# An Executable Model for Security Protocol JFKr

David L. Rager
Department of Computer Sciences, The University of Texas at Austin
Taylor Hall 2.124, Austin, TX 78712
ragerdl@cs.utexas.edu

## ABSTRACT

JFKr is a security protocol that establishes a shared encryption key between two participants. This paper briefly describes the different components of JFKr and the security property each component is intended to provide. It then describes an executable model, interleaving pieces of code to help the reader understand how the model represents the protocol specification. Finally, it presents some theorems about the model.

The contributions of this work include (1) an executable model for a key establishment protocol about which we can reason, (2) a model for an attacker that permits the injection, modification, and removal of messages between the participants, and (3) formalizations of a subset of desired security properties.

## Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Software/Program Verification*

## General Terms

security protocol verification

## Keywords

JFKr, security, key establishment protocol, verification, theorem proving, ACL2

## 1. INTRODUCTION

JFKr is a security protocol that establishes a shared key between two participants. Its roots stem from the original Diffie-Hellman protocol [3]. Through this original protocol, two parties took advantage of the properties of modular exponentiation to create a key from one private piece of information and one public piece of information. Since this original Diffie-Hellman protocol had no method for authenticating users, no notion of protecting identity (since there was no identity exchange in the first place), and was vulnerable to denial of service attacks, it has evolved over time into ISO 9798-3, Internet Key Exchange (IKE), and finally Just Fast Keying (JFK).

As with any security protocol, we would like to have formal proofs for why a protocol guarantees the behavior we expect. Many have attempted proofs establishing the following properties: the establishment of a secret and shared key, authentication, protection of identity, prevention of denial of service attacks, and resilience against replay attacks. Our goal is to present another possible method of proof for the first two of these properties, specifically in the ACL2 theorem prover.

This paper briefly describes the different security components of JFKr and the security property each component is intended to provide. It then describes an executable model, interleaving pieces of code to help the reader understand how the model represents the protocol specification. Finally, it mentions two proven authentication theorems and an unproven theorem about the sameness of the derived key.

## 2. PREVIOUS WORK

Previous work includes verification of parts of JFKr, including key secrecy by Larry Paulson [4]. Datta and his co-authors also identify some properties worth proving [2]. JFK has also been modeled by Abadi and his co-authors in the pi calculus [1]. Much of this work is based off what Vitaly Shmatikov has taught me.

## 3. PARTIAL EXPLANATION OF JFKR

The following figure shows the network messages exchanged during the protocol's execution. These message components have abbreviated names, as follows. N stands for nonce[1]. X stands for a public Diffie-Hellman value (explained in section 3.1). T and h are both keyed hashes of the values displayed in the diagram. E is an encryption of the specified values. The diagram omits some of the irrelevant message components, but explanations of the relevant components follow afterwards. This diagram is based off a set of slides by Shmatikov [5].

[1]A nonce is just a random value. Nonces are used to prevent *replay-attacks*, which are not covered in this paper

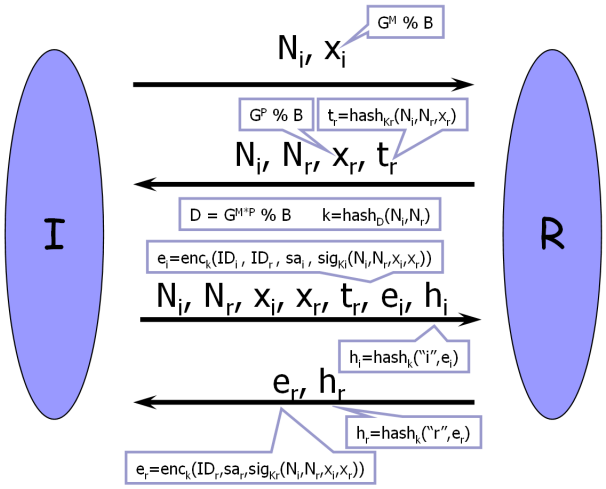


**Figure 1: Simplified JFKr Message Exchange**

## 3.1 Derivation of a Secret Shared Key

One aspect central to understanding this work is the derivation of a shared key. The Diffie-Hellman protocol works as so: The initiator (I) and responder (R) both pick a random value. In this explanation, the initiator's random value is labeled M, and the responder's random value is labeled P. The initiator and responder also publicly share two random values, G and B. G will serve as the base for modular exponentiation and B will serve as the modulo for modular exponentiation[2]. The initiator publicly sends ($G^M$) % B (named $x_i$ in the diagram), and the responder publicly sends ($G^P$) % B (named $x_r$ in the diagram). They then raise each others' public value to their own private value and compute a shared number which they can use to derive a shared key[3]. Unless an actor has at least one of the private values, it is computationally infeasible to discover the shared value. This infeasibility guarantees that the key remains secret to the initiator and responder.

## 3.2 Authentication

During the protocol the initiator and responder authenticate themselves to each other. They accomplish this by: (1) telling each other who they are and (2) providing a signature of a message which requires a private key to sign.

In our proofs we assume the private key to indeed be private and that no other actors or attackers have access to it. The hashes under the established key ($h_i$ and $h_r$ in the diagram) guarantee the integrity of the signature.

## 3.3 Other Security Properties

Security properties intentionally omitted from this paper include protection of identity, protection from denial of service attacks, and the prevention of replay attacks. All of these properties are an important part of JFKr's evolution as a protocol, but due to their complexities, a deeper explanation has been omitted from this paper. The unexplained portions of the diagram are related to these properties.

[2]Modular exponentiation has the property that (((G$^M$) % B)$^P$) % B == (((G$^P$)% B)$^M$) % B.

[3]This shared key is computed from the shared number and the two nonces sent in messages 1 and 2.

# 4. BUILDING A PROOF

Before delving into our JFKr protocol proofs, we must first establish a model for cryptography and prove some properties about the Diffie-Helman protocol. After establishing these two components, we present our theorems about JFKr.

## 4.1 An Assumption: Perfect Cryptography

As with any security protocol, we must first develop a model for cryptography functions. Real cryptography functions tend to be heavily-laden with complex math and are thus difficult to reason about. Due to this complexity, security experts often assume the cryptography portion of protocols to be perfect[4]. Since the use of these perfect cryptography functions is standard, we created our own toy versions of the functions about which we can easily prove these properties. These toy versions are executable and could be replaced by the actual cryptography functions. An example[5] and its relevant theorems follow.

```
(defun encrypt-asymmetric-list (lst key)
  (if (atom lst)
      nil
    (cons (+ (car lst) key)
          (encrypt-asymmetric-list (cdr lst) key))))

(defun decrypt-asymmetric-list (lst key)
  (if (atom lst)
      nil
    (cons (+ (car lst) key)
          (decrypt-asymmetric-list (cdr lst) key))))

; Decrypting an asymmetrically encrypted list with a
; public-private key pair yields the original list.

(defthm decrypt-of-encrypt-asymmetric-equals-plaintext
  (implies (and (encryptable-listp lst) ; integer-listp
                (public-private-key-pairp key1 key2))
           (and (equal (decrypt-asymmetric-list
                        (encrypt-asymmetric-list lst key1)
                        key2)
                       lst)
                (equal (decrypt-asymmetric-list
                        (encrypt-asymmetric-list lst key2)
                        key1)
                       lst))))

; Decrypting an asymmetrically encrypted list with keys
; that are not a public-private key pair does not yield
; the original list.

(defthm decrypt-of-encrypt-asymmetric-needs-key
  (implies (and (encryptable-listp lst) ; integer-listp
                (not (null lst))
                (keyp keyA)
                (keyp keyB)
                (not (public-private-key-pairp keyA keyB)))
           (not (equal (decrypt-asymmetric-list
                        (encrypt-asymmetric-list lst keyA)
                        keyB)
                       lst))))
```

[4]Perfect cryptography is the notion that there exist ideal functions that provide a set of security properties.

[5]This example shows a pair of toy cryptography functions that use public-private-key infrastructure to encrypt and decrypt a list. In this implementation, the public key is an integer and the private key is the negative value of that integer. It is called asymmetric encryption because the encryption and decryption keys are different values.

After proving these theorems, the definitions of `encrypt-asymmetric-list` and `decrypt-asymmetric-list` are disabled. This causes ACL2 to use the proven abstractions instead of the actual definitions of the functions. These definitions and proofs, along with other cryptography definitions, can be found in the ACL2 book `security/jfkr/encryption.lisp`.

## 4.2 The Diffie-Helman Library

It was necessary to develop an executable version of the Diffie-Helman protocol that we could use for reasoning and execution. This implementation can be found in the book `security/jfkr/diffie-helman.lisp`. A few of the more interesting definitions and theorems follow[6]. Thanks go to Robert Krug for developing most of the lemmas in the Diffie-Helman book.

```
(defun compute-public-dh-value (g exponent-value b)
  (mod (expt g exponent-value) b))

(defun compute-dh-key
  (a-public-exponentiation a-private-value b)
  (mod (expt a-public-exponentiation a-private-value) b))

(defthm dh-computation-produces-the-same-key
  (implies ...
   (equal
    (compute-dh-key (compute-public-dh-value g x-exponent b)
                    y-exponent
                    b)
    (compute-dh-key (compute-public-dh-value g y-exponent b)
                    x-exponent
                    b))))
```

Before we can prove that the derivation of the session key is privy only to the initiator and responder, we must formalize the notion that the derivation of the session key requires either the initiator's or responder's privately-chosen exponent[7]. Note that the `guard` of the following function is `nil`, preventing the function from being evaluated. We thank Matt Kaufmann for helping formalize this assumption.

```
(defun session-key-requires-one-part-of-key
  (g b x-exponent y-exponent cracker-guess)
  (declare (xargs :guard nil
                  :verify-guards nil))
  (implies (and ...
                (not (equal cracker-guess x-exponent))
                (not (equal cracker-guess y-exponent)))

           (let ((x-public-value
                  (compute-public-dh-value g x-exponent b))
                 (y-public-value
                  (compute-public-dh-value g y-exponent b))
                 (session-key
                  (compute-dh-key
                   (compute-public-dh-value g x-exponent b)
                   y-exponent
                   b)))
             (and
              (not (equal
                    (compute-dh-key x-public-value
                                    cracker-guess b)
                    session-key))
              (not (equal
                    (compute-dh-key y-public-value
                                    cracker-guess b)
                    session-key))))))
```

[6]The type constraints that require `g`, `b`, `x-exponent`, and `y-exponent` to be positive integers are elided to save space.

[7]The actor's privately-chosen exponent is labeled `cracker-guess` in the function `session-key-requires-one-part-of-key`.

# 5. AN EXECUTABLE MODEL

We considered two approaches in modeling the protocol within ACL2. First, we could find C code that implements this protocol, create an interpreter for that code, and then try and prove some properties about that code. Unfortunately, creating such an interpreter would take an inordinate amount of time. As an alternative, we decided to implement the protocol in ACL2 itself. Since ACL2 is an executable programming language, if the model is written correctly, we will be able to execute it and convince ourselves through testing that the model at least seems to behave correctly.

## 5.1 Steps of the Protocol

We now explain how we implement the protocol. The first item to establish is the concept of what an initiator knows and what a responder knows. An actor[8] can pull its knowledge from one of three places: (1) a list of constants that it created before running the protocol, (2) a list of constants in the public domain, or (3) the state it has accumulated while running the protocol. As each actor goes through a step of the protocol, it will update its own state with information gained during that step. The initiator and responder have completely separate sets of constants (1) and completely separate states (3). This separation of actor constants and states is important for reasoning about actors individually. At some point, we will want to say the initiator runs the protocol for its three steps, and at the end, the initiator will have a successful connection only if the signature given in his received message matches the public key associated with the identity of the responder, with which he thinks he is connected. A statement like this requires being able to reason about the initiator separately from any responder. Below is an example of running the initiator and responder in a deterministic protocol that does not allow an attacker to intercept and modify messages:

```
(mv-let
 (network-s-after-1 initiator-s-after-1)
 (initiator-step1 network-s initiator-s
                  initiator-constants public-constants)
 (mv-let
  (network-s-after-2 responder-s-after-2)
  (responder-step1 network-s-after-1 responder-s
                   responder-constants public-constants)
  (mv-let
   (network-s-after-3 initiator-s-after-3)
   (initiator-step2 network-s-after-2 initiator-s-after-1
                    initiator-constants public-constants)
   (mv-let
    (network-s-after-4 responder-s-after-4)
    (responder-step2 network-s-after-3 responder-s-after-2
                     responder-constants public-constants)
    (mv-let
     (network-s-after-5 initiator-s-after-5)
     (initiator-step3 network-s-after-4 initiator-s-after-3
                      initiator-constants public-constants)
     (mv network-s-after-5 initiator-s-after-5
         responder-s-after-4))))))
```

In a real execution environment, the network could contain any set of information, either valid network messages between legitimate actors, random garbage from an attacker, or even carefully crafted messages from an attacker that try to match what a legitimate actor would send.

Each attempt to make progress is called a step in the protocol. The initiator has three steps, and the responder has

[8]An actor is an initiator or responder.

two steps. A simulation of the protocol between an initiator and responder involves letting an initiator add a message to the network. Then the responder reads and adds a message, and they continue to read and add messages until they have completed their steps. With our cryptography and Diffie-Helman books, we can formalize the notion that, at the end of a run without an attacker, both (1) the initiator and responder will have the same key and (2) the key is only known to them.

## 5.2 Modeling the Attacker

We model the attacker by interleaving functions defined with `defstub`'s between the legitimate parties' network communications[9]. This allows the attacker to inject, drop, and modify messages on the network. The following formalization shows how the attacker can be interleaved from the perspective of the initiator:

```
(mv-let
 (network-s-after-1 initiator-s-after-1)
 (initiator-step1 network-s initiator-s
                  initiator-constants public-constants)
 (mv-let
  (network-s-after-2)
  (function-we-know-nothing-about network-s-after-1)
  (mv-let
   (network-s-after-3 initiator-s-after-3)
   (initiator-step2 network-s-after-2 initiator-s-after-1
                    initiator-constants public-constants)
    (mv-let
      (network-s-after-4)
      (function-we-know-nothing-about network-s-after-3)
      (mv-let
       (network-s-after-5 initiator-s-after-5)
       (initiator-step3 network-s-after-4 initiator-s-after-3
                        initiator-constants public-constants)
       (list network-s-after-5 initiator-s-after-5))))))
```

## 5.3 Formalized Properties

The properties we focus on formalizing follow. The first two have been proven, and the proof of the third remains as future work.

1. If the responder successfully completes the protocol, the responder has established the protocol with a party that knows the initiator's private key.
2. If the initiator successfully completes the protocol, the initiator has established the protocol with a party that knows the responder's private key.
3. If both the initiator and responder successfully complete the protocol, they have the same view of the derived key.

## 5.4 Theorems

The proofs of the first and second property are quite similar to each other. The theorem we want to prove for the first property is named `run-4-steps-with-badly-forged-attacker-yields-responder-failure` and can be found in `security/jfkr/jfkr.lisp`. Note that it is the contra-positive of this theorem that represents the desired property. When proving that the responder fails, the protocol need only run for four steps, because the responder completes the protocol after step four. We have another version of the theorem also accepted by ACL2 that applies to the initiator.

[9]The `defstub`'d functions are named `function-we-know-nothing-about`$X$, where $X$ is a number from 1 to 4

The third goal of proving key sameness involves comparing the end state of the initiator to the end state of the responder. The theorem about key samness, shown in the end of `security/jfkr/jfrk.lisp`, is future work.

## 5.5 Lessons Learned

In the beginning of this project we tried to reason about specific constant lists instead of defining what a constant list looks like and then creating theorems about the constant list abstraction. Given there is almost no point in proving theorems about particular executions, this abstraction is necessary for doing anything useful.

We have already demonstrated how lemmas about cryptography functions provide a nice level of abstraction. If we later substitute the actual cryptography functions for our toy functions, the abstraction will be even more important. It is likely that if one uses the real functions, an assumption similar to the assumption about the Diffie-Helman protocol will need to be formalized.

# 6. CONCLUSION

We have created an executable model of JFKr that can be updated to use real cryptography functions once abstractions about those functions are created, and we have created a method for letting a nondeterministic attacker into the system. Additionally, we have formalized the properties related to identity verification and the derived encryption key. Although this work is not completely novel, taking a different approach than others gives us higher assurance that the protocol is correct and lays the groundwork for verification of other security protocols in ACL2.

# 7. ACKNOWLEDGMENTS

As mentioned through out the paper, we thank Vitaly Shmatikov for teaching us this material, Matt Kaufmann for helping formalize our assumption about the Diffie-Helman protocol, and Robert Krug for helping develop proofs about the key that Diffie-Helman produces. We also thank the ACL2 group and Warren A. Hunt Jr. for their general support.

# 8. REFERENCES

[1] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. *ACM Trans. Inf. Syst. Secur.*, 10(3):9, 2007.

[2] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system and compositional logic for security protocols. *J. Comput. Secur.*, 13(3):423–482, 2005.

[3] B. t. Levy. Diffie-helman method for key agreement. On the Web, 1997. http://postdiluvian.org/ seven/diffie.html.

[4] L. C. Paulson. Proving properties of security protocols by induction. In *In 10th IEEE Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, 1997.

[5] V. Shmatikov. Just fast keying. On the Web, 2004. http://www.cs.utexas.edu/users/shmat/courses/cs395t_fall04/05jfk.ppt.