ProScript-TLS: Verifiable Models and Systematic Testing for TLS 1.3

Karthikeyan Bhargavan
INRIA
karthikeyan.bhargavan@inria.fr

Nadim Kobeissi
INRIA
nadim.kobeissi@inria.fr

***Abstract*—As TLS progresses into major new revisions in TLS 1.3, implementers are once again tasked with upgrading existing TLS code. This includes dealing with concerns such as testing new cryptographic mechanisms while preserving backwards compatibility and preventing downgrade attacks. It would be attractive for implementers to be able to quickly extract a composite symbolic model from their evolving code and formally verify it against a variety of attacker models. Even when formal verification is not feasible, systematic testing for known and new state machine vulnerabilities is a must, in light of recent attacks like SKIP and FREAK.**

In this position paper, we describe our ongoing work on implementing TLS 1.0-1.3 in ProScript, a statically typed subset of JavaScript. The type system guarantees that our implementation avoids common pitfalls of JavaScript programming and that it can be properly isolated from unverified application code. The ProScript compiler translates the source code into human-readable symbolic cryptographic protocol models that can be verified using ProVerif. These models allow TLS implementers to automatically find logical errors in their code, as well as formally evaluate different combinations of protocol mechanisms for advanced security properties such as forward secrecy. Furthermore, we are using our ProScript implementation to build the next version of FlexTLS: a systematic testing framework for TLS 1.0-1.3 in JavaScript. We show how users can write specific attack scenarios in JavaScript and systematically test implementations for state machine vulnerabilities.

I. Introduction

Writing robust cryptographic software is hard, and TLS implementations are no exception, as evidenced by a series of high-profile attacks, even on well-vetted TLS libraries. Even a single, seemingly innoccuous, programming error can lead to vulnerabilities like HeartBleed that leak long-term secrets and hence completely break the security of the protocol. Finding and preventing such bugs should be within the reach of modern software verification and testing tools, but such tools have difficulty in scaling up to the complexity of TLS libraries, and to the frequency of code updates, allowing bugs to slip in.

Furthermore, protocols evolve, and so too must their implementations. Modern TLS implementations support multiple versions, hundreds of ciphersuites, and dozens of protocol extensions proposed by different groups of authors with different security goals in mind. Each of these protocol mechanisms may be secure in isolation, but sometimes it is only when they are composed in implementations that flaws in their design come to light, sometimes many years after standardization. For example, the Triple Handshake attacks only appear when standard TLS handshakes are composed with both session resumption and mutually-authenticated renegotiation, a combination that had not been analyzed before but was commonly supported by TLS libraries [2].

Even if programming errors like HeartBleed were eliminated (via a static type system, for example) and the composition of all protocol modes that we wish to support were proved secure (by a cryptographer), the TLS implementator still has to address the challenge of designing a composite state machine that multiplexes between all these supported modes depending on the result of in-protocol negotiation. Designing correct composite state machines has proved to be surprisingly hard for developers, leading to serious attacks like SKIP and FREAK on mainstream TLS libraries [1].

Formal Verification. One approach to addressing these challenges is formal verification, as demonstrated by the miTLS project [4]. The protocol implementation is broken down into small modules each of which represents a single protocol mechanism, such as a cryptographic construction, or message formats for a sub-protocol. Each module is then annotated with a formal specification that consists of cryptographic assumptions, functional correctness invariants, and security goals. A dependent type checker can then verify that the module meets its specification. By composing the type-based proofs for all the modules, we obtain a cryptographic security theorem about the full miTLS API, which essentially says that, under certain assumptions, any connection between an honest miTLS client and honest miTLS server can be seen as a (particular kind of) secure channel.

The miTLS verification method is well suited to building a protocol implementation and its cryptographic proof side-by-side. However, this style is not designed to give quick feedback about a new protocol mechanism or an experimental implementation design. In particular, dependent type checking as used in miTLS cannot find new attacks. At best, if type checking fails, we may be able to occasionally interpret this failure as a potential attack, but more often than not, such failures indicate a missing type annotation.

Verifying a TLS 1.3 implementation with ProVerif. As TLS 1.3 reaches stability, every draft adds a host of new features to the protocol. Our goal is to build a verified implementation of TLS 1.3 that is capable of giving quick feedback on these new features with minimal annotation effort. To this end, we choose a more light-weight verification method than miTLS. From our implementation, we extract symbolic protocol models in the applied pi calculus, and verify them with ProVerif, a state-of-the-art cryptographic protocol analyzer [5].

ProVerif can automatically analyze the extracted protocol in the so-called Dolev-Yao or symbolic model of cryptography and prove security properties such as secrecy and authenticity, for unlimited sessions of the protocol, even under sophisticated key compromise assumptions. When the proof fails, ProVerif typically produces a counter-example that can be read as an attack trace. However, since the protocol verification problem is undecidable, ProVerif may not terminate.

If ProVerif says that a protocol is secure in the symbolic model, this does not usually translate to a cryptographic proof of security. Consequently, positive verification results for our TLS 1.3 implementation should be seen as adding confidence to the protocol design and to our implementation, but not as a definitive proof. Conversely, our method is capable of finding early attacks and influencing the protocol. For example, the Unknown Key Share and Key Compromise Impersonation attacks on 0-RTT client authentication documented in the current TLS 1.3 draft were found by analysis with ProVerif.

Implementing and Testing TLS 1.3 in JavaScript. To summarize, our design goal for our TLS 1.0-1.3 implementation is that it should enable the following capabilities:

1) **Rapid Prototyping** As new features, such as post-handshake client authentication, are added to TLS 1.3, we should be able to quickly implement these features and test for interoperability with other TLS 1.3 libraries.
2) **Ease of Deployment** Other TLS developers should be able to use our implementation for testing and verification. Furthermore, we want to make it easy for TLS developers to contribute new tests and features to our code. To this end, we choose to build our implementation in JavaScript using the Node framework, which we believe will be more accessible than F* or OCaml implementations.
3) **Automated Verification** We should be able to symbolically verify the full TLS protocol as implemented by our code. So, we write our core TLS code in ProScript, a statically typed subset of JavaScript, and automatically extract ProVerif models from this code.
4) **Systematic Testing** We want to implement a systematic testing framework, like FlexTLS [1], that TLS 1.3 developers can use to run a series of attack scenarios against their fresh code.

By the TRON workshop, we plan to have an interoperable implementation, verified models, and a systematic testing framework for TLS 1.3. The rest of this paper describes our ongoing work towards these goals.

II. Implementing TLS in ProScript

Our TLS implementation supports TLS 1.0-1.3 and is written in a mix of standard unverified JavaScript and verified ProScript modules. Standard JavaScript is used for defining clients and servers; this code can use the Node libraries, for example to initiate and accept TCP connections, or to manage packet buffers, but is not used for any cryptographic computations. The core protocol code is written in ProScript, a statically typed subset of JavaScript.

The full details of ProScript are in a paper currently under submission to another conference. A confidential draft of a paper (currently under review) is available at [6]. As we shall see in the next section, ProScript is designed to write cryptographic protocol that can be easily translated to the input syntax of ProVerif. The type system of ProScript is adapted from Defensive JavaScript [3]; it guarantees that ProScript code is well-typed and that its behavior cannot be tampered with by external JavaScript code, an essential starting point for any verified code running within an untrusted application.

The ProScript TLS code is written in a purely functional state-passing style; it relies on the statically-typed (but otherwise unverified) ProScript cryptographic library (PSCL), which also includes utility functions for manipulating bitstrings and byte arrays, but the protocol code does not access any other external libraries. By running the ProScript compiler on this protocol code, we extract a composite protocol model for TLS 1.0-1.3 in ProVerif that can be verified for various protocol-specific security properties. We could have written such a ProVerif model by hand, but the key advantage of using our method is that the extracted model is faithful to an interoperable implementation of the protocol, and hence is less likely to overlook an important but messy detail.

Our implementation currently implements TLS 1.0-1.2 and is being extended to TLS 1.3 draft 11. It supports RSA, DHE, and ECDHE key exchanges, RSA signatures, AES-CBC and AES-GCM encryption, and SHA-1 and SHA-256 hashing. We have tested our code for interoperability against other TLS libraries. In total, our implementation currently has 3,500 lines of code (plus 3,000 lines of ProScript libraries).

The implementation provides a connection API that consists of four functions, two in JavaScript and two in ProScript:

- **TLS Server Instantiation and Callbacks** The JavaScript function `tls_server(port, callbacks)` starts a TLS server instance that listens at the port `port`. When it receives a flight of messages from its peer, it calls the corresponding function from `callbacks`. The caller is free to define these callbacks in any way she wants, and as we shall see, this proves to be quite useful for writing FlexTLS scenarios. By default, however, the application would always use the standard TLS 1.0-1.3 callbacks that it can obtain by calling the ProScript function `tls_server_callbacks(config)`. This function returns an object with several state-passing functions, each of which processes one flight of messages and provides a new flight in return. For example, in TLS 1.0-1.3 `hs_recv_client_hello(msgs,state)` is called when the server receives the first flight of messages (`msgs` consists of a single ClientHello); it returns a modified `state` and a flight of server messages to be sent to the client. In TLS 1.0-1.2, `hs_recv_client_ccs(msgs,state)` processes the second flight of messages up to ClientCCS and installs new record keys in `state`, while `hs_recv_finished` processes the ClientFinished and returns the server's last flight up to ServerFinished.
- **TLS Client Instantation and Callbacks** On the client side, the JavaScript function `tls_client(host, port, callbacks)` connects to `host` on `port` and invokes the callbacks to

process each flight of messages. The default TLS callbacks are returned by the ProScript function `tls_client_callbacks(config)`. The function `hs_send_client_hello(state)` returns the ClientHello, while `hs_recv_server_hello_done` and `hs_recv_server_finished` process the two flights of server messages. Like the server callbacks, these are all state-passing functions that take one array of messages and return another.

The full TLS API also provides functions for sending and receiving application data and alerts at appropriate stages of the connection. For instance, in TLS 1.3, the client can send 0-RTT data after the client hello and 1-RTT data after the handshake is complete. Using this API, it is easy to deploy simple client and server TLS applications, and also to write non-conformant TLS clients or servers for testing.

We are using our ProScript TLS implementation as a basis for building the next version of FlexTLS [1] in JavaScript. FlexTLS allows developers and protocol experts to write brief attack scenarios and try them out against other TLS implementations. An analyst can also use FlexTLS in a exploratory mode, by enumerating a large set of invalid TLS traces and systematically testing them all on some TLS peer.

The previous version of FlexTLS was written in F# and was based on the miTLS code base. Although this tool was instrumental in finding new attacks like FREAK, and to demonstrate the effectiveness of attacks like Logjam, we found that the old version was hard to install on different platforms, and developers were reluctant to write new tests in an unfamiliar language. We believe that the new JavaScript version will be easier to deploy and to use. Furthermore, we aim to produce a large test-suite that TLS developers can use to evaluate their new TLS 1.3 implementations.

Our TLS API makes it easy for developers to tamper with the internals of the protocol. Although the `tls_client` and `tls_server` functions described above normally use the standard TLS callbacks, the application can override this behavior by redefining these callback functions. For example, it can change how the master secret and connection keys are computed, or what messages are sent in a particular flight.

As an illustrative example, we show how to implement the SKIP CCS attack [1] in JavaScript using our API. In this attack, a malicious server skips the ServerCCS message which is meant to signal the start of record encryption and instead goes straight to the ServerFinished message. The client should refuse to accept this message sequence, but it turned out that Java and CyaSSL/WolfSSL clients would accept this server behavior and continue the connection, but with encryption disabled. This is already a suspicious bug, but in combination with other state machine flaws, it can be exploited by a network attacker to fully impersonate any TLS server (without needing to know its certificate private key.)

To test whether a client is vulnerable to the SKIP CCS attack, we write the attack scenario depicted in Figure 1. This code starts a TLS server (at port 443) that leaves all callbacks unchanged except for `hs_recv_client_finished`; on receiving the second client flight (ending in ClientFinished), the server first calls the standard TLS callback, which is expected to return an array with ServerCCS and ServerFinished. It deletes the

```
const tls_cb = tls_server_callbacks(config);
const SKIP_server_callbacks = {
 hs_recv_client_hello: tls_cb.hs_recv_client_hello,
 hs_recv_client_ccs: tls_cb.hs_recv_client_ccs,
 hs_recv_client_finished: function(msgs, state) {
   let out_msgs =
      tls_cb.hs_recv_client_finished(msgs, state);
   state.require_alert = true;
   return [out_msgs[1]];
 }
}
tls_server(443, SKIP_server_callbacks);
```

Fig. 1. SKIP CCS: a FlexTLS attack scenario in JavaScript

first message (by returning just `[out_msgs[1]]`) and marks the `state` with `require_alert` which means that the next message received from the client must be a fatal alert; otherwise the server would log an error message.

We found this scenario to be easy and intuitive to write; other scenarios require deeper changes to the callbacks, but still require as much or less effort than the F# code in FlexTLS. We are currently building a full test suite that can test for all such state machine bugs in TLS 1.0-1.3 implementations.

III. Extracting ProVerif Models from ProScript

In this section, we give a flavor of what ProScript programs and their extracted ProVerif models look like. We refer the reader to [6] for more details on the ProScript language, its compiler, and a larger case study.

ProScript is a purely functional language with a focus on automatic compilation to formally verifiable models. ProScript code is written defensively, in that it cannot, even accidentally, access external libraries or extensible JavaScript functionalities such as object instatiation, or redefinable properties such as `Array.split`. These restrictions are necessary in JavaScript where external functions can completely redefine the behavior of all libraries and object prototypes. The resulting style enforces syntactic scoping and strict type checking for all variables and functions, and disallows implicit coercions and dynamic extensions of arrays and objects.

ProScript programs are structured with two guiding principles:

1) Modules for verifiable components: Each ProScript program consists of parts that are trusted (such as the cryptographic primitives), parts that are verified (such as the core protocol code), and parts that are untrusted and unverified (such as message parsing). We employ Node modules in order to demarcate entire modules in one of these three categories. In the translated ProVerif, trusted modules are mapped to idealized symbolic models, whereas untrusted modules are assumed to be controlled by the attacker. Other modules are translated to ProVerif functions. For example, in our TLS implementation, the `TLS.protocol` module contains handshake code and is verified, the `ProScript.crypto` module contains the trusted cryptographic library (PSCL) and is translated to a symbolic cryptographic model, and the `TLS.application` module, which is in JavaScript and manages TCP connections, is untrusted and assumed to be controlled by the attacker.

2) Protocols as pure state-passing functions: ProScript modules consist of a collection of pure functions, that have

```
const ctx:bitstring.
const s:key [private].
free secretMessage:bitstring [private].
event Send(key, key, bitstring).
event Recv(key, key, bitstring).
query a:key,b:key,m:bitstring; event(Recv(a, b, m)) ==>
     event(Send(a, b, m)).
query attacker(secretMessage).

process
 let gS = ProScript_crypto_DH25519(s, key_15) in
 (* Client process *)
(let sendOutput = fun_sendMessage(gS, ctx, secretMessage)
      in
 let gX = ProScript_crypto_DH25519(Object_sendOutput_get_x(
      sendOutput), key_15) in
 event Send(gX, gS, secretMessage);
 out(io, (Object_sendOutput_get_data(sendOutput), gX,
      Object_sendOutput_get_iv(sendOutput))))
| (* Server process *)
(in(io, (recvData:bitstring, gC:key, civ:iv));
 let recvOutput = fun_recvMessage(s, ctx, gC, recvData, civ
      ) in
 if (Object_aesgcmresult_get_valid(recvOutput) = true) then
 event Recv(gC, gS, Object_aesgcmresult_get_data(recvOutput
      )))
```

Fig. 2. Security queries and top-level process for the example protocol

no side-effects, but may take a state object as a parameter and return a modified state in the result. As we showed in our TLS API, this style can be used to code protocols as a set of message processing functions. Not all the functions in a module are public; using the Node module system, we only export selected functions to other modules.

Verification Example: In order to illustrate how formal verification works, we implement a simple protocol inspired by TLS 1.3's 0-RTT mode. The client already knows the server's semi-static public value g^s (presumably obtained during a prior 1-RTT exchange.) The client now generates a new ephemeral key pair (x, g^x), and sends a secret application data message under an encryption key derived from g^{xs}.

Figure 3 shows a simple but fully functional implementation of this protocol in ProScript. Figure 4 shows this same code compiled into a ProVerif symbolic model. Unlike previous attempts at automated translations into symbolic models, ProScript-produced models are designed to be human-readable and can be modified by hand. This allows the analyst to easily model a variety of attackers and security goals, and to experiment with small modifications of the protocol without modifying the source code.

In Figure 2, we show a top level process that models a client and server running in parallel. The client encrypts and sends a private `secretMessage` to the server who decrypts it. The client marks its intention to send the message by issuing a `Send` event before sending and the server issues a `Recv` send to signal that it accepts the message. Note that we do not explicitly model the attacker; ProVerif automatically models an attacker who is able to read all public values and send and receive on all public channels.

Above the process, the figure also shows security queries that test for confidentiality and integrity of `secretMessage`. ProVerif can verify these queries in seconds. We can also verify that `secretMessage` is indistinguishable from a freshly generated message, or that it is forward secret. In this model, of course, the message is not forward secret if the attacker can compromise the server's semi-static key `s`, and ProVerif finds this attack and produces a counterexample.

Figure 4 also illustrates how the ProScript cryptographic library is mapped to a symbolic model in ProVerif. For example, the Diffie-Hellman relationship $g^{ab} = g^{ba}$ is defined equationally on lines 31-34, where `key_15` is the base point value for Curve25519. We have defined such standard models for each primitive in our library, but the user is free to change this model to reflect different cryptographic assumptions.

IV. Work in Progress

This position paper describes work in progress. Our goal is to have a fully interoperable TLS 1.3 implementation, a verifiable protocol model, and a comprehensive test suite released and ready for demonstration at TRON.

A key goal for our implementation and ProVerif model is that we should be able to evaluate and justify all the design decisions made in the draft TLS 1.3 protocol. For example, we are currently able to reconstruct the Unknown Key Share and Key Compromise Impersonation attacks on 0-RTT client authentication, and then show that former is prevented by adding the server certificate to the 0-RTT handshake hash. We would like to similarly evaluate other modes of the protocol, and especially the composition of different ciphersuites and protocol versions. We believe that analyzing such composite protocols makes automated verification necessary; humans struggle to find corner cases in such large protocols.

In parallel to our effort, the miTLS team is building a verified implementation of TLS 1.0-1.3 in F*, and we see that effort as complementary to ours (some of us are contributing to both implementations.) In the long run, we hope to use the F*-to-JavaScript backend to generate some of the core TLS modules that we use in our ProScript implementation. These modules would then be provably secure by construction, but we would still use ProScript to implement new experimental mechanisms, verify them quickly with ProVerif, and find concrete attacks early, before integrating them in miTLS.

References

[1] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J.K. Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, 2015.

[2] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, , Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *IEEE Symposium on Security & Privacy 2014 (Oakland'14)*. IEEE, 2014.

[3] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security Symposium*, pages 653–670, 2013.

[4] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459, 2013.

[5] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[6] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Proscript: Automated formal verification for cryptographic web applications, 2015. Unpublished, under review. Confidential draft available at https://jsverif.org/doc/proscript.pdf. Username: tron2016 - Password: proscript-tls.

```
const Type_key = {
 construct: function() {
  return [
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00
  ]
 },
 assert: function(a) {
  var i = 0; for (i = 0; i < 32; i++) {
   a[i&31] & 0x00
  }; return a
 }
}

const Type_iv = {
 construct: function() {
  return [
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00
  ]
 },
 assert: function(a) {
  var i = 0; for (i = 0; i < 16; i++) {
   a[i&15] & 0x00
  }; return a
 }
}

const Type_sendOutput = {
 construct: function() {
  return {
   x: Type_key.construct(),
   iv: Type_iv.construct(),
   data: ''
  }
 },
}

const sendMessage = function(gS, ctx, m) {
 const iv = ProScript.crypto.random16Bytes('iv')
 const x = ProScript.crypto.random32Bytes('pk')
 const gXS = ProScript.crypto.DH25519(x, gS)
 return {
  x: Type_key.assert(x),
  iv: Type_iv.assert(iv),
  data: ProScript.crypto.AESGCMEncrypt(HKDF(gXS, ctx), iv, m)
 }
}

const recvMessage = function(s, ctx, gX, c, iv) {
 const gXS = ProScript.crypto.DH25519(s, gX)
 const p = ProScript.crypto.AESGCMDecrypt(
  HKDF(gXS, ctx), Type_iv.assert(iv), c
 )
 if (p.valid) {
  return {
   valid: true,
   data: p.data
  }
 }
 else {
  return {
   valid: false,
   data: p.data
  }
 }
}
```

Fig. 3. A ProScript implementation of a simple messaging protocol inspired by the TLS 1.3 0-RTT application data protocol.

```
free io:channel.

type number.
type function.
type args.
type return.
type object_aesgcmresult.
type key.
type iv.
type object_sendOutput.

fun Type_key_construct():key.
fun Type_key_toBitstring(key):bitstring [data, typeConverter].
fun Type_key_fromBitstring(bitstring):key [data, typeConverter].
reduc forall a:key; Type_key_assert(a) = a.
fun Type_iv_construct():iv. [...]
fun Type_sendOutput_construct():object_sendOutput. [...]

fun Object_aesgcmresult(bitstring, bool): object_aesgcmresult [data].
[...]

fun Object_sendOutput(bitstring, iv, key):object_sendOutput [data].
[...]

const string_7:bitstring [data]. (* iv *)
const string_11:bitstring [data]. (* pk *)
const key_0:key [data].
const iv_0:iv [data].
const key_15:key [data].

fun ProScript_crypto_DH25519(key, key):key.
equation forall a:key, b:key;
 ProScript_crypto_DH25519(b, ProScript_crypto_DH25519(a, key_15)) =
 ProScript_crypto_DH25519(a, ProScript_crypto_DH25519(b, key_15)).
fun ProScript_crypto_AESGCMEncrypt(key, iv, bitstring): bitstring.
reduc forall k:key, i:iv, m:bitstring; ProScript_crypto_AESGCMDecrypt(
 k, i, ProScript_crypto_AESGCMEncrypt(k, i, m)
) = Object_aesgcmresult(m, true).
fun ProScript_crypto_random32Bytes(bitstring):key [private].
fun ProScript_crypto_random16Bytes(bitstring):iv [private].
fun fun_HKDF(key, bitstring):key.

letfun fun_sendMessage(gS:key, ctx:bitstring, m:bitstring) =
let iv = ProScript_crypto_random16Bytes(string_7) in
let x = ProScript_crypto_random32Bytes(string_11) in
let gXS = ProScript_crypto_DH25519(x, gS) in
Object_sendOutput(
 ProScript_crypto_AESGCMEncrypt(
 fun_HKDF(gXS, ctx), iv, m), Type_iv_assert(iv), Type_key_assert(x)
).

letfun fun_recvMessage(s:key, ctx:bitstring, gX:key, c: bitstring, iv:iv) =
let gXS = ProScript_crypto_DH25519(s, gX) in
let p = ProScript_crypto_AESGCMDecrypt(fun_HKDF(gXS, ctx), Type_iv_assert(iv), c) in
if (Object_aesgcmresult_get_valid(p)) then (
 Object_aesgcmresult(Object_aesgcmresult_get_data(p), true) )
else (
 Object_aesgcmresult(Object_aesgcmresult_get_data(p), false )).


process
```

Fig. 4. A ProVerif model automatically generated by the ProScript compiler. See Figure 2 for security queries and top-level process.