ {(i ≤ j ∨ in_dom(M[j],L))⟨1⟩} ∧
376   exists (kp:key_pair). { pk⟨1⟩ = pkey(kp) ∧ sk⟨1⟩ = skey(kp) } ⟹
377  ={res,pk,sk,L,S,M,i,j,a'} ∧ {(i ≤ j ∨ in_dom(M[j],L))⟨1⟩} ∧
378   exists (kp:key_pair). { pk⟨1⟩ = pkey(kp) ∧ sk⟨1⟩ = skey(kp) };;
379 derandomize; wp;;
380 rnd (dlg(f_inv(sk,g^a_0))), (dlg(f(pk,g^a_0))); trivial;;
381 save;;
382
383 equiv S_G3_G4 : G3.Sign ~ G4.Sign :
384  ={m,pk,sk,L,S,M,i,j,a'} ∧ {( i≤j ∨ in_dom(M[j],L) )⟨1⟩} ∧
385   exists (kp:key_pair). { pk⟨1⟩ = pkey(kp) ∧ sk⟨1⟩ = skey(kp) } ⟹
386  ={res,pk,sk,L,S,M,i,j,a'} ∧ {( i≤j ∨ in_dom(M[j],L) )⟨1⟩} ∧
387   exists (kp:key_pair). { pk⟨1⟩ = pkey(kp) ∧ sk⟨1⟩ = skey(kp) };;
388 inline H; derandomize; wp;;
389 rnd (dlg(f_inv(sk,g^a_0))), (dlg(f(pk,g^a_0))); trivial;;
390 save;;
391
392 equiv G3_G4 : G3.Main ~ G4.Main :
393  {true} ⟹
394  ={res,L,S,pk,sk,mm,sigma,M,j,i} ∧
395   exists (kp:key_pair). { pk⟨1⟩ = pkey(kp) ∧ sk⟨1⟩ = skey(kp) };;
396
397 app 1 1 ={kp};;
398 auto inv {true};;
399
400 app 9 10
401  (={pk,sk,L,S,M,i,j} ∧ {i⟨1⟩ ≤ j⟨1⟩} ∧
402    exists (kp:key_pair). { pk⟨1⟩ = pkey(kp) ∧ sk⟨1⟩ = skey(kp) });;
403 rnd; wp; repeat rnd; trivial;;
404
405 inline H; derandomize; wp;;
406 auto inv
407  (={pk,sk,L,S,M,i,j,a'} ∧ {(i ≤ j ∨ in_dom(M[j],L))⟨1⟩} ∧
408    exists (kp:key_pair). { pk⟨1⟩ = pkey(kp) ∧ sk⟨1⟩ = skey(kp) });;
409 rnd (dlg(f_inv(sk,g^a_0))), (dlg(f(pk,g^a_0)));
410 rnd; trivial;;
411 save;;
412
413 claim PrG3_G4 : G3.Main[res ∧ i ≤ q ∧ in_dom(j,M) ∧ mm = M[j] ] =
414                 G4.Main[res ∧ i ≤ q ∧ in_dom(j,M) ∧ mm = M[j] ]
415 using G3_G4;;
416
417 equiv G4_G4 : G4.Main ~ G4.Main :
418  {true} ⟹
419  ={res,mm,S,M,j,bad,i} ∧
420  ( { res⟨1⟩ } ⇒ { ¬in(mm⟨1⟩,S⟨1⟩) } ) ∧
421  ( { ( in_dom(j,M) ∧ ¬in(M[j],S))⟨1⟩ } ⇒ {¬bad⟨1⟩} );;
422 app 1 1 ={kp};;
423 auto inv {true};;
424 auto inv (={pk,sk,L,S,M,i,j,a',bad} ∧
425      (∀ (k:int). { in_dom(k,M⟨1⟩) } ⇒ { k < i⟨1⟩ }) ∧
426      (∀ (m:message). { in(m, S⟨1⟩) } ⇒ { in_rng(m, M⟨1⟩) }) ∧
427      (∀ (m:message). { in_dom(m, L⟨1⟩) } = in_rng(m, M⟨1⟩) }) ∧
428      ({ ( in_dom(j,M) ∧ in(M[j],S))⟨1⟩ } <> { bad⟨1⟩ }));;
429 repeat rnd; wp; trivial;;
430 save;;
431
432 prover alt-ergo;;
433
434 claim PrG4_G4 : G4.Main[res ∧ i ≤ q ∧ in_dom(j,M) ∧ mm = M[j] ] =
435                 G4.Main[res ∧ i ≤ q ∧ in_dom(j,M) ∧ mm = M[j] ∧ ¬bad]
436 using G4_G4;;
437
438
439 game G5 = G4
```

```
 where Sign = {
  var h, r : group;
  S = m :: S;
  h = H(m);
  if ( in_dom(j,M) ∧ (m = M[j]) ) {
    bad = true;
    r = P[m];
  } else {
    r = f_inv(sk, h);
  }
  return r;
 };;

checkproof;;
equiv G4_G5 :  G4.Main ~ G5.Main :
 {true} ⟹ (={bad}) ∧  ({¬bad⟨2⟩} ⇒ ={res,mm,j,M,i});;
app 1 1 ={kp};;
auto inv {true};;
ggg
auto inv upto { bad } with ( ={L,S,pk,sk,M,j,i,mm,P,sigma,a'} ) ;;
rnd; derandomize; wp; repeat rnd; trivial;;
save;;

claim PrG4_G5 :
  G4.Main[ res ∧ i≤ q ∧ in_dom(j,M) ∧ mm = M[j] ∧ ¬bad] ≤
  G5.Main[ res ∧ i≤ q ∧ in_dom(j,M) ∧ mm = M[j] ∧ ¬bad]
using G4_G5;;

claim PrG5_G5 :
  G5.Main[ res ∧ i ≤ q ∧ in_dom(j,M) ∧ mm = M[j] ∧ ¬bad] ≤
  G5.Main[ res ∧ i ≤ q ∧ in_dom(j,M) ∧ mm = M[j] ]
same;;


game G6 = G3
 var P : (message, group) map

 where H = {
  var h : group;
  var a : int;
  if ( ¬in_dom(m,L) ) {
   if ( (i = j) ) {
    h = g^a';
   } else  {
    a = [0..n-1];
    h = f(pk,g^a);
    P[m] = g^a;
   }
   M[i] = m;
   i = i + 1;
   L[m] = h;
  };
  return L[m];
 }

 and Sign = {
  var h : group;
  S = m :: S;
  h = H(m);
  return P[m];
 }

 and Main = {
  var kp : key_pair;
  var h : group;
  kp = KG();
  pk = pkey(kp);
```

```
507    sk = skey(kp);
508    L = empty_map();
509    S = [];
510    M = empty_map();
511    P = empty_map();
512    i = 0;
513    mm = dummy;
514    j = [0 .. q];
515    a' = [0..n-1];
516    sigma = g;
517    h = g;
518    (mm,sigma) = A(pk);
519    h = H(mm);
520    return ( h = f(pk, sigma) ∧ ¬in(mm,S) );
521   };;
522
523 equiv G5_G6 : G5.Main ~ G6.Main :
524  {true} ⟹ ={res,L,S,mm,j,M,pk,sk,P,a',i,sigma}
525  app 1 1 ={kp};;
526  auto inv {true};;
527  auto inv
528   (={pk,sk,L,S,M,i,j,a',P,mm} ∧
529    (∀ (k:int). { in_dom(k,M⟨1⟩) } ⇒ { k < i⟨1⟩ }) ∧
530    (∀ (m:message). { in_dom(m, L⟨1⟩) } <=>
531    { in_dom(m, P⟨1⟩) } ∨ (in_dom(j⟨1⟩,M⟨1⟩) ∧ M⟨1⟩[j⟨1⟩] = m) }) ∧
532    (∀ (m:message). { in_dom(m,P⟨1⟩) ∧ in_dom(m,L⟨1⟩) } ∧
533     ( (¬in_dom(j⟨1⟩,M⟨1⟩)) ∨ (m<>M⟨1⟩[j⟨1⟩]) ) } ⇒
534     {f_inv(sk⟨1⟩, L⟨1⟩[m]) = P⟨1⟩[m]}) ∧
535   exists (kp:key_pair). { pk⟨1⟩ = pkey(kp) ∧ sk⟨1⟩ = skey(kp) });;
536 repeat rnd; wp; repeat rnd; trivial;;
537 save;;
538
539 claim PrG5_G6 :
540   G5.Main[res ∧ i ≤ q ∧ in_dom(j,M) ∧ mm = M[j] ] =
541   G6.Main[res ∧ i ≤ q ∧ in_dom(j,M) ∧ mm = M[j]]
542 using G5_G6;;
543
544
545 game G7 = {
546  var L : (message, group) map
547  var S : message list
548  var pk : public_key
549  var sk : secret_key
550  var sigma : group
551  var mm : message
552  var M :  (int, message) map
553  var j :  int
554  var i :  int
555  var a' : int
556  var P : (message, group) map
557
558  fun H(m : message) : group = {
559   var h : group;
560   var a : int;
561   if ( ¬in_dom(m,L) ) {
562    if ( (i = j) ) {
563     h = g^a';
564    } else  {
565     a = [0..n-1];
566     h = f(pk,g^a);
567     P[m] = g^a;
568    }
569    M[i] = m;
570    i = i + 1;
571    L[m] = h;
572   };
573   return L[m];
```

```
574  }
575
576  fun Sign(m: message) : group = {
577   var h : group;
578   S = m :: S;
579   h = H(m);
580   return P[m];
581  }
582
583  abs A = A {H, Sign}
584  abs KG = KG {}
585
586  fun Main() : bool = {
587   var kp : key_pair;
588   var h : group;
589   kp = KG();
590   pk = pkey(kp);
591   sk = skey(kp);
592   L = empty_map();
593   S = [];
594   M = empty_map();
595   P = empty_map();
596   i = 0;
597   mm = dummy;
598   j = [0 .. q];
599   a' = [0..n-1];
600   sigma = g;
601   h = g;
602   (mm, sigma) = A(pk);
603   h = H(mm);
604   return (f_inv(sk, g^a') = sigma);
605  }
606 }
607
608 prover simplify;;
609
610 equiv G6_G7 : G6.Main ~ G7.Main :
611  {true} ==>
612  ={L,S,mm,j,M,pk,sk,P,a',sigma,i} /\
613  (forall (k:int). { in_dom(k,M⟨1⟩) } => { k < i⟨1⟩ }) /\
614  (forall (m:message). { in_dom(m, L⟨1⟩) = in_rng(m, M⟨1⟩) }) /\
615  ({in_dom(j⟨1⟩,M⟨1⟩)} => { in_dom(M⟨1⟩[j⟨1⟩],L⟨1⟩) /\
616     (L⟨1⟩[M⟨1⟩[j⟨1⟩]] = g^a'⟨1⟩)}) /\
617  ({ (res /\ in_dom(j,M) /\ mm = M[j])⟨1⟩ } => { res⟨2⟩ });;
618 app 1 1 ={kp};;
619 auto inv {true};;
620 inline H; derandomize; wp;;
621 auto inv (={L,S,mm,j,M,pk,sk,P,a',i,sigma} /\
622     (forall (k:int). { in_dom(k,M⟨1⟩) } => { k < i⟨1⟩ }) /\
623     (forall (m:message). { in_dom(m, L⟨1⟩) = in_rng(m, M⟨1⟩) }) /\
624     ({in_dom(j⟨1⟩,M⟨1⟩)} =>
625      { in_dom(M⟨1⟩[j⟨1⟩],L⟨1⟩) /\ (L⟨1⟩[M⟨1⟩[j⟨1⟩]] = g^a'⟨1⟩)}));;
626 repeat rnd; trivial;;
627 save;;
628
629 claim PrG6_G7 :
630   G6.Main[res /\ i <= q /\ in_dom(j,M) /\ mm = M[j] ] <=
631   G7.Main[res /\ i <= q]
632 using G6_G7;;
633
634
635 game OW = {
636  var a' : int
637  var L : (message, group) map
638  var P : (message, group) map
639  var pk : public_key
640  var sk : secret_key
```

```
641  var x : group
642  var i : int
643  var j : int
644
645  fun H (m:message) : group = {
646   var h : group;
647   var a : int;
648   if ( ¬in_dom(m,L) ) {
649    if (i = j)  {
650     h = g^a';
651    } else {
652     a = [0..n-1]; // challenge
653     h = f(pk,g^a);
654     P[m] = g^a;
655    }
656     i = i + 1;
657     L[m] = h;
658   };
659   return L[m];
660  }
661
662  fun Sign (m : message) : group = {
663   var h : group;
664   h = H(m);
665   return P[m];
666  }
667
668  abs A = A {H, Sign}
669  abs KG = KG {}
670
671  fun Inv (pk': public_key) : group = {
672   var m : message;
673   var sigma : group;
674
675   L = empty_map();
676   P = empty_map();
677   i = 0;
678   m = dummy;
679   j = [0..q];
680   sigma = g;
681   (m,sigma) = A(pk');
682   return sigma;
683  }
684
685   fun Main() : bool = {
686     var kp : key_pair;
687     kp = KG();
688     pk = pkey(kp);
689     sk = skey(kp);
690     a' = [0..n-1];
691     x = g;
692     x = Inv(pk);
693     return f_inv(sk, g^a') = x;
694  }
695 };;
696
697 prover alt-ergo;;
698
699 equiv G7_OW : G7.Main ~ OW.Main :
700   {true} ==> ={res,a',pk,sk,j} ∧ { sigma⟨1⟩ = x⟨2⟩ };;
701 app 1 1 ={kp};;
702 auto inv {true};;
703 inline Inv, H; derandomize;; wp;;
704 auto inv (={a',pk,sk,j,i,L,P}  ∧ { sigma⟨1⟩ = x⟨2⟩ });;
705 pop⟨1⟩ 2; pop⟨1⟩ 1; repeat rnd; trivial;;
706 save;;
707
```

```
708 claim PrG7_OW : G7.Main[res ∧ i ≤ q] ≤ OW.Main[res]
709 using G7_OW;;
```

Referencias

Bar. Zanella Barthe. The easycrypt tool. URL: `https://www.easycrypt.info/trac/`. Cited in page 35.

BGHZ09. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt. In *5th International workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *Lecture Notes in Computer Science*, pages 1–19, Heidelberg, 2009. Springer. Cited in pages 2 and 12.

BGHZB11. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, Lecture Notes in Computer Science. Springer, 2011. Cited in pages 2, 12, 13, 34, and 35.

BGLZB11. Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. Beyond provable security. Verifiable IND-CCA security of OAEP. In *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196, Heidelberg, 2011. Springer. Cited in pages 12 and 13.

BGZ09. Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2009*, pages 90–101, New York, 2009. ACM. Cited in pages 2, 12, 14, and 16.

BGZB10. Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Programming language techniques for cryptographic proofs. In *1st International conference on Interactive Theorem Proving, ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 115–130, Heidelberg, 2010. Springer. Cited in pages 2 and 12.

Bla06. Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *27th IEEE symposium on Security and Privacy, S&P 2006*, pages 140–154. IEEE Computer Society, 2006. Cited in pages 2, 12, and 35.

Bob. Contejean Bobot, Conchon. The alt-ergo automated theorem prover. URL: `http://alt-ergo.lri.fr`. Cited in page 15.

BP06. Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554, Heidelberg, 2006. Springer. Cited in pages 12 and 35.

BR93. Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *1st ACM conference on Computer and Communications Security, CCS 1993*, pages 62–73, New York, 1993. ACM. Cited in pages 4, 5, 17, 19, and 20.

BR96. Mihir Bellare and Phillip Rogaway. The exact security of digital signatures – How to sign with RSA and Rabin. In *Advances in Cryptology – EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416, Heidelberg, 1996. Springer. Cited in pages 17, 19, and 37.

BR04. Mihir Bellare and Phillip Rogaway. The game-playing technique, 2004. URL: `http://eprint.iacr.org/2004/331`. Cited in pages 7, 8, and 10.

BR06. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, Heidelberg, 2006. Springer. Cited in pages 11 and 16.

Cor02. Jean-Sébastien Coron. Optimal security proofs for PSS and other signature schemes. In *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 272–287, Heidelberg, 2002. Springer. Cited in pages 36 and 37.

DNS05. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005. Cited in page 15.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Advances in Cryptology – CRYPTO 1986*, pages 186–194. Springer, 1987. Cited in page 4.

GM82. Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 365–377, New York, NY, USA, 1982. ACM Press. Cited in page 6.

GMR88. Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988. Cited in page 4.

Hal05. S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. Cited in pages 2, 11, 35, and 38.

JK03. J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003. Cited in page 17.

JYL01. Bengt Jonsson, Wang Yi, and Kim G. Larsen. Probabilistic extensions of process algebras. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 685–710. Elsevier, Amsterdam, 2001. Cited in page 14.

KR96. Joe Kilian and Phillip Rogaway. How to protect des against exhaustive key search. In *Journal of Cryptology*, pages 252–267. Springer-Verlag, 1996. Cited in page 7.

RSA78. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21:120–126, 1978. Cited in page 5.

Sho04. Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. 2004. Cited in page 7.

The10. The Coq development team. Coq Proof Assistant Reference Manual Version 8.3. Online – `http://coq.inria.fr`, 2010. Cited in page 16.

Zan10. Santiago Zanella Béguelin. *Formal Certification of Game-Based Cryptographic Proofs*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris – Mines ParisTech, 2010. Cited in pages 2, 12, 35, and 36.

ZGBO09. Santiago Zanella Béguelin, Benjamin Grégoire, Gilles Barthe, and Federico Olmedo. Formally certifying the security of digital signature schemes. In *30th IEEE symposium on Security and Privacy, S&P 2009*, pages 237–250, Los Alamitos, 2009. IEEE Computer Society. Cited in page 12.