IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 28, 2020, accepted January 15, 2021, date of publication January 18, 2021, date of current version January 27, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3052578

Formal Analysis of QUIC Handshake Protocol Using Symbolic Model Checking

JINGJING ZHANG[1,2], LIN YANG[2], XIANMING GAO[2], GAIGAI TANG[3], JIYONG ZHANG[4], (Member, IEEE), AND QIANG WANG[2]

[1]College of Command and Control Engineering, Army Engineering University of PLA, Nanjing 210007, China
[2]National Key Laboratory of Science and Technology on Information System Security, Institute of System Engineering, PLA Academy of Military Science, Beijing 100039, China
[3]College of Computer Science and Technology, Harbin Engineering University, Harbin 150001, China
[4]School of Automation, Hangzhou Dianzi University, Hangzhou 310018, China

Corresponding authors: Qiang Wang (wenjunwang.nudt@gmail.com) and Jiyong Zhang (jzhang@hdu.edu.cn)

This work was supported by the National Key Laboratory of Science and Technology through Information System Security.

ABSTRACT This work presents a security analysis of the QUIC handshake protocol based on symbolic model checking. As a newly proposed secure transport protocol, the purpose of QUIC is to improve the transport performance of HTTPS traffic and enable rapid deployment and evolution of transport mechanisms. QUIC is currently in the IETF standardization process and will potentially carry a significant portion of Internet traffic in the emerging future. For a better understanding of the essential security properties, we have developed a formal model of the QUIC handshake protocol and perform a comprehensive formal security analysis by using two state-of-the-art model checking tools for cryptographic protocols, i.e., ProVeirf and Verifpal. Our analysis shows that ProVerif is generally more powerful than Verifpal in terms of verifying authentication properties. Moreover, both tools produce a counterexample to some security properties, which reveal a design flaw in the current protocol specification. Last but not least, we analyze the root causes of this counterexample and suggest a possible fix.

INDEX TERMS Model checking, applied pi calculus, cryptographic protocol, QUIC, formal verification, ProVerif, Verifpal.

I. INTRODUCTION

QUIC [1] is a multiplexed and encrypted transport protocol recently suggested by Google that allows for both rapid and reliable Internet connectivity. It is designed on the top of UDP and employs an encrypted transport to prevent modifications by middleboxes. It also makes use of a cryptographic handshake protocol to protect the entities and minimize the connection latency. QUIC aims at replacing most elements of the conventional HTTPS stack (as shown in Fig.1). It operates entirely in the user-space and is currently integrated in the Chromium web explorer for a rapid configuration and exploration.

QUIC is still under review for standardization, which usually takes the format of an RFC: a natural language (normally English) document that offers implementation advice to protocol engineers. However, a natural language document is nonetheless ambiguous and open to various interpretations, some of which are even contradicting. As for the current

The associate editor coordinating the review of this manuscript and approving it for publication was Wei Huang.

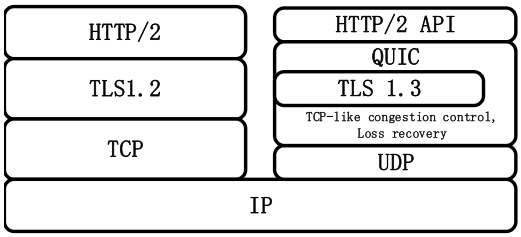

FIGURE 1. QUIC in the traditional HTTPS stack.

QUIC authentication protocol, it is still unclear whether or not it conforms to the security attributes claimed in the IETF standardization documents. As reported in [2], a possible way to resolve the ambiguities and rigorously validate the protocol design is through formal verification, where a formal model of the protocol is first constructed and then analyzed with respect to the specified security properties. Symbolic model checking [3] has been a popular method for the formal verification of cryptographic protocols. Since the pioneering work in [4] that discovered the Needham-Schroeder protocol's design flaws, symbolic model checking has been

widely and actively used to formally analyze cryptographic protocols [5]–[14].

In the symbolic model checking for cryptographic protocols, protocol behavior, as well as the intruders, are often symbolically represented by a formal model (e.g. applied pi calculus [11], [15], constraint systems [16] or horn clauses [17]), where the protocol messages are modeled by abstract terms and the cryptographic primitives (such as encryption and decryption functions) are modeled by function symbols and assumed to be uncrackable. The security analysis for the cryptographic protocols reduces to a constraint solving or unification problem for the formal model. In the event of termination, model checking can automatically discover if the protocol model meets the specified security properties. Moreover, when certain properties are breached, a counterexample can be constructed to show the violation. In the past decades, various model checking algorithms and prototype tools for cryptographic protocols have been developed. We refer to [3] for a detailed exploration of this research area.

In this work, we employ two cryptographic protocol model checking tools to analyze the formal model of the QUIC handshake protocol, namely ProVerif [18] and Verifpal [19]. They both take a formal protocol model detailed in a formal language and can automatically check whether this formal model satisfies the security properties in the presence of a hostile adversary. The major advantages are as follows. Firstly, they provide a robust modeling mechanism that enables the description of a large variety of cryptographic primitives through the use of rewrite rules and equation theory. They also support the specification of a variety of security properties, including both strong and weak confidentiality, authentication and observational equivalence properties. They can also deal with an unlimited number of parallel protocol implementations, which is critical for detecting subtle attacks, such as man-in-the-middle attacks. Finally, they can automatically construct counterexamples, when certain properties are violated.

These two model checkers differ in the way how cryptographic protocols are formally modeled. ProVerif uses applied pi calculus as the formal modeling language and it expects that the users are able to reason about the protocol logic as a calculus process. In contrast, Verifpal offers a more natural way to model the protocols, as a sequence of messages exchanges between the explicit principals. Verifpal's inner logic continues to rely on the deconstruction and reconstruction of abstract terms, similar to ProVerif. However, Verifpal's modeling language is more close to the normal way how engineers think about cryptographic protocols, while still being accurate and expressiveness enough. We would like to compare both the modelling and the verification capabilities of these two verifiers (i.e., Verifpal and ProVerif), to understand their strengthens and weaknesses.

The main contributions we have made in this work are as follows:

1) We construct from the natural language documents the first formal model of the QUIC handshake protocol in applied pi calculus. We also derive a set of security properties from the informal standardization documents and specify them in formal languages. We would like to point out that the formal modelling of protocols and security properties is non-trivial, because it demands a profound understanding of the protocol logic and the possible attack behaviors.
2) We also construct a formal model of the QUIC handshake protocol in Verifpal and compare it with the ProVerif's model to illustrate the characteristics of these two modeling languages.
3) We carry out a comparative formal analysis of the QUIC handshake protocol using two state-of-the-art verifiers: the ProVerif and Verifpal. Our analysis shows that ProVerif is generally more powerful than Verifpal in terms of verifying authentication properties. Moreover, both verifiers reveal a design flaw in the current protocol specification. We also discuss the possible causes and suggest potential fixes.

The organization of this paper is as follows. Section II gives the most related works on formal analysis of cryptography protocols. Section III presents a thorough description of the QUIC handshake protocol and the set of security properties. Section IV presents the formal protocol model as well as the encodings of the security properties. Section V reports the verification results. Finally, Section VI concludes this paper.

II. RELATED WORKS

Previously, symbolic model checking has been used successfully to analyze the security properties of cryptographic protocols. In this section, we review the most relevant works.

In [12] and [13], the authors use Tamarin [20] to model and analyze the security properties of the 5G AKA protocol. They find that the 5G AKA protocol lacks integrity protection for the identity of the server network. Furthermore, the authors report in [13] an attack that takes advantage of potential competition conditions.

In [21], the authors develop a complete symbolic model of TLS 1.3 specification that takes into account all possible interactions of the available handshake patterns. They analyze most of the specified security requirements using the Tamarin verifier and find a behavior that could cause security problems in protocol applications.

In [22], the authors present a novel modeling framework that accounts for all recent attacks on TLS, including those relying on weak cryptography. They apply ProVerif to perform the first symbolic analysis of the Draft-18 of TLS 1.3 standard and the first composite analysis of TLS 1.3 and TLS 1.2. Their analyses uncover both known and new vulnerabilities that influence the final design of Draft-18.

In [23], the authors use ProVerif and CryptoVerif [24] to perform formal analysis of the variants of signal protocols against a series of security objectives. They also implement

the signal protocol in ProScript [23], a new domain-specific language for writing code for cryptographic protocols. The implementation in ProScript can be executed in a JavaScript program or automatically translated into a readable model in applied pi calculus. Their analysis reveals several weaknesses in the protocol, including previously unreported replay and key leak impersonation attacks.

In [25], the authors develop a formal specification for wireless protocols and use this specification to generate an automatic randomization test tool. Then they use this tool to determine whether the implementation of the QUIC protocol satisfies the formal specification they have developed.

In [19], the author who is developing Verifpal, provides the first formal model of the DP-3T decentralized pandemic tracing protocol [26] and analyzes unlinkability, freshness, confidentiality and message authentication. The author also claims that Verifpal has been proven to successfully verify the security properties for TLS 1.3 and Signal.

In [27], the authors propose a novel proxy-based design for QUIC, called QSOCKS, that can enhance the capabilities of QUIC. QSOCKS improves the 0-RTT/1-RTT connection time, enhances the handshake mechanism and transmission security.

In [28], the authors have applied state machine inference to analyze the QUIC implementation. This method works in a fully black-boxed way, which makes it applicable to any QUIC implementation without the implementation specifics. They also give the correctness proof of the QUIC implementation of Google.

In [29], the authors find the denial-of-service attack of the QUIC protocol and establish a mathematical model based on the finite state machine to explain the attack principle. They propose feasible suggestions for preventing such attacks.

In [30], the authors present a new formal security model for the QUIC protocol. They have found that the QUIC protocol may not enable 0-RTT (0 round trip time) connections. An attacker can make QUIC fall-back to TCP/TLS or lead to client and server handshake views that are inconsistent, resulting in an inconsistent state and more latency.

Finally, we remark that this work is based on our previous work [31], where we model and formally analyze the QUIC handshake protocol in ProVerif. We have extended our previous work by carrying out a comparative formal analysis of the QUIC handshake protocol in two state-of-the-art model checkers, that is ProVerif and Verifpal. First, we build two complementary formal models of the QUIC handshake protocol in ProVerif and Verifpal. ProVerif provides a general modeling language for security protocols based on process calculus, which requires a deep understanding of the formal semantics of the protocol. Verifpal offers a more intuitive language instead. We elicit the essential security properties from the informal description in the IETF document and encode them in both ProVerif's model and Verifpal's model. We also discuss the major differences between ProVerif and Verifpal in both protocol modeling and security property specification. Second, we compare the formal analysis results and performances of the two model checkers. The results show that ProVerif is in general more efficient than Verifpal, while both of them are consistent in revealing design flaws of the QUIC handshake protocol.

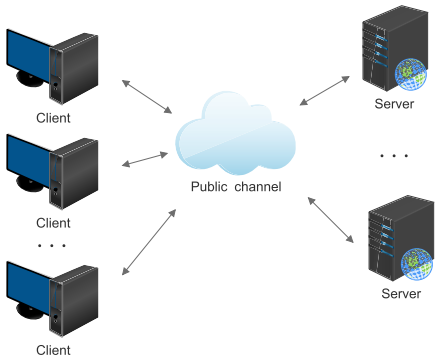

FIGURE 2. A reference architecture model of the QUIC network.

III. THE QUIC PROTOCOL

A. INTRODUCTION TO QUIC PROTOCOL

In this section, we present a detailed review of the QUIC protocol described in the IETF document [32].

The typical architecture of the QUIC network is shown in Fig. 2. The client represents a subscriber's device (e.g., mobile phone or computer) that intends to start a secure connection to the network. The server is where the client may connect to obtain a service. We assume that the public channel through which the client communicates with the server is under the control of malicious attackers.

QUIC performs encryption in the transport layer during the handshake process, reducing the number of round trips required for setting up a secure connection. QUIC initial connections are common 1-RTT, meaning that all initial connection data can be sent immediately without waiting for a reply from the server, which is more efficient compared to the 3 round trips required for TCP/TLS before application data can be sent. The timeline of QUIC's initial 1-RTT handshake and TCP/TLS's initial 3-RTT handshake are shown in Fig.3. QUIC is functionally equivalent to TCP/TLS/HTTP2, but implemented on top of UDP. The key advantages over TCP/TLS/HTTP2 include:

- Low connection establishment latency
- Flexible congestion control
- Multiplexing without head-of-line blocking
- Authenticated and encrypted payload
- Stream and connection flow control
- Connection migration

QUIC uses encrypted transport handshake to establish a secure transport connection. After a successful handshake, the client caches the server information including the hostname and the port number. In subsequent connections with the same server, the client can establish an encrypted connection directly with the server without any additional handshakes. Encrypted packets from the client can be sent immediately (i.e., 0-RTT).

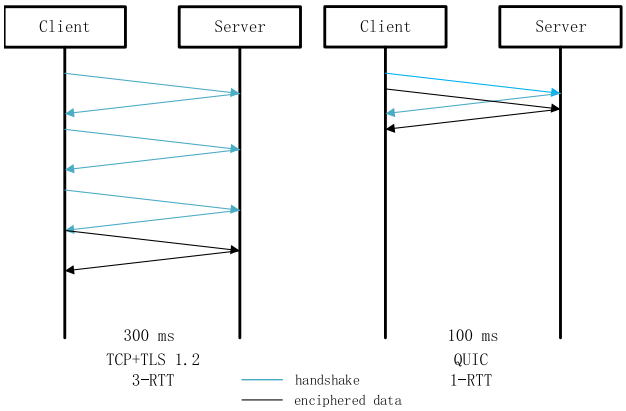

FIGURE 3. Comparison of the QUIC initial 1-RTT handshake process and the TCP/TLS initial 3-RTT handshake process.

In order to perform the 0-RTT handshake, the client needs to obtain verified configuration and authentication information from the server. Initially, we assume that the client contains no server information. Before attempting the first handshake, the client sends a hello message to elicit the configuration and proof of authenticity from the server. There could be several rounds of client hellos before the client receives all the information that it needs because the server may refuse to send entire proof of authenticity to an unvalidated IP address.

Upon a client hello message, the server either sends a rejection message, or a server hello. The server hello indicates a successful handshake and can never result from an inchoate client hello as it doesn't contain enough information to perform a handshake. Rejection messages contain information that the client can use to perform subsequent better handshake attempts.

The flow of the QUIC handshake process as defined in the IETF document [32] is shown in Fig.4.

- Since the client does not cache the server configuration information in the beginning, the client needs to send a hello message (CHLO) to the server to get the reject message (REJ) from the server.
- When the server receives a CHLO message, it sends the REJ message to the client. The REJ message contains the following parts: (1) the config information including the server's long-term Diffie-Hellman public value, (2) a certificate chain authenticating the server, (3) a signature of the server config using the private key from the leaf certificate of the chain, and (4) a source address token (as an authenticated encryption block).
- When the client receives the server's configuration information, it validates the configuration information with the server's certificate and signature, and calculates its initial secret key using the server's long-term Diffie-Hellman public value and its own ephemeral Diffie-Hellman private key. It then sends the complete CHLO message to the server, including its ephemeral Diffie-Hellman public value.
- If the handshake is successful, the server calculates the initial keys using the client's ephemeral Diffie-Hellman public value and its long-term Diffie-Hellman private value. Moreover, it calculates its own ephemeral Diffie-Hellman public value using its ephemeral Diffie-Hellman private value and calculates its final keys using the client's ephemeral Diffie-Hellman public value and its ephemeral Diffie-Hellman private value. It sends its ephemeral Diffie-Hellman public value encrypted with the initial secret key as a server hello message (SHLO) to the client. Finally, the server encrypts subsequent communication data using its forward-secure key.
- When the client receives the SHLO message, it sends the packet encrypted with its forward-secure key.

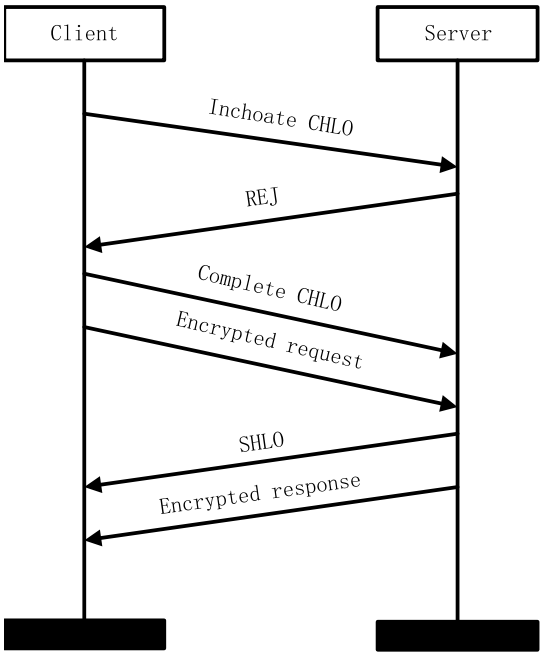

FIGURE 4. Initial 1-RTT handshake process of QUIC.

The detailed steps of the QUIC handshake process are depicted in Fig.5 and Table. 1.

B. REQUIRED SECURITY PROPERTIES

The IETF document [32], [33] provides informal descriptions of the security requirements for the QUIC protocol. We extract the descriptions that directly affect the QUIC security properties from the informal specifications.

Fig.6 shows the description of the authentication property of the QUIC handshake protocol [32], [33]. The client must be able to authenticate the identity of the server and the server should have the ability to authenticate the client. In other words, the QUIC handshake protocol guarantees that the client and the server can authenticate each other. Therefore, authentication can be thought of as the client and server agreeing on each other's identity and security key. We summarize the authentication properties of the QUIC handshake protocol as follows:

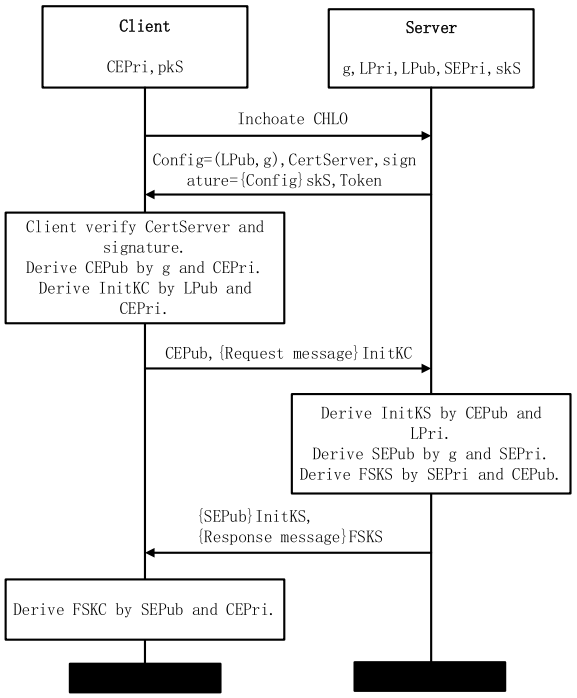

FIGURE 5. The detailed steps of the QUIC handshake process.

TABLE 1. Protocol messages and their annotations.

Message	Annotation
CEPri	Client's ephemeral Diffie-Hellman private value.
LPri	Server's long-term DH private value.
SEPri	Server's ephemeral Diffie-Hellman private value.
LPub	Server's long-term DH public value.
g	Primitive Root.
CEPub	Client's ephemeral Diffie-Hellman public value.
SEPub	Server's ephemeral Diffie-Hellman public value.
InitKC	Initial key of Client.
InitKS	Initial key of Server.
FSKC	Forward-secure key of Client.
FSKS	Forward-secure key of Server.
pkS	Public signature key of the server.
skS	Private signature key of the server.
{·}skS	{·} is signed using the private signature key of the server.
{·}InitKC	{·} is encrypted using the initial key of client.
{·}InitKS	{·} is encrypted using the initial key of server.
{·}FSKS	{·} is encrypted using the forward-secure key of server.

A1 Both the client and the server should agree on the correspondence after successful termination.
A2 Both the client and the server should agree on the request message *ReqM* after successful termination.
A3 Both the client and the server should agree on the response message *ResM* after successful termination.

For the confidentiality requirement of the QUIC handshake protocol, we extract and show the most relevant description in Fig.7. We also assume that the client and the server have the ability to verify each other's certificates. The document [32] specifies the confidentiality of the forward key ($F1$) and the session messages ($S1, S2$). We interpret these requirements as the following secrecy properties.

Authentication
The QUIC handshake shall satisfy the following requirements.
Client authentication: A client MUST authenticate the identity of the server.
Serving network authentication: A server MAY request that the client authenticate during the handshake. A server MAY refuse a connection if the client is unable to authenticate when requested. The requirements for client authentication vary based on application protocol and deployment.

FIGURE 6. Definition of authentication (from [33] p.16).

Confidentiality
The following security features are provided with respect to confidentiality of data:
Cipher key agreement:: The TLS authenticated key exchange occurs between two endpoints: client and server. The client initiates the exchange and the server responds. If the key exchange completes successfully, both client and server will agree on a secret. TLS supports both pre-shared key (PSK) and Diffie-Hellman over either finite fields or elliptic curves((EC)DHE) key exchanges. PSK is the basis for 0-RTT; the latter provides perfect forward secrecy (PFS) when the (EC)DHE keys are destroyed.
Confidentiality of data: All QUIC packets except Version Negotiation packets use authenticated encryption with additional data to provide confidentiality and integrity protection; Initial packets are protected using keys that are statically derived. This packet protection is not effective confidentiality protection. Initial protection only exists to ensure that the sender of the packet is on the network path. Any entity that receives the Initial packet from a client can recover the keys necessary to remove packet protection or to generate packets that will be successfully authenticated; Packets protected with 0-RTT and 1-RTT keys are expected to have confidentiality and data origin authentication; the cryptographic handshake ensures that only the communicating endpoints receive the corresponding keys.

FIGURE 7. Definition of confidentiality (from [32] p.69).

F1 The forward key *FSKC* or *FSKS* that is captured in the current session by the attacher cannot affect the confidentiality of the other sessions.
S1 The adversary must not be able to obtain the request message *ReqM*.
S2 The adversary must not be able to obtain the response message *ResM*.

IV. FORMAL MODEL OF THE QUIC HANDSHAKE PROTOCOL

In this section, we report the formal model of the QUIC handshake protocol.

A. ATTACKER MODEL AND THE PERFECT CRYPTOGRAPHY ASSUMPTION

In our work, we assume that the participants in the agreement are honest. In other words, the client and server strictly follow the specification of the protocol. We also consider the existence of an attacker who has complete control over the message on the public channel. In other words, the attacker can intercept, tamper and replay the information on the public channel. However, the attacker cannot decrypt the ciphertext without knowing the correct secret key. This attacker model is called the Dolev-Yao model [34].

An important feature of the symbolic model is the assumption of perfect cryptography. We represent symmetric encryption with a binary constructor *senc*(). This constructor takes two arguments, one is of type bitstring and another is of type key. Its return value is the ciphertext of type bitstring. Similarly, the decryption function is represented by a destructor *sdec*(). The relationship between the encryption function and the decryption function can be expressed by the formula $sdec(senc(m, k), k) = m$. The formula shows that the encryption function $senc(m, k)$ can only get the message m by decrypting it with the same secret key k.

```
fun senc(bitstring,key):bitstring.
reduc forall m:bitstring,k:key; sdec(senc(m,k),k) = m.
```

We define a digital signature as a constructor function *sign*(), which relies on a pair of asymmetric keys. We define the private key as *skey* and the public key as *pk*. As a function to verify the signature, we represent it with a destructor *checksign*().

```
fun pk(skey):pkey.
fun sign(G, skey): bitstring.
reduc forall m: G, k: skey; getmess(sign(m,k)) = m.
reduc forall m: G, k: skey; checksign(sign(m,k), pk(k)) = m.
```

The Diffie-Hellman key agreement algorithm is modeled as follows:

```
type G.
type exponent.
const g:G.
fun exp(G,exponent):G.
equation forall x: exponent, y: exponent;
exp(exp(g,x),y)=exp(exp(g,y),x).
```

Where g is the generator, and exp models modular exponentiation $exp(x, y) = x^y$. The equation means that $(g^x)^y = (g^y)^x$.

The public channel between client and server is modeled *c*.

```
free c:channel.
```

B. QUIC HANDSHAKE PROTOCOL IN APPLIED PI CALCULUS

1) SYMBOLIC PROTOCOL MODEL

Client process is shown in Fig. 8. First, the client sends an *InchoateCHLO* message on the channel *c*. We remark that an attacker can access this public channel. Then it waits for the message including five variables bounded to variables $x1$, $x2$, $x3$, $x4$ and pkX respectively. After that, the client checks if the variables $x3$ and pkX are respectively certificate and public key pkS of the server. Then the client checks the signature of $x2$ using variable pkX, which is the public key belonging to the server. If the result of the signature check is equivalent to $x1$, then the client calculates its initial key *InitKC* using $x1$ and its ephemeral private value *CEPri*. Subsequently, it sends the ephemeral public value *CEPub* ($exp(g, CEPri)$) and the ciphertext of the request message ($enc(ReqM, InitKC)$) on the channel. Then it waits for a message of form ($x5, x6$). When the client captures the message, it decrypts the variable $x5$ using its initial key *InitKC*, and then bounds the return to variable $x7$. Normally, the variable $x7$ should be the server's ephemeral public value. Finally, the client calculates its forward-secure key *FSKC* using variable $x7$ and its ephemeral private value *CEPri*, and decrypts the ciphertext $x6$ using the forward-secure key *FSKC*. In order to specify authentication, we add specific events to the protocol model, which we will explain later.

```
let Client(pkS:pkey, InitKC:G) =
        out(c, CHLO);
        in(c, (x1:G, x2:bitstring, x3:bitstring,
        x4:bitstring, pkX:pkey));
        if x3 = CertServer then
        if pkX = pkS then
        let (=x1) = checksign(x2, pkX) in
        new CEPri: exponent;
        let InitKC = exp(x1, CEPri) in
        new ReqM:bitstring;
        event InitC(CHLO);
        event sendReqM(ReqM);
        out(c, (exp(g, CEPri), enc(ReqM, InitKC)));
        in(c, (x5:G, x6:bitstring));
        let x7 = decP(x5, InitKC) in
        let FSKC = exp(x7, CEPri) in
        let ResMx = dec(x6, FSKC) in
        event acceptResM(ResMx);
        event EndC(x1).
```

FIGURE 8. The client process in applied pi calculus.

Server process is shown in Fig. 9. First, the server bounds the message of its input to variable $x1$. It checks whether $x1$ is the legal inchoate client hello (*InchoateCHLO*). Then the server sends its certificate (*CertServer*), the long-term public value *LPub* ($exp(g, LPri)$), the signature of *LPub*, *Token* and its public key *pkS* on the channel *c*. Next, the server waits for the message including two parts which are respectively bounded to $x2$ and $x3$. It obtains its initial key *InitKS* using $x2$ and the long-term private value *LPri*. The server checks if the ciphertext $x3$ is the request message using *InitKS*. Then the server calculates the *SEPub* with *SEPri* and obtains its forward-secure key *FSKS* with $x2$ and *SEPri*. Finally, it sends the ciphertext of *SEPub* and response message (*ResM*) on the channel *c*.

The protocol process is shown in Fig. 10. First, the process declares the initial secret keys of the protocol, such as *InitKC* and *InitKS*, and then generates the privates key *skS* and sends

```
let Server(skS:skey, pkS:pkey, InitKS:G) =
        in(c, x1:bitstring);
        if x1 = CHLO then
        new LPri: exponent;
        new Token: bitstring;
        out(c, (exp(g, LPri), sign(exp(g, LPri), skS),
        CertServer, Token, pkS));
        in(c, (x2:G, x3:bitstring));
        let InitKS = exp(x2, LPri) in
        let ReqMx = dec(x3, InitKS) in
        event acceptReqM(ReqMx);
        new SEPri: exponent;
        let SEPub = exp(g, SEPri) in
        let FSKS = exp(x2, SEPri) in
        new ResM:bitstring;
        event InitS(exp(g, LPri));
        event sendResM(ResM);
        out(c, (encP(SEPub, InitKS), enc(ResM, FSKS)
        ));
        event EndS(x1).
        phase 1; out(c, FSKS).
```

FIGURE 9. The server process in applied pi calculus.

```
process
        new InitKC: G;
        new InitKS: G;
        new skS:skey;
        let pkS = pk(skS) in
        out(c, pkS);
        ( (!Client(pkS, InitKC)) |
        (!Server(skS, pkS, InitKS)))
```

FIGURE 10. The protocol process in applied pi calculus.

the corresponding public keys on the public channel c. Then, it declares that both the client process and the server process are the parallel composition of infinite replications.

2) SECURITY PROPERTY SPECIFICATION

In ProVerif, a fact is modeled as a ground term. To prove the secrecy of a term M, ProVerif essentially solves a reachability problem, i.e., whether the attacker can reach a state where the term M is available. In this work, ProVerif takes the following queries in the protocol model to check the secrecy of the request message *ReqM* and the response message *ResM*.

$$query\ \ attacker(new\ ReqM)$$
$$query\ \ attacker(new\ ResM)$$

ProVerif is able to verify the forward confidentiality of the protocol by providing a temporal branch called *phase*. For example, we use the following queries to specify the scenario: when the secret keys of a protocol participant are leaked to the attacker at phase 1, the attacker cannot use the obtained secret keys to decrypt messages at phase 0.

$$query\ \ attacker(new\ FSKC)\ \ phase\ \ 0$$
$$query\ \ attacker(new\ FSKS)\ \ phase\ \ 0$$
$$phase\ \ 1;\ \ out(c,\ \ FSKC)$$
$$phase\ \ 1;\ \ out(c,\ \ FSKS)$$

Authentication properties are captured by correspondence assertions, which can express the relationships between events in the form "if some event has been executed in the protocol, then some other event has been previously executed." In ProVerif, events are of the form $event\ \ e(M_1, \ldots, M_n)$ and the query of a correspondence assertion is

$$query\ \ x_1 : t_1, \ldots, x_n : t_n;\ \ event(e(M_1, \ldots, M_j)) \Rightarrow event(e'(N_1, \ldots, N_k))$$

where terms $M_1, \ldots, M_j$, $N_1, \ldots, N_k$ are built by applying constructors to variables $x_1, \ldots, x_n$. The query is satisfied if, for each occurrence of event $e(M_1, \ldots, M_j)$, there is a previous execution of event $e'(N_1, \ldots, N_k)$. There is also a stronger variant of correspondence assertion, where can capture the one-to-one relationship between events. They are often called injective correspondence assertions, which take the form:

$$query\ \ x_1 : t_1, \ldots, x_n : t_n;\ \ inj-event(e(M_1, \ldots, M_j)) \Rightarrow inj-event(e'(N_1, \ldots, N_k))$$

Informally, this correspondence asserts that, for each occurrence of the event $e(M_1, \ldots, M_j)$, there is a distinct earlier occurrence of the event $e'(N_1, \ldots, N_k)$. This differs from the previous correspondence assertions in that no single event $e'(N_1, \ldots, N_k)$ can map to two more events $e_1(M_1, \ldots, M_j), e_2(M'_1, \ldots, M'_j)$.

In this work, we declare the following events in the formal model:

- $event\ \ InitC(x)$, indicating the client believes that she has accepted to run the protocol with the server and the supplied parameter;
- $event\ \ InitS(x)$, indicating the server believes that she has accepted to run the protocol with the client and the supplied parameter;
- $event\ \ EndC(x)$, indicating the client believes that she has terminated a protocol run using the given parameter;
- $event\ \ EndS(x)$, indicating the server believes that she has terminated a protocol run using the given parameter;
- $event\ \ sendReqM(x)$, indicating the client believes that she has transmitted the request message to the server given as the parameter;
- $event\ \ acceptReqM(x)$, indicating the server believes that she has accepted the request message from the client given as the parameter.
- $event\ \ sendResM(x)$, indicating the server believes that she has transmitted the response message to the client given as the parameter;
- $event\ \ acceptResM(x)$, indicating the client believes that she has accepted the response message from the server given as the parameter.

Therefore, we consider the following correspondence assertions to prove authentication properties.

- $query\ x : bitstring; inj-event(acceptReqM(x)) \Rightarrow inj-event(sendReqM(x))$.
- $query\ x : bitstring; inj-event(acceptResM(x)) \Rightarrow inj-event(sendResM(x))$.

```
principal client[
        generates CHLO
        knows public c0
]
client -> server: [CHLO]
principal server[
        generates CertServer
        generates Token, LPri
        knows public c0
        knows private skS
        pkS = G^skS
        LPub = G^LPri
        ssign = SIGN(skS, LPub)
]
server -> client: [CertServer], Token, LPub, ssign, [pkS]
principal client[
        generates CEPri
        generates ReqM
        _ = SIGNVERIF(pkS, LPub, ssign)?
        CEPub = G^CEPri
        InitKC = LPub^CEPri
        e_ReqM = AEAD_ENC(InitKC, ReqM, c0)
]
client -> server: CEPub, e_ReqM
principal server[
        generates SEPri
        generates ResM
        InitKS = CEPub^LPri
        SEPub = G^SEPri
        FSKS = CEPub^SEPri
        e_ReqMx = AEAD_DEC(InitKS, e_ReqM, c0)?
        e_ResM = AEAD_ENC(FSKS, ResM, c0)
        e_SEPub = AEAD_ENC(InitKS, SEPub, c0)
]
server -> client: e_ResM, e_SEPub
principal client[
        SEPubs = AEAD_DEC(InitKC, e_SEPub, c0)?
        FSKC = SEPubs^CEPri
        _ = AEAD_DEC(FSKC, e_ResM, c0)?
]
```

FIGURE 11. The Verifpal model of the QUIC handshake protocol.

- $query\ x\ :\ bitstring; inj-event(EndS(x)) \Rightarrow inj-event(InitC(x))$.
- $query\ x\ :\ G; inj-event(EndC(x)) \Rightarrow inj-event(InitS(x))$.

C. QUIC HANDSHAKE PROTOCOL IN VERIFPAL

1) SYMBOLIC PROTOCOL MODEL

Verifpal also follows the perfect cryptography assumption. It offers a rich set of primitives to capture cryptographic functions. In the following, we show the declaration of symmetric encryption and decryption in Verifpal.

```
ENC(key, plaintext): ciphertex.
DEC(key, ENC(key, plaintext)): plaintext.
```

Verifpal supports both classic signature primitive and authenticated signature with a corresponding signature verification function as shown in the following.

```
SIGN(key, message): signature.
SIGNVERIF(G^key, message, SIGN(key, message)): message.
```

The Diffie-Hellman arithmetic is declared as follows, where G is the root and gxy and gyx are considered equivalent according to the property of discreate logarithm.

```
generates x
generates y
gx = G^x
gy = G^y
gxy = gx^y
gyx = gy^x
```

Verifpal considers both passive and active attackers. In the passive mode, the attacker can only obtain the messages on the public channel but cannot tamper or inject the messages. For the active model, the attacker is able to modify messages in a certain manner and inject the new messages into the protocol executions. We use the same parameters as in IV-B and present the QUIC handshake protocol model in Verifpal as shown in Fig. 11. Unlike ProVerif, Verifpal models the protocol in a way that is closer to the flowchart of the protocol interaction. Therefore, we can directly map the QUIC handshake protocol model in Verifpal to the protocol steps in Fig.5 and the revised protocol model is shown in Fig.12.

Verifpal uses *principal* to represent the actions of each protocol participant. The transmission of messages is described in the following way:

$$RoleA- > RoleB : message1, [message2].$$

This expression means that the protocol role *RoleA* sends messages *message*1 and *message*2 to another role *RoleB*. Notice that, *message*2 is surrounded by brackets []. This makes the *message*2 a guarded constant, meaning that while an active attacker can still read it, they cannot tamper with it. This approach satisfies the protection of consensus messages. We use it to ensure that an honest participant agrees that the message is not compromised by an attacker. In our work, we add the bracket for the message *CHLO*, the server's certification *CertServer* and the server's public key *pkS*. The reason is that *CHLO* is the specific message at the beginning of the protocol and this message should be consensual by both the client and the server. Also, we assume that the client has obtained *CertServer* and *pkS* before the protocol execution.

In the beginning, we define a variable *CHLO* as the client hello message through the keyword *generates* in the first principal. This keyword allows a principal to describe a *fresh* value, i.e. a value that is regenerated every time the protocol is executed. Verifpal provides another keyword *knows* to define constants. In this principal, we define a public constant $c0$ with the keyword *knows*. This constant $c0$ represents an additional payload that is not encrypted, but provided exactly in the decryption function for a successful decryption. We use $client- > server : [CHLO]$ to indicate that the client sends the message *CHLO* to the server.

In the next principal, we define the server's certificate (*CertServer*), token (*Token*), and long-term private value (*LPri*) as three variables. In addition to defining the constant $c0$, we also define a server's private key *skS*. Through equation $pkS = G^{\wedge} skS$, we bind the server's public key *pk*

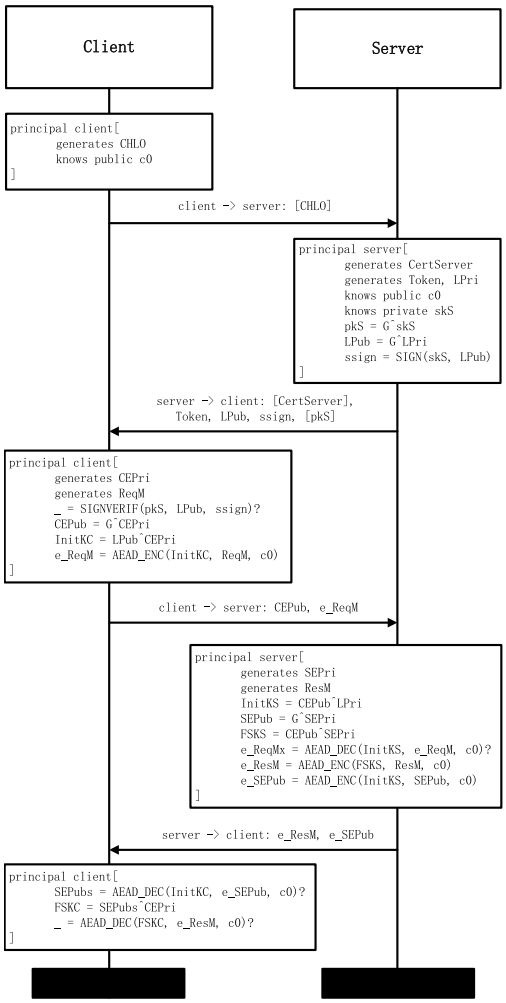

FIGURE 12. The detailed steps of the QUIC handshake protocol in Verifpal.

to its private key *skS*. Similar to the asymmetric secret key, we bind the server's long-term private value to its long-term public value using $LPub = G \hat{} LPri$. For the signature of the server's config, we use the specific constructor *sign*() to sign the long-term public value of the server with the server's private key *skS*, and bind the return to the value *ssign*. We use $server- > client : [CertServer], Token, LPub, ssign, [pkS]$ to indicate that the server sends messages to the client.

As the next step in the protocol execution, the client can verify the received signature by using the public key *pkS* of the server, which is defined as $\_ = SIGNVERIF(pkS, LPub, ssign)?$. This definition means that the execution will abort when *SIGNVERIF* fails to verify the signature *ssign* against the provided message *LPub* and the public key *pkS*. The behavior of the client calculating the initial secret key *InitKC* is defined as $InitKC = LPub \hat{} CEPri$. We use $e\_ReqM = AEAD\_ENC(InitKC, ReqM, c0)$ to express the symmetric encryption where *e_ReqM* represents the value of the ciphertext, *InitKC* represents the encryption key, *ReqM* represents the encrypted message and *c*0 is the public constant of the encryption. We use $client- > server : CEPub, e\_ReqM$ to indicate the sending from the client.

In the next principal, the server calculates the initial key *InitKS* and the forward-secure key *FSKS*. We use $e\_ReqMx = AEAD\_DEC(InitKS, e\_ReqM, c0)?$ to express the symmetric decryption where *e_ReqMx* is the plaintext obtained after decryption and *InitKS* is the decryption key. The server then sends the ciphertext *e_ReqM* and *e_SEPub* to the client: $server- > client : e\_ResM, e\_SEPub$.

In the last principal, the client decrypts the ciphertext *e_SEPub* and calculates its forward-secure key *FSKC*. The expression $\_ = AEAD\_DEC(FSKC, e\_ResM, c0)?$ means whether the client can correctly decrypt *e_ResM* by using *FSKC* and *c*0.

2) SECURITY PROPERTY SPECIFICATION

Verifpal takes a different approach to specify the security properties: while is being analyzed the model, the outputs note which values can deconstruct, conceive of, or reconstruct. When a contradiction is found for a query, the result is related in a readable format that ties the attack to a real-world scenario. This is done by using some terminology to indicate how the attack could have been possible, such as through a man-in-the-middle attack on ephemeral keys [19].

To prove the secrecy of a message, Verifpal provides *confidentiality queries* check:

$$confidentiality?\ ResM$$
$$confidentiality?\ ReqM$$

The above check asks: "can the attacker obtain the request message *ReqM* and the response message *ReqM*?", where *ReqM* and *ResM* are the sensitive messages. If the attacker satisfies the above query, it can obtain the messages.

Similar to ProVerif, Verifpal uses *phases* to simulate the leaking of secret keys at the different temporal branch. Therefore, it is also able to verify the forward confidentiality of the protocol participant's secret key. For example, Verifpal can leak the principal's secret key *FSKC* and *FSKS* to the attacker in phase 1, the attacker cannot use the obtained secret keys to decrypt messages in phase 0.

$$phase[1]$$
$$principal\ client\ [leaks\ FSKC]$$
$$principal\ server\ [leaks\ FSKS]$$

In Verifpal, the authentication queries are more complex than the confidential queries. For instance, the following queries ask: "if the client (or server) successfully decrypts and authenticates the ciphertext of *ResM* (or *ReqM*), does that necessarily mean that the server (or client) sent the ciphertext *e_ResM* (or *e_ReqM*) to the client (or server)?".

$$authentication?client- > server : e\_ReqM$$
$$authentication?server- > client : e\_ResM$$

This means that if an attacker can induce the client (or server) to decrypt the ciphertext *e_ResM* (or *e_ReqM*), then the attacker has successfully impersonated the server (or client).

3) COMPARISON OF THE TWO MODELS

The two models differ from each other in both protocol modeling and property specification. ProVerif offers the complex message types as well as user-defined types, which however are not declared in Verifpal. By forbidding user-defined types, Verifpal avoids the modeling uncertainty associated with the message types.

ProVerif defines a channel in the form of *free c* : *channel*[*private*], where c is the name of a private channel. When the keyword private is not added, a public channel is declared instead. In Proverif, communication is captured by input and output events. The input event $in(m, x : t)$ means that a message of type t from the channel m is expected and bounded to variable x when received. Similarly, the output event $out(m, n)$ sends the message m on the channel n. For instance, in the QUIC handshake protocol, event $out(c, CHLO)$ indicates that the message CHLO is sending on the public channel c and at the same time event $in(c, x1 : bitstring)$ indicates a message of type bitstring is receiving on the channel c.

Verifpal models communication in a much simpler and straightforward manner. It does not declare any channels but represents the message passing in the form of *client* $->$ *server* : *message*. For instance, the same passing message event above is modeled by *client* $->$ *server* : *CHLO*. The drawback however is that the lack of channel descriptions makes it difficult to consider private channels.

ProVerif describes each role of the protocol as a complete process, whereas Verifpal is different from that it describes the behavior of the protocol roles in the message-passing order, which is similar to the execution flow of the protocol and makes the model easier to understand.

In modeling message confidentiality, ProVerif and Verifpal both use a similar approach. They provide a confidentiality query to determine whether an attacker has access to the sensitive messages. For forward confidentiality, they both use the concept of *phases*, which enables the protocol to validate forward confidentiality of messages in multiple sessions. In terms of authentication, ProVerif uses *events* in the protocol process model to determine whether authentication is satisfied. In contrast, the Verifpal model is relatively concise and its determination of authentication is similar to confidentiality why does not require adding additional expressions artificially in the protocol process.

V. RESULTS

We apply ProVerif 2.00 and Verifpal 0.18.1 respectively to verify the QUIC handshake protocol model and give the experimental results in Table 2. The secrecy properties $S1, F1$ and the agreement property $A3$ are satisfied, while the agreement properties $A1, A2$ and the secrecy property $S2$ are violated. The violation of properties $A1$ and $A2$ means that both the client and the server cannot agree on the correspondence and the request message *ReqM* after successful terminations. While the violation of property $S2$ means that the adversary is able to obtain the request message *ReqM*. The reason for these violations is that the protocol designer ignores the protection of *CEPub*. Although the transmission of *CEPub* in plaintext does not give an attacker access to the session secret key, an attacker can easily disguise himself as some legitimate client to complete the protocol handshake with the server. In other words, it means that the server does not have the ability to authenticate the client.

In terms of efficiency, ProVerif outperforms Verifpal in almost all experiments. We give a brief explanation in the following. ProVerif represents the protocol in an extension of the applied pi calculus. This representation is then translated into an abstract representation by Horn clauses, which is used to prove the desired correspondence. Therefore, the protocol can be verified by this method is less than 1s. Notice that ProVerif can verify complex protocol models, such as the 5G EAP-TLS protocol [14], in seconds. Verifpal separates protocol analysis into five stages in which it gradually allows itself to modify more and more elements of principals' states [19]. Verifpal validates each property from the first stage. Some properties may get the result in the first stage analysis, while others may need to analyze all five stages to get the result. Therefore, Verifpal probably takes more time on the protocol analysis than ProVerif. In fact, this few seconds gap indeed indicate a large efficiency difference between the two verifiers in formal verification. This gap will be even more obvious in complex models.

As for the correctness of our formal models, we take two complementary verifiers to perform the modelling and verification. In particular, the tool Verifpal provides a straightforward and intuitive modelling language, that fits well with the protocol interaction flow. And, the protocol interaction flow is obtained directly from the protocol standardization document. Thus, we believe that the formal protocol model is faithful for analysis. Moreover, both verifiers (i.e., Verifpal and ProVeirf) produce the same counterexample, which cannot happen if one of the formal model is incorrect. We also look at the counterexample closely and analyze its feasibility, which gives us confidence that this counterexample is due to some design flaws.

A. COUNTER EXAMPLE AND A POSSIBLE FIX

ProVerif and Verifpal can generate the counterexamples for the violated properties. We do not report their complex output in this article. We discuss the defects and fix of the QUIC handshake protocol later. In Fig. 13, we represent the attacker's behavior and messages in red. We expose the models and results of this article in the Github repository.[1]

The QUIC handshake protocol uses the Diffie-Hellman algorithm twice to calculate the initial keys and the forward-secure keys. The client transmits the *CEPub* to the

[1] https://github.com/bxk2008/model-QUIC

TABLE 2. Comparison of the verification results by Proverif and Verifpal.

Security properties (A:Agreement; S:Secrecy; F:Forward secrecy)	ProVerif		Verifpal	
	Result	Time	Result	Time
A1. Both the client and the server should agree on the correspondence after successful termination.	False	<1s	False	7s
A2. Both the client and the server should agree on the request message $ReqM$ after successful termination.	False	<1s	False	1s
A3. Both the client and the server should agree on the response message $ResM$ after successful termination.	True	<1s	True	7s
S1. The adversary must not be able to obtain the request message $ReqM$.	True	<1s	True	5s
S2. The adversary must not be able to obtain the response message $ResM$.	False	1s	False	<1s
F1. The forward key $FSKC$ or $FSKS$ that is captured in the current session by the attacher cannot affect the confidentiality of the other sessions.	True	1s	True	7s

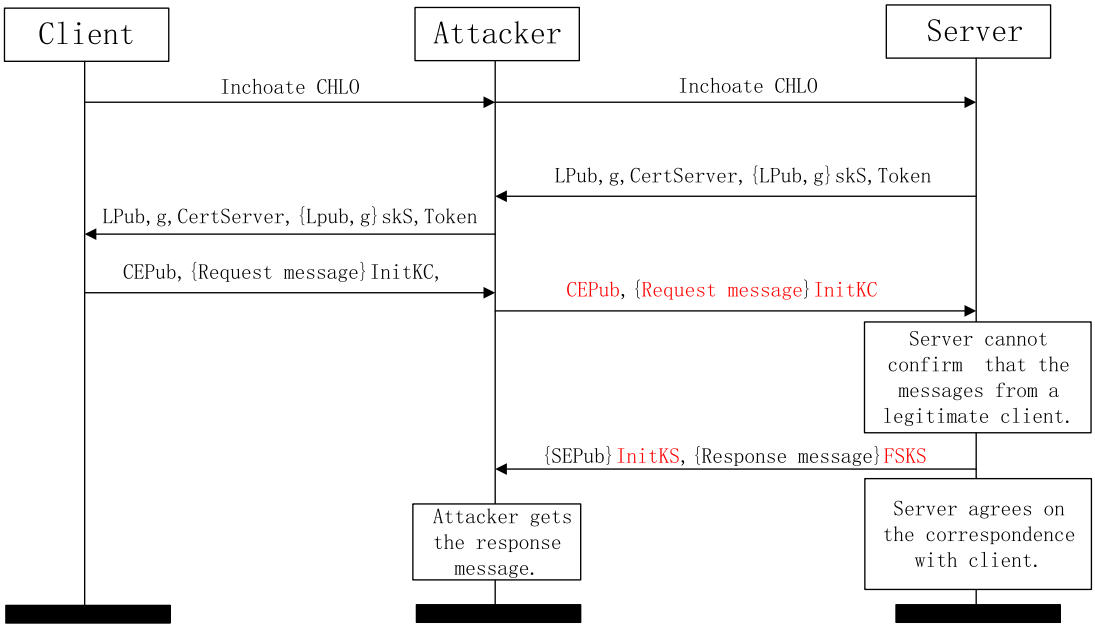

FIGURE 13. The counterexample for property A1, A2 and S2.

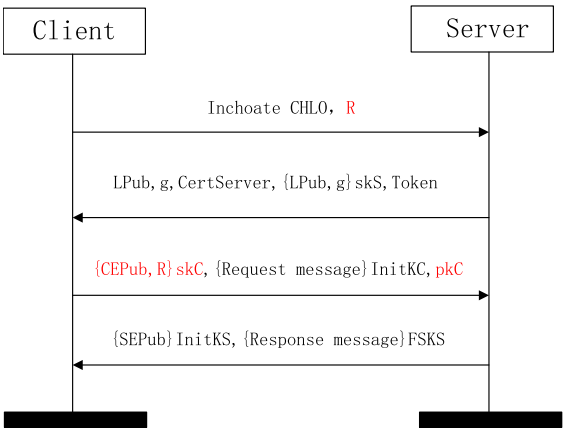

FIGURE 14. The revised QUIC handshake protocol.

server without any protection, such as signature or encryption. The server cannot confirm that *CEPub* is from a legitimate client. The attacker can impersonate the legal client to complete the handshake process with the server. Therefore, the properties $A1$, $A2$, $S2$ are violated, the counterexample is shown in Fig. 13. However the server signs its *LPub* message, the attacker cannot impersonate the server to complete the handshake process with the client. As a possible fix, we propose a revised QUIC handshake protocol shown in Fig.14. First, we add a nonce *R* when the client sends the *CHLO* message to the server. And then we sign both *CEPub* and the nonce *R* with the private key of the client and send it to the server along with the public key certificate of the client. Therefore, the attacker cannot impersonate the client to complete the initial key calculation with the server. We have verified the newly revised QUIC handshake protocol shown in Fig.14, and the newly added elements are marked in red. Our analysis shows that this revised protocol satisfies all the security properties we have considered.

VI. CONCLUSION

In this work, we first investigate the IETF documents of QUIC and extract the security properties that QUIC handshake protocol should be satisfied. Then we model the QUIC handshake protocol and verify its properties using two state-of-the-art symbolic model checkers, namely ProVerif and Verifpal. We also discuss the differences between ProVerif and Verifpal in both protocol modeling and property specification. We also give a comparison of their performance in verifying the QUIC handshake protocol. The verification

results reveal a defect of the protocol design. Furthermore, we also propose a possible fix to repair this design defect.

Our analysis is based on the symbolic model checking, we remark that our analysis is based on the assumption that the cryptographic primitives are perfect. For instance, the hash functions are perfect one-way functions, and not susceptible to attacks like the length extension attack. This is the fundamental difference between tools like ProVerif and Verifpal which operate in the symbolic model, and the other tool like CryptoVerif [35] which operates in the computational model.

In the future, we would like to go one step further to investigate the correctness of the protocol implementations, with respect to the specification. One possible technique to achieve this goal is configurable software verification [36]. We will also extend the current work to the computational cryptography model, where the cryptographic primitives are no longer assumed to be perfect, but the probability of breaking the cryptographic primitives is taken into account.

REFERENCES

[1] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, and J. Bailey, "The QUIC transport protocol: Design and Internet-scale deployment," in *Proc. Conf. ACM Special Interest Group Data Commun., SIGCOMM*, Los Angeles, CA, USA, Aug. 2017, pp. 183–196, doi: 10.1145/3098822.3098842.

[2] L. Hirschi, R. Sasse, and J. Dreier, "Security issues in the 5G standard and how formal methods come to the rescue," *ERCIM News*, vol. 2019, no. 117, 2019.

[3] D. Basin, C. Cremers, and C. Meadows, "Model checking security protocols," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Cham, Switzerland: Springer, 2018, pp. 727–762, doi: 10.1007/978-3-319-10575-8_22.

[4] G. Lowe, "An attack on the needham-schroeder public-key authentication protocol," *Inf. Process. Lett.*, vol. 56, no. 3, pp. 131–133, Nov. 1995.

[5] J. C. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using murϕ," in *Proc. IEEE Symp. Secur. Privacy*, May 1997, pp. 141–151.

[6] D. X. Song, "Athena: A new efficient automatic checker for security protocol analysis," in *Proc. 12th IEEE Comput. Secur. Found. Workshop*, Jun. 1999, pp. 192–202.

[7] E. M. Clarke, S. Jha, and W. Marrero, "Verifying security protocols with brutus," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 443–487, Oct. 2000.

[8] A. Armando and L. Compagna, "SATMC: A SAT-based model checker for security protocols," in *Logics in Artificial Intelligence*, J. J. Alferes and J. Leite, Eds. Berlin, Germany: Springer, 2004, pp. 730–733.

[9] D. Basin, S. Mödersheim, and L. Vigano, "OFMC: A symbolic model checker for security protocols," *Int. J. Inf. Secur.*, vol. 4, no. 3, pp. 181–208, Jun. 2005.

[10] V. Cortier, S. Delaune, and P. Lafourcade, "A survey of algebraic properties used in cryptographic protocols," *J. Comput. Secur.*, vol. 14, no. 1, pp. 1–43, Feb. 2006.

[11] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Found. Trends Privacy Secur.*, vol. 1, nos. 1–2, pp. 1–135, 2016.

[12] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5G authentication," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1383–1396.

[13] C. Cremers and M. Dehnel-Wild, "Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2019, pp. 1–15.

[14] J. Zhang, L. Yang, W. Cao, and Q. Wang, "Formal analysis of 5G EAP-TLS authentication protocol using proverif," *IEEE Access*, vol. 8, pp. 23674–23688, 2020.

[15] M. Abadi, B. Blanchet, and C. Fournet, "The applied pi calculus: Mobile values, new names, and secure communication," *J. ACM*, vol. 65, no. 1, pp. 1–41, Jan. 2018.

[16] H. Comon-Lundh, S. Delaune, and J. K. Millen, "Constraint solving techniques and enriching the model with equational theories," in *Formal Models and Techniques for Analyzing Security Protocols* (Cryptology and Information Security Series), vol. 5, V. Cortier and S. Kremer, Eds. Amsterdam, The Netherlands: IOS Press, 2011, pp. 35–61.

[17] B. Blanchet, "Using horn clauses for analyzing security protocols," in *Formal Models and Techniques for Analyzing Security Protocols* (Cryptology and Information Security Series), vol. 5, V. Cortier and S. Kremer, Eds. Amsterdam, The Netherlands: IOS Press, Mar. 2011, pp. 86–111.

[18] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proc. 14th IEEE Comput. Secur. Found. Workshop*, Jun. 2001, pp. 82–96.

[19] N. Kobeissi, G. Nicolas, and M. Tiwari, "Verifpal: Cryptographic protocol analysis for the real world," in *Proc. 21st Int. Conf. Cryptol. India Prog. Cryptol. (INDOCRYPT)*, Bengaluru, India, Dec. 2020, pp. 151–202.

[20] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Germany: Springer, 2013, pp. 696–701.

[21] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of TLS 1.3," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1773–1788.

[22] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 483–502.

[23] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Apr. 2017, pp. 435–450.

[24] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Trans. Depend. Sec. Comput.*, vol. 5, no. 4, pp. 193–207, Oct. 2008.

[25] K. L. Mcmillan and L. D. Zuck, "Formal specification and testing of QUIC," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 227–240, doi: 10.1145/3341302.3342087.

[26] C. Troncoso *et al.*, "Decentralized privacy-preserving proximity tracing," 2020, *arXiv:2005.12273*. [Online]. Available: http://arxiv.org/abs/2005.12273

[27] M. R. Kanagarathinam, S. Singh, S. R. Jayaseelan, M. K. Maheshwari, G. K. Choudhary, and G. Sinha, "QSOCKS: 0-RTT proxification design of SOCKS protocol for QUIC," *IEEE Access*, vol. 8, pp. 145862–145870, 2020, doi: 10.1109/ACCESS.2020.3013524.

[28] A. Rasool, G. Alpár, and J. de Ruiter, "State machine inference of QUIC," 2019, *arXiv:1903.04384*. [Online]. Available: http://arxiv.org/abs/1903.04384

[29] X. Cao, S. Zhao, and Y. Zhang, "0-RTT attack and defense of QUIC protocol," in *Proc. IEEE Globecom Workshops (GC Wkshps)*, Dec. 2019, pp. 1–6, doi: 10.1109/GCWkshps45667.2019.9024637.

[30] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, "How secure and quick is QUIC? Provable security and performance analyses," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 214–231, doi: 10.1109/SP.2015.21.

[31] J. Zhang, L. Yang, X. Gao, and Q. Wang, "Formal analysis of QUIC handshake protocol using ProVerif," in *Proc. 7th IEEE Int. Conf. Cyber Secur. Cloud Comput. (CSCloud)/ 6th IEEE Int. Conf. Edge Comput. Scalable Cloud (EdgeCom)*, Aug. 2020, pp. 132–138.

[32] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," in *Proc. Internet Eng. Task Force, Internet-Draft Draft-Ietf-Quic-Transp.-27. Work Prog.*, Feb. 2020. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-27

[33] J. Iyengar and M. Thomson, "Using TLS to secure QUIC," in *Proc. Internet Eng. Task Force, Internet-Draft Draft-Ietf-Quic-Tls-27. Work Prog.*, Feb. 2020. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-27

[34] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. IT-29, no. 2, pp. 198–208, Mar. 1983.

[35] B. Blanchet. (2017). *Cryptoverif: A Computationally-Sound Security Protocol Verifier*. [Online]. Available: https://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/cryptove%rif.pdf

[36] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *Proc. 19th Int. Conf. Comput. Aided Verification CAV*, Berlin, Germany, Jul. 2007, pp. 504–518, doi: 10.1007/978-3-540-73368-3_51.

JINGJING ZHANG received the B.S. degree in information security from Beijing Information Science and Technology University, Beijing, China, and the M.S. degree in electronics and communication engineering from the Beijing University of Posts and Telecommunications, Beijing, China. He is currently pursuing the Ph.D. degree with the College of Command and Control Engineering, Army Engineering University of PLA. His research interest includes formal analysis of security protocols from the perspective of design models and implementations.

LIN YANG received the M.S. degree in automation and the Ph.D. degree in communication and electronic system from the National University of Defense Technology, Changsha, Hunan, in 1995 and 1999, respectively. From 2005, he was a Senior Engineer with the China Electronic Equipment and System Engineering Corporation. His research interests include computer security, information system security, network security, trusted computing, security protocol analysis, and big-data security.

XIANMING GAO received the Ph.D. degree in computer science and technology from the National University of Defense Technology, Changsha, Hunan, in 2017. Since 2017, he has been an Engineer with the China National Key Laboratory of Science and Technology on Information System Security. His current research interests include future network architecture, intelligence routing protocol, and network security.

GAIGAI TANG received the B.S. degree in thermal engineering from Tianjin Chengjian University, Tianjin, China, and the M.S. degree in naval architecture and marine engineering from the Jiangsu University of Science and Technology, Zhenjiang, Jiangsu, China. He is currently pursuing the Ph.D. degree in information technology with Harbin Engineering University. His research interests include software security and machine learning.

JIYONG ZHANG (Member, IEEE) received the B.S. and M.S. degrees in computer science from Tsinghua University, in 1999 and 2001, respectively, and the Ph.D. degree in computer science from the Swiss Federal Institute of Technology at Lausanne (EPFL), in 2008. He is currently a Distinguished Professor with Hangzhou Dianzi University. His research interests include intelligent information processing, machine learning techniques, data sciences, and recommender systems.

QIANG WANG received the bachelor's and master's degrees from the National University of Defense Technology, in 2010 and 2012, respectively, and the Ph.D. degree from the Swiss Federal Institute of Technology at Lausanne (EPFL), Switzerland, in 2017, where he has been a key person in the development of the component-based embedded system design framework BIP. His current research interests include formal verification techniques and tools for safety and security critical systems.