Towards a Verified Reference Implementation of a Trusted Platform Module

Aybek Mukhamedov[1,2], Andrew D. Gordon[1], and Mark Ryan[2]

[1] Microsoft Research
[2] University of Birmingham

Abstract. We develop a reference implementation for a fragment of the API for a Trusted Platform Module. Our code is written in a functional language, suitable for verification with various tools, but is automatically translated to a subset of C, suitable for interoperability testing with production code, and for inclusion in a specification or standard for the API. One version of our code corresponds to the widely deployed TPM 1.2 specification, and is vulnerable to a recently discovered dictionary attack; verification of secrecy properties of this version fails producing an attack trace and highlights an ambiguity in the specification that has security implications. Another version of our code corresponds to a suggested amendment to the TPM 1.2 specification; verification of this version succeeds. From this case study we conclude that recent advances in tools for verifying implementation code for cryptographic APIs are reaching the point where it is viable to develop verified reference implementations. Moreover, the published code can be in a widely understood language like C, rather than one of the specialist formalisms aimed at modelling cryptographic protocols.

1 Introduction

The Trusted Platform Module (TPM) is designed to enable trustworthy computation and communication over open networks by realizing robust platform integrity measurement and reporting, secure platform attestation, secure storage and other security mechanisms. TPM is part of the Trusted Computing Base and interacts with applications via a pre-defined set of commands (an API). To guarantee its reliability it is important that the TPM commands are defined unambiguously and do not give rise to subtle interactions and functionality unforeseen by its designers. Design failures can lead to expensive recalls of computers and costly replacements of the embedded TPMs.

Since the TPM 1.2 specification defines more than 90 commands (and the next version is expected to have even more functionality), it is not feasible to reliably check the API for the absence of such issues by manual inspection alone. Indeed, several security vulnerabilities have already been identified in the literature [12, 18, 19, 13]. The API's security analysis needs to be automated with computer-aided verification techniques. Formal methods from the security protocols verification domain appear appropriate.

The underlying motivation for employing formal security protocols analysis methods is that security APIs such as that of the TPM can be seen as two-party protocols between a user and the security module. The aim of the attacker is to compose a sequence of messages that breaks the expected security property (e.g. divulges a secret data stored in the module). However, analysis of security APIs is different from that of standard protocols, as security modules maintain mutable states across sessions and may exhibit subtle interactions via error messages and conditions.

We have developed a reference implementation for the TPM's authorization and encrypted transport session protocols in the F# programming language [21]. Our implementation is based on data structures and command instructions taken from the specification of the TPM commands [22]. We subsequently performed formal verification of the protocols using the verification toolchain FS2PV [8] and ProVerif [9], for the secrecy of weak authorization data (authdata). The analysis captured the weak authdata attacks on the authorization protocols recently uncovered by Chen and Ryan [13] and highlighted an ambiguity in the specification of the encrypted transport session protocol that has security implications. Our analysis also pointed us towards a simple amendment to the encrypted transport session protocol that allows it to protect authorization protocols against weak authdata attacks. Lastly, we have implemented a translator that converts TPM commands and data structure specification written in an F# fragment into executable C code. We have coded up a sample TPM client in C++ in order to demonstrate executability of the generated C specification.

Our contributions. In summary, the contributions of this work are:

- A reference implementation of the TPM's authorization and encrypted transport session protocols in the F# programming language.
- A formal analysis of the implementation code with the FS2PV/ProVerif toolchain against weak authdata secret attacks, that captures a known attack[13], highlights an ambiguity in the specification document[22], and proves correctness of our proposed amendment.
- A translator F2C that generates executable C code from an implementation written in a functional fragment of F#.

Related work. The research into API security analysis has been instigated by Bond et al. [11, 10], who thoroughly examined interfaces of several security devices, including IBM's 4758 hardware security module (HSM) and uncovered many pernicious attacks. Although fruitful, the analysis was ad hoc and consisted only of manual API inspection. Such an approach clearly can neither guarantee that all attacks are discovered nor prove the absence of them. A similar study by Berkman et al. [5] uncovered PIN derivation attacks on financial PIN processing APIs.

Subsequently, a joint MIT/Cambridge research group attempted to perform a formal analysis of 4758 API with general purpose verification tools: the theorem prover Otter [20] and the model checker Failures-Divergence Refinement (FDR)

[17]. They were able to uncover some unknown attacks with Otter (with human guidance) [19], but analysis with FDR did not prove to be so fruitful.

In another approach Steel et al. [14] performed a formal analysis of the revised IBM 4758 HSM with a model checker CL-AtSe from the AVISPA tool set [1]. They found a weakness in a symmetric key importing operation, and proved correctness of other revisions. The authors also proposed a class of protocols that includes the IBM HSM API, in which they showed secrecy to be decidable for an unbounded number of sessions, and developed an ad hoc decision procedure that can perform such analysis. More recently, Steel et al. [15] using the NuSMV model checker to perform the first verification of the PKCS#11 API to account for mutable states, albeit with a bounded number of nonces and other restrictions.

Contents of this Paper. Section 2 gives an overview of the TPM architecture, its authorisation mechanisms, a recent vulnerability (due to Chen and Ryan [13]) that we use as motivation for this work, and a brief overview of the FS2PV framework we use in our formal analysis. Section 3 explains our reference implementation of a fragment of the TPM API, its formal analysis and C code generation. Our verification tools confirm the Chen/Ryan vulnerability, and verify a potential amendment to the API. Section 4 concludes and sketches some potential future work.

2 Background

2.1 TPM overview

The Trusted Platform Module (TPM) specification is an international standard coordinated by the Trusted Computing Group (TCG), for realizing the Trusted Platform in commodity hardware [22, 3, 4]. TPMs are chips that aim to enable computers to achieve greater levels of security than is possible in software alone. There are over 100 million TPMs currently in existence, mostly in high-end laptops. Application software such as Microsoft BitLocker and HP ProtectTools use the TPM in order to guarantee security properties.

The Trusted Platform provides three features: protected storage, platform integrity measurement and integrity reporting. Security guarantees that these features provide rely on three components that implement critical operations of the trusted platform. They are called *roots of trust* and they must be trusted to function correctly: Root of Trust for Measurement (RTM), Root of Trust for Reporting (RTR) and Root of Trust for Storage (RTS). In the current implementations, the TPM acts as RTR and RTS.

Authorization. TPM objects that do not allow "public" access have authorisation data, *authdata* for short, associated with them (such objects include TPM keys, encrypted blobs and owner privileged commands). Authdata is a 20 byte secret that may be thought of as a password required to access such TPM objects. A process requesting access needs to demonstrate that it knows the relevant authdata. This is realized via authorization protocols, where a TPM command is accompanied with an HMAC [2] keyed on the authdata or on a shared secret derived from the authdata. When a new object is created the client chooses its authdata, which ideally should be a high-entropy value.

The TPM provides two kinds of authorisation protocols, called *object independent authorisation protocol* (OIAP) and *object specific authorisation protocol* (OSAP). OIAP allows multiple objects to be used within the same session, but it does not allow commands that introduce new authdata, and it does not allow authdata for an object to be cached for use over several commands. An OSAP session is restricted to a single object, but it does allow new authdata to be introduced and it creates a session secret to securely cache authorisation over several commands. Figures 1 and 2 below demonstrate sample message flows in OIAP and OSAP executions with `TPM_OwnerClear` command, which resets the TPM to un-initialized and un-owned state and clears its internal secret values (ownerAuth, tpmProof, SRK, etc.).

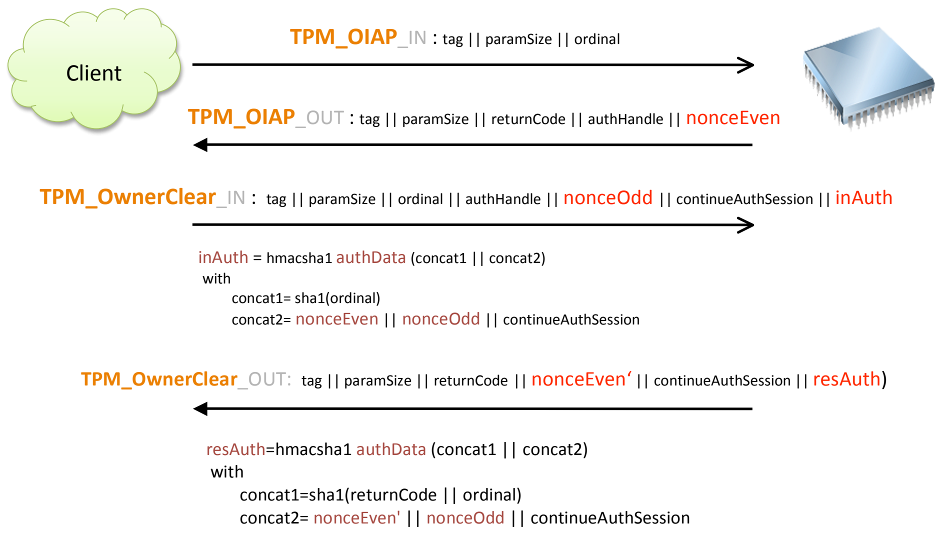

Client
TPM_OIAP_IN : tag || paramSize || ordinal
TPM_OIAP_OUT : tag || paramSize || returnCode || authHandle || nonceEven
TPM_OwnerClear_IN : tag || paramSize || ordinal || authHandle || nonceOdd || continueAuthSession || inAuth
inAuth = hmacsha1 authData (concat1 || concat2)
with
concat1= sha1(ordinal)
concat2= nonceEven || nonceOdd || continueAuthSession
TPM_OwnerClear_OUT: tag || paramSize || returnCode || nonceEven' || continueAuthSession || resAuth)
resAuth=hmacsha1 authData (concat1 || concat2)
with
concat1=sha1(returnCode || ordinal)
concat2= nonceEven' || nonceOdd || continueAuthSession

Fig. 1. TPM_OwnerClear executed in a session initiated using Object Independent Authorisation Protocol (OIAP).

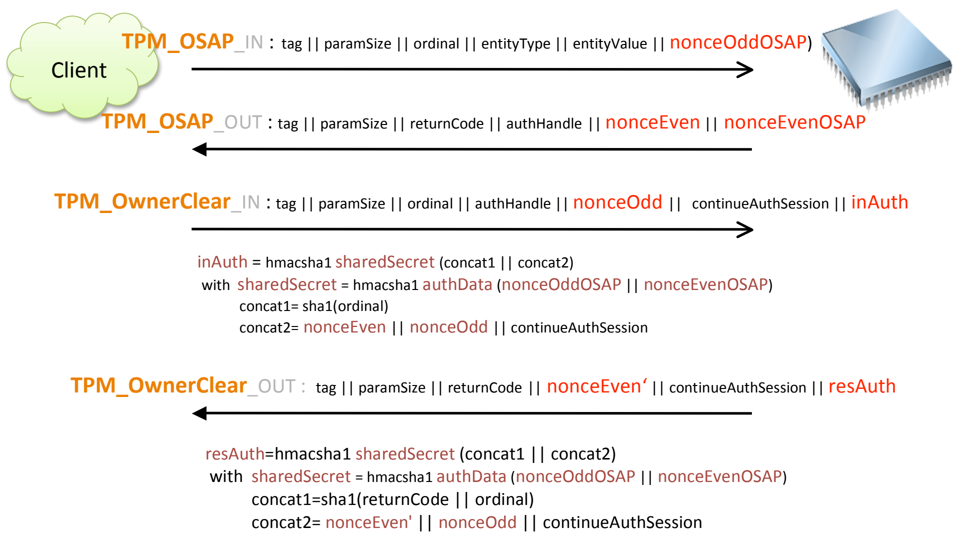

Client
TPM_OSAP_IN : tag || paramSize || ordinal || entityType || entityValue || nonceOddOSAP)
TPM_OSAP_OUT : tag || paramSize || returnCode || authHandle || nonceEven || nonceEvenOSAP
TPM_OwnerClear_IN : tag || paramSize || ordinal || authHandle || nonceOdd || continueAuthSession || inAuth
inAuth = hmacsha1 sharedSecret (concat1 || concat2)
with sharedSecret = hmacsha1 authData (nonceOddOSAP || nonceEvenOSAP)
concat1= sha1(ordinal)
concat2= nonceEven || nonceOdd || continueAuthSession
TPM_OwnerClear_OUT : tag || paramSize || returnCode || nonceEven' || continueAuthSession || resAuth
resAuth=hmacsha1 sharedSecret (concat1 || concat2)
with sharedSecret = hmacsha1 authData (nonceOddOSAP || nonceEvenOSAP)
concat1=sha1(returnCode || ordinal)
concat2= nonceEven' || nonceOdd || continueAuthSession

Fig. 2. TPM_OwnerClear executed in a session initiated using Object Specific Authorisation Protocol (OSAP).

2.2 Weak secrecy

Weak secrets are secret values that have low entropy, such as those derived from memorable passwords and short PIN numbers. The domain that such secrets are chosen from can be efficiently enumerated by the attacker and therefore protocols that make use of weak secrets need to ensure they are obfuscated with high entropy values when sent in messages on the open network.

Guessing attacks on weak secrets can be either on-line or off-line. In the former case, the attacker is able to interact with other agents to verify all their guesses. On-line guessing attacks can be mitigated, for example, by limiting the number of successive failures allowed in protocol execution. Such mechanism, in particular, needs to be employed by the TPM as stated in the specification documents (the details are manufacturer specific). In off-line guessing attacks, the attacker tries to verify their guesses with the help of intercepted messages sent between protocol participants.

In this paper we are concerned with weak secrecy analysis against off-line attacks. Such secrecy has been defined by Blanchet, Abadi and Fournet in [9], which informally can be stated as follows.

Definition 1 (Weak secrecy). *A protocol prevents off-line guessing attacks against the weak secret w, if after the execution of the protocol the attacker cannot distinguish w used in the protocol from an unrelated fresh value.*

Off-line weak authdata attack. A vulnerability was uncovered by Chen and Ryan [13], which allows the attacker to recover low-entropy authdata secrets by off-line dictionary attack on messages exchanged between a legitimate user and the TPM. The attacker can access the messages either by tapping the TPM's databus, compromising the software stack that manages communication of the user with the TPM, or in case of a remote user, by listening in on the unprotected traffic at any point on the network. Knowledge of authdata gives the attacker unrestricted access to the TPM object with which it is associated (for example, knowing authdata of a signing key will allow the attacker to create digital signatures on messages of its own choice based on that key, via a call to `TPM_Sign` command).

Chen and Ryan observed that in OIAP and OSAP sessions all high entropy message components (nonces) are sent out in the clear, and therefore, do not obfuscate the weak authdata used as an HMAC key value in constructing command authorization digests.

For OIAP sessions the attacker tries to test their guess $\mathsf{authdata}'$ by attempting to reconstruct the authorization digest sent by the user to the TPM (value **inAuth** in the third message in Figure 1):

$$\mathsf{HMAC}_{\mathsf{authdata}'}(\mathsf{sha1}(param), \mathsf{nonceEven}, \mathsf{nonceOdd})$$

$$? =$$

$$\mathsf{HMAC}_{\mathsf{authdata}}(\mathsf{sha1}(param), \mathsf{nonceEven}, \mathsf{nonceOdd})$$

where *param* is a concatenation of command parameters, $\mathsf{nonceEven}$ and $\mathsf{nonceOdd}$ are rolling session nonces. All of those message components are available to the attacker as they are sent in clear over the network.

Similarly for OSAP sessions, the attacker tries to test their guesses by reconstructing authorization digest sent by the user to the TPM (value **inAuth** in Figure 2).

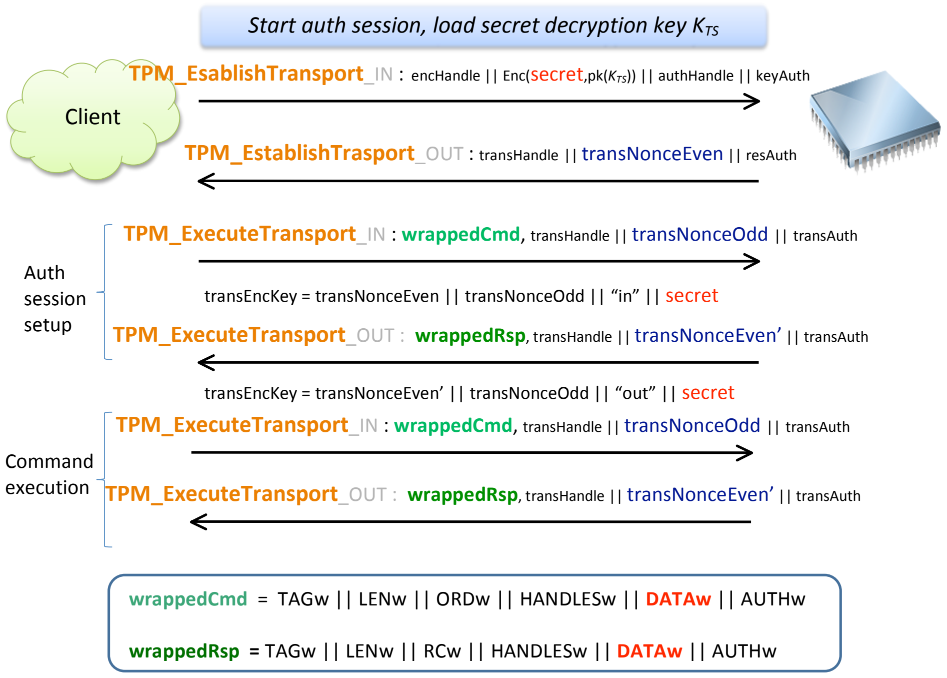
2.3 Encrypted Transport Protocol

TPM supports encrypted transport protocol to allow logging and encryption of commands using a transport session. We studied the protocol to ascertain whether this readily available TPM facility can protect authorization sessions against the weak authdata attacks mentioned above.

Transport sessions can intuitively be thought of as a wrapper for other commands. They proceed by establishing a shared secret that is subsequently used to authorize and encrypt relevant commands sent and received by the TPM. The user of the transport session can execute any command within a transport sessions, except for a command that creates another transport session.

Transport sessions are initiated with the `TPM_EstablishTransport` command, which exchanges transport session nonces and a secret generated by the user. The secret is used later as an HMAC key to generate authorization digests and as a part of the symmetric encryption key for wrapped commands. `TPM_ExecuteTransport` delivers a wrapped command to the TPM and its output returns the result of the execution back to the user.

Transport sessions do not encrypt all components of the wrapped commands and some commands stipulate further exceptions as to the encryption of their parameters [22]. A schematic presentation of a transport session is given in Figure 3. **DATAw** in the figure denotes encryption of the components of the wrapped command with transEncKey, and Enc{secret,pk(K_{TS})} stands for asymmetric encryption of the transport session secret by the public key whose corresponding private key is K_{TS}.



Start auth session, load secret decryption key K_{TS}
Client
TPM_EsablishTransport_IN : encHandle || Enc(secret,pk(K_{TS})) || authHandle || keyAuth
TPM_EstablishTrasport_OUT : transHandle || transNonceEven || resAuth
Auth session setup
TPM_ExecuteTransport_IN : wrappedCmd, transHandle || transNonceOdd || transAuth
transEncKey = transNonceEven || transNonceOdd || "in" || secret
TPM_ExecuteTransport_OUT : wrappedRsp, transHandle || transNonceEven' || transAuth
transEncKey = transNonceEven' || transNonceOdd || "out" || secret
Command execution
TPM_ExecuteTransport_IN : wrappedCmd, transHandle || transNonceOdd || transAuth
TPM_ExecuteTransport_OUT : wrappedRsp, transHandle || transNonceEven' || transAuth
wrappedCmd = TAGw || LENw || ORDw || HANDLESw || DATAw || AUTHw
wrappedRsp = TAGw || LENw || RCw || HANDLESw || DATAw || AUTHw

Fig. 3. Encrypted transport session execution.

2.4 FS2PV Verification Framework

We utilize the FS2PV toolkit developed by Bhargavan et al. [8] for the verification of our F# implementation. FS2PV accepts programs written in a first-order subset of F# and outputs a model that can be analyzed with the automated theorem prover ProVerif [9].

The toolkit provides cryptographic, communication and other auxiliary libraries that programs need to use for cryptography, network communication, message composition and other operations. Each operation provided by the framework has dual *concrete* and *symbolic* implementations. Concrete libraries are used for compilation of an executable of the protocol, whereas the symbolic counterparts are employed in the ProVerif model extraction. In this work, we only utilize FS2PV's symbolic library and generate executable code with our F2C translator.

FS2PV's symbolic implementation of cryptographic and network operations encodes strong black-box assumptions on cryptography, which are assumed to be safe abstractions (in the style of Dolev and Yao [16]). To model cryptographically constructed byte arrays there is an algebraic datatype bytes equipped with symbolic constructors and pattern-matching to represent message manipulation and cryptographic primitives. For example, a value sym_encrypt(m, k) stands for a symmetric encryption of m with key k. An operation sym_decrypt unpacks such a value, and is defined by pattern-matching. There is no other way to obtain partial information about m.

Two substantial case studies using FS2PV are analyses of the Information Card [7] and TLS [6] protocols.

3 Reference Implementation and Analysis

Our F# implementation comprises four modules: TPM, TPM_data_structs, TPM_internal_structs, and TPM_command_structs. These modules represent TPM commands and data types. Client behaviour and auxiliary operations are specified in other modules. Our implementation makes use of FS2PV's symbolic library (Crypto, Net, Db) for cryptographic, network and state maintenance operations. We introduced small extensions to the library to include new data types used by the TPM.

Before embarking on verification, we first perform a symbolic execution of the implementation by generating an executable with the F# compiler. The code is appended with instructions that launch instances of the TPM and a client, e.g. as follows:

```
do Pi.fork(fun() -> Client())
do TPM()
```

Symbolic execution pretty-prints messages exchanged between the parties in the console. We found this facility of FS2PV invaluable in debugging the implementation code, as well as making sure that the formal verification is not carried out on vacuous models (that is, models that are trivially secure because no messages are accepted).

3.1 Formal analysis

We use the FS2PV/ProVerif toolchain to verify our reference implementation for authorization protocols OIAP and OSAP, and for the encrypted transport session protocol.

In our threat model, we assume there is an active attacker on the network between the client and the TPM that can intercept, manipulate and inject new messages constrained by the perfect cryptography assumption [16]. FS2PV framework formalizes the attacker as an arbitrary program that is able to call interfaces defined by our implementation code and the symbolic libraries.

The TPM and a client maintain state during authorization and transport sessions protocols, which stores latest session nonces, shared secrets, handles and other information. In our implementation of authorization sessions we have used databases from FS2PV's `Db` module to store the state data, which FS2PV translates into message passing over private channels.

The underlying verification engine ProVerif, however, encountered difficulties verifying and reconstructing attack traces for larger models that make use of private channels, so we had to tweak our handling of state information and the client code. In the encrypted transport session protocols, instead of using `Db`, we have extended the datatype `bytes` with private data constructors `TSD` and `CSD` to store the session state. In this approximation, the state data is wrapped with these constructors that the attacker cannot deconstruct and the state is output into and read in from the open network. This allowed ProVerif to reconstruct non-trivial attack traces. The verification of the corrected version of the encrypted transport session protocols, however, still did not succeed causing ProVerif to run out of memory (2GB) due to the amount of state information. We further approximated our model by allowing the attacker to access the client's state information, removing some of the client's integrity checks, and allowing transport session nonce generation function used by the TPM to produce free names instead of fresh ones (this is akin to having a faulty random number generator that produces known values). This allowed us to prove correctness of the fixed version of the encrypted transport session protocol, which implies correctness of the fix without such approximations.

We have expressed secrecy of weak authdata in the following form, so that the name used for authdata in the query file and the source code is propagated into ProVerif script generated by FS2PV:

```
(*** Declaration in an implementation file ***)
let ownerAuthData = Pi.name "broccoli"
let fOwnerAuthData = Fresh ownerAuthData

(*** Query file ***)
weaksecret ownerAuthData.

(*** Resulting declaration in the ProVerif script generated by fs2pv ***)
private free ownerAuthData.
 ...
weaksecret ownerAuthData.
```

Authorization sessions. Our verification of authorization protocols (OSAP and OIAP) found Chen and Ryan's attacks on the secrecy of weak authdata and produced attack traces.

Encrypted Transport Sessions. We have written a concrete F# implementation of the transport session protocol with the aim of formally verifying whether it can protect authorization sessions OIAP and OSAP against weak authdata attacks. The analysis highlighted an ambiguity in `TPM_ExecuteTransport` command's specification that had security implications in our analysis.

The specification of `TPM_ExecuteTransport` command states that for authorization session initiation commands `TPM_OSAP`, `TPM_OIAP` no parameters are encrypted in the request sent from the client, but it does not specify any caveats for the output sent back to the client. Therefore, following the specification of the command, the responses of `TPM_OSAP` and `TPM_OIAP` are sent back in encrypted form. Our analysis showed that in this case weak authdata leak is prevented for OSAP sessions, but not for OIAP sessions. Intuitively, OIAP sessions are not protected, since each command executed within OIAP produces an HMAC digest keyed on authdata, and all other data included in the digest are sent out in clear.

The specification of `TPM_OIAP` and `TPM_OSAP` commands, however, stipulates that no input or output parameters are encrypted when wrapped in a transport session. In this case, clearly wrapping OSAP sessions with encrypted transport protocol does not stop weak authdata leaks.

Our verification highlighted the following simple amendments to the encrypted transport protocol that will protect the weak authdata:

– for the wrapped OSAP session, encrypt nonceEvenOSAP sent to the client in the response of `TPM_OSAP` command;
– for the wrapped OIAP session, encrypt the rolling nonceEven in each response of a command executed in the session.

These amendments correspond to including nonces in the **DATAw** component in Figure 3, instead of sending them out in clear as part of **AUTHw**. The verification proved correctness of these fixes.

Our analysis also revealed a potential weakness in the encrypted transport protocol when a key of type `TPM_KEY_LEGACY` is used as an encryption key for the transport session's secret. If such key is not used with RSA OAEP encryption scheme and has a weak authdata associated with it, then an attacker can acquire the transport session secret by invoking the `TPM_UnBind` command, which decrypts the secret without checking decrypted message structure. To avoid this weakness, a client needs to ensure that a high-entropy authdata is chosen for the transport secret encryption key, or choose a key that is either a storage key (`TPM_KEY_STORAGE`) or is used with RSA OAEP encryption. The TPM commands specification states that use of the key type `TPM_KEY_LEGACY` in general is not recommended, and our finding corroborates the recommendation.

3.2 F2C: translation into an executable C specification

We have implemented a tool that generates executable C code from the TPM implementation written in an F# fragment. It is developed on top of FS2PV's library that builds the AST of the F# input code. We apply the translator to TPM, TPM_data_structs, TPM_internal_structs, TPM_command_structs modules to generate executable C code for commands and data structures of the TPM.

Figure 5 below shows a sample C code generated from the corresponding F# code in Figure 4.

```
let TPM_OSAP (input:TPM_OSAP_IN) : TPM_OSAP_OUT =
    if (input.tag_osapIn = TPM_TAG_RQU_COMMAND) then
      if (input.ordinal_osapIn = TPM_ORD_OSAP) then begin
        let nonceEven : TPM_NONCE = mkNonce() in
        let nonceEvenOSAP : TPM_NONCE = mkNonce() in
        let xNonceOddOSAP : TPM_NONCE = input.nonceOddOSAP_osapIn in
        let hmac_data : BYTES = dconcatSK nonceEvenOSAP xNonceOddOSAP in
        let handle : TPM_AUTHHANDLE = allocHandle() in
        let entityType : TPM_ENTITY_TYPE = input.entityType_osapIn in
        if (entityType=TPM_ET_OWNER) then begin
            let pd : TPM_PERMANENT_DATA = loadPermData() in
            let authData : TPM_SECRET = pd.ownerAuth in
            let tsharedSecret : TPM_HMAC =
                tpm_hmacsha1((key authData.digest),hmac_data,sizeof(hmac_data)) in
            let s1:TPM_SESSION_DATA = {
                sHandle=handle;
                pid=TPM_PID_OSAP;
                nonceEven=nonceEven;
                sharedSecret=tsharedSecret;
                entityValue = input.entityValue_osapIn
            } in saveState s1;
            let res : TPM_OSAP_OUT = {
                tag_osapOut = TPM_TAG_RSP_COMMAND;
                paramSize_osapOut = UINT32 0;
                returnCode_osapOut = TPM_SUCCESS;
                authHandle_osapOut = handle;
                nonceEven_osapOut = nonceEven;
                nonceEvenOSAP_osapOut = nonceEvenOSAP } in
            setSize(res,res.paramSize_osapOut); res
        end else failwith (string TPM_FAIL);
      end else failwith (string TPM_BAD_ORDINAL)
    else failwith (string TPM_FAIL)
```

Fig. 4. Sample TPM command code (fTPM.fs).

```
TPM_OSAP_OUT TPM_OSAP (TPM_OSAP_IN input) {
  if (input.tag_osapIn == TPM_TAG_RQU_COMMAND) {
    if (input.ordinal_osapIn == TPM_ORD_OSAP) {
      TPM_NONCE nonceEven = mkNonce();
      TPM_NONCE nonceEvenOSAP = mkNonce();
      TPM_NONCE xNonceOddOSAP = input.nonceOddOSAP_osapIn;
      struct {TPM_NONCE nonceEvenOSAP_sf; TPM_NONCE xNonceOddOSAP_sf;}
        hmac_data_st={nonceEvenOSAP, xNonceOddOSAP};
      BYTES hmac_data = (BYTE*)& hmac_data_st;
      TPM_AUTHHANDLE handle = allocHandle();
      TPM_ENTITY_TYPE entityType = input.entityType_osapIn;
      if (entityType == TPM_ET_OWNER) {
        TPM_PERMANENT_DATA pd = loadPermData();
        TPM_SECRET authData = pd.ownerAuth;
        TPM_HMAC tsharedSecret = tpm_hmacsha1(authData.digest,hmac_data,sizeof(hmac_data_st));
        TPM_SESSION_DATA s1 = {0,nonceEven,TPM_PID_OSAP,handle,tsharedSecret};
        saveState(s1);
        TPM_OSAP_OUT res = {handle,nonceEvenOSAP,nonceEven,0,TPM_SUCCESS,TPM_TAG_RSP_COMMAND};
        res.paramSize_osapOut=sizeof(res);
        return res;
      }
      else {mkErrorReturn_TPM_OSAP(TPM_FAIL);}
    }
    else {mkErrorReturn_TPM_OSAP(TPM_BAD_ORDINAL);}
  }
  else {mkErrorReturn_TPM_OSAP(TPM_FAIL);}
}
```

Fig. 5. Generated sample C code.

	LoC F#	LoC C	Verification of the secrecy of weak auhdata
OIAP session	250	260	attack found
OSAP session	265	275	attack found
OIAP wrapped in an encrypted transport session	780	660	attack found
OSAP wrapped in an encrypted transport session	800	680	attack found if `TPM_OSAP` command response is not encrypted
OIAP, OSAP wrapped in the amended encrypted transport session	800	680	correctness proved

Table 1. Summary of the verification results.

4 Conclusion and Future Work

We have implemented concrete F# specifications for OIAP, OSAP and encrypted transport session protocols and were able to perform their formal verification using FS2PV, as well as to generate corresponding C specification with F2C. For authorization protocols, we have implemented a TPM client in C++ to demonstrate executability of the generated C specification. In our formal analysis, we have captured the weak authdata attacks on the authorization sessions and verified that our enhancements to the encrypted transport sessions thwart such attacks.

It would be desirable to demonstrate interoperability between our reference implementation and existing software for communicating with a TPM; this has not yet been attempted. Another future direction is to develop our tools to the point where they could be used to define a reference implementation for the whole TPM or some other cryptographic token. Our results so far suggest this goal is probably within reach with the current generation of verification tools.

Acknowledgements The research described in this paper was performed during Aybek Mukhamedov's internship at Microsoft Research. We thank Karthik Bhargavan, Johannes Borgström, and Cédric Fournet for useful discussions about this work. We also thank Paul England and David Wooten, Microsoft representatives on the Trusted Computing Group, for their support and advice.

References

1. AVISPA Tool Set. http://www.avispa-project.org/.
2. ISO/IEC 9797-2: Information technology – Security techniques – Message authentication codes (MACs) – Part 2: Mechanisms using a dedicated hash-function.
3. ISO/IEC PAS DIS 11889: Information technology – Security techniques – Trusted platform module.
4. Ross Anderson. Trusted Computing FAQ. http://www.cl.cam.ac.uk/rja14/tcpa-faq.html, 2003.
5. O. Berkman and O. M. Ostrovsky. The unbearable lightness of pin cracking. In *Financial Cryptography and Data Security*, Trinidad and Tobago, 2007.
6. K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for tls. In *CCS'08*, 2008.

7. K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy. Verified implementations of the Information Card federated identity-management protocol. In *ASIACCS'08*, 2008.
8. K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, 2006.
9. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.
10. Mike Bond. *Understanding Security APIs.* PhD thesis, University of Cambridge, 2005.
11. Mike Bond and Ross Anderson. API-level attacks on embedded systems. *Computer*, 34(10):67–75, 2001.
12. Danilo Bruschi, Lorenzo Cavallaro, Andrea Lanzi, and Mattia Monga. Replay attack in TCG specification and solution. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 127–137, Washington, DC, USA, 2005. IEEE Computer Society.
13. L. Chen and M. D. Ryan. Offline dictionary attack on TCG TPM weak authorisation data, and solution. In D. Grawrock, H. Reimer, A. Sadeghi, and C. Vishik, editors, *Future of Trust in Computing.* Vieweg & Teubner, 2008.
14. Véronique Cortier, Gavin Keighren, and Graham Steel. Automatic analysis of the security of xor-based key management schemes. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 538–552. Springer, 2007.
15. Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11. In *CSF*, pages 331–344. IEEE Computer Society, 2008.
16. D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.
17. Michael Goldsmith. FDR2 User's Manual version 2.82. Formal Systems (Europe) Ltd., 2005.
18. Sigrid Gürgens, Carsten Rudolph, Dirk Scheuermann, Marion Atts, and Rainer Plaga. Security evaluation of scenarios based on the TCG's TPM specification. In *ESORICS*, pages 438–453, 2007.
19. Amerson H. Lin. Automated Analysis of Security APIs. Master's thesis, MIT, 2005. http://sdg.csail.mit.edu/pubs/theses/amerson-masters.pdf.
20. William McCune. OTTER 3.3 Reference Manual. Aragonne National Laboratory, 2003.
21. D. Syme, A. Granicz, and A. Cisternino. *Expert F#.* Apress, 2007.
22. Trusted Computing Group. TPM Specification version 1.2. Parts 1–3. www.trustedcomputinggroup.org/specs/TPM/, 2007.