Towards a Formal Security Analysis of the SEVECOM API

Graham Steel

LSV, INRIA & CNRS & ENS-Cachan

1 Introduction

SEVECOM [1] was an EU 6th Framework project aimed at defining "a consistent and future-proof solution to the problem of Vehicular communications VC [Vehicular communications]/IVC [Inter-Vehicular Communications] security". An early conclusion of the project was that the solution must include the use of a tamper-resistant cryptographic device on board the vehicle. Indeed, the design of such a device has been a common conclusion of much recent research in the area [9]. A feature of the SEVECOM project is that it defines the API for the tamper resistant device (TRD). This is a security-critical part of the design, since the API regulates how applications on the vehicle (which in the worst case may include malware, viruses, applications deliberately placed in the system by crooked vehicle owners, etc.) access the secret key material stored in the TRD. Flaws in the designs of previous APIs for similar devices in the ATM (cash machine) network and for cryptographic USB key tokens have been shown to lead to catastrophic attacks, whereby sensitive keys are revealed in the clear [2, 3].

In the last few years, we have been researching formal techniques for analysing such interfaces and checking them for flaws. This has resulted in the detection of a number of previously unknown flaws, as well as positive results of assurance of the absence of certain classes of attack on revised designs [4, 6, 10]. In this paper, we describe the first results of the application of our techniques to the SEVECOM API. In doing so, we aim to make clear the security policy the device is intended to implement, and to investigate the extent to which it achieves this. We find that some, but not all, of the stated security goals are achieved by the design, and we show how the shortcomings can be exploited. We discuss changes to the API which prevent the exploit.

The paper is structured as follows: first we describe the operation of the SEVECOM API and the goals it sets out to achieve, in section 2. Then we describe our formal modelling approach in section 3, showing how we formalised the goals and the threat model, and explaining our abstractions. We give our results obtained using the SAT-based model checker SATMC in section 4. We describe our plans for further analysis and conclude in section 5.

2 The Sevecom API

The design of the SEVECOM is described in the 'Baseline Architecture' document [8]. The SEVECOM TRD is assumed to support random number generation, asymmetric cryptogra-

[1] http://www.sevecom.org/

phy and hashing. The API of the TRD provides a variety of services. Primarily it provides a decryption, signing and timestamping service. The API also provides key management services to back up these functions. This includes key generation, and revocation and update of keys based on messages received from a trusted authority. We summarise the specification below, concentrating on the aspects we have covered in our security analysis. We will first describe the data stored by the TRD, then the commands of the API, and then the protocols these API functions are designed to support.

2.1 Data Stored by the TRD

Cryptographic keys stored in the TRD are divided into five types based on their roles.

1. Short term signature generation keys, used to authenticate short-term pseudonyms used by applications running on the vehicle
2. Short term decryption keys, used to decrypt messages intended for applications running on the vehicle
3. A long-term signature generation key, used to authenticate the true identity of vehicle
4. A long-term decryption key, used to decrypt encrypted messages intended for the vehicle
5. Two-long term public root keys used to verify the authenticity of commands sent by the authorities to the TRD.

Each key is stored in memory along with the following data

1. The key identifier
2. The key value
3. A reference to the associated cryptographic algorithm and associated parameters for the key
4. Key flags: key type (one of the five above), whether the key is removable (i.e. deletable, not exportable) from the TRD, and whether the key is currently under update.
5. Lock counter

In addition to the keys, the TRD also stores a 128 bit device identifier, that is intended to be set e.g. by the vehicle manufacturer during the initialisation of the unit. This value is used later used during long term key update or to disable the TRD.

2.2 API Commands

The API contains a single command for decryption, and a single command for signing. To use the commands, the application program provides the key identifier, and the data to be signed or decrypted. The command succeeds only if the key referenced is of a suitable type. Apart from this, the API is agnostic about what it signs or attempts to decrypt. However, during signing, a timestamp is appended to all messages using the TRD's realtime clock.

The private keys used for decryption and signing never leave the device - the keypairs are generated by the TRD, and the public halves are exported to the host OBU. There is a single generation command, to which the application supplies parameters which implicitly specify the type.

In addition to these commands, the API offers commands supporting protocols for updating the long term keys. We describe these in the relevant sections below.

2.3 Pseudonym Update Protocol

The main intended use of the short-term signature generation keys is to authenticate the privacy-preserving pseudonyms used by the vehicle in its beacon signals. To obtain a new set of pseudonyms, software running on the vehicle calls the TRD API to ask that new short-term signing key pairs are generated. The private halves are stored on the TRD, and the public halves are sent to the OBU. The OBU contacts the pseudonym provider (PP) to obtain a certificate for each pseudonym. Some details of the protocol are given in the specification [8, §4.2.4]: it should be executed on an 'authenticated and confidential' communication link. A clue as to how this link can be established is given [8, §7.3.4]: "certificate requests can be passed back to the TRD for being authenticated, typically with the long-term master signature generation key of the TRD. But again, this is transparent to the TRD, because certificate requests are treated as any other data to be signed by the TRD."

Once the authenticated and confidential channel is established, the protocol includes a simple challenge-response from the PP to the OBU - the PP sends a fresh nonce to the OBU, which must response by signing each nonce with the private half of the pseudonym key stored in the TRD. Once these responses have been checked, the PP replies with the certificates for the pseudonyms.

2.4 Long Term Key Update Protocol

This protocol is used to update the long term signing or long term decryption key. First, the application programme calls the API command 'InitLongTermKeyUpdate', giving the handle of one of the long term keys. This results in the generation of a new keypair, the private half of which is stored, and the public half of which is send as output. The application software sends the public key off to be stored by the operator. The returned package contains a signed hash of a tag specifying the purpose of the message (long term key update), the public key, the device identifier, and a timestamp. This is given as input to the FinaliseLTKUpdateCommand, which builds its own hash and checks it against the signed one. If it is correct, the new private key is activated and the old one expelled.

2.5 Root Key Update

A new root key is only accepted when one of the two existing keys has been revoked. The decision to revoke a key is taken by the operator. First, a revocation message is sent to the vehicle. This consists of one of the root public keys signed by its own private half, along with a command tag giving the purpose of the message. This is given as input to the revokeRootKey command. The TRD checks that it has two unrevoked root keys, checks the signature, and if it verifies ok, it revokes the corresponding public key, and moves into a 'one key' state. To insert a new key, the operator sends the public half of the new key signed by the private half of the unrevoked root key. This the vehicle gives to the TRD as input to the setRootKey command. The TRD checks the signature, checks that it is in a 'one key' state, i.e. that it has revoked the other root key, and then stores the new root key and returns to its 'two keys' state.

2.6 Other Protocols

There are several other protocols proposed in the Sevecom specification, including protocols for initialising the TRD, disabling the TRD, and updating the real time clock embedded in the TRD. Each has associated with it specific commands in the API. We have not yet modelled these protocols, hence we omit the details.

2.7 Summary of Security Goals

There is no explicit statement of a security policy for the API in the specification. However, there are several properties that we can straightforwardly infer, and in addition, the sections describing protocols often list security goals that the protocol coupled with the API should achieve. From these we extracted the following security properties that we investigated in our formal work. We will show how we specified them formally in section 3.

P1: Private key halves unknown For all asymmetric keypairs managed by the device, the private half remains unknown to the intruder.

P2: Authenticity of pseudonyms If the device receives a certificate for a pseudonym, then the private half of that pseudonym remains unknown to the intruder.

Property **P1** may at first seem trivial, since when a key is generated, only the public halves are exported, and there is no command to allow a private half to be exported. However, this property also covers attacks whereby the intruder is able to inject a key of his own into the device, for example by abusing the long-term key update protocol.

Property **P2** is inferred from the description of the use of pseudonyms in the beaconing protocol [8, 5.2.1]: 'we infer the sender is actually a valid participant of the network (e.g., a vehicle, RSU, traffic sign, etc.)' and 'the identified sender has sent the message, not another one'. Both these properties are asserted to be a consequence of the use of a valid certified pseudonym.

3 Formal Modelling

Our modelling of the API follows the approach we used for the widely-used RSA standard API PKCS#11 [6]. The idea is to model the device as being under the complete control of an intruder, representing a malicious piece of software running on the OBU. The intruder can call the commands of the API in any order he likes using any values that he knows. We abstract away details of the cryptographic algorithms in use, following the classical approach of Dolev and Yao [7]. Bitstrings are abstracted to terms in an abstract algebra and the rules of the API and the abilities of an attacker are written as deduction rules in the algebra. The intruder is assumed not to be able to crack the encryption algorithm by brute-force search or similar means, thus he can only read an encrypted message if he has the correct key.

The language we use for the model is the 'intermediate format' which stems from the AVISPA security protocol analysis research project. It is based on multiset rewriting and defined formally in a project deliverable [11]. We give just the basic notions here. Terms in the model are built from variables $\mathsf{x}, \mathsf{y}, \mathsf{z}$ and function symbols $\{\!|\,.\,|\!\}_.$ ($\{\!|\mathsf{x}|\!\}_{\mathsf{y}}$ represents asymmetric encryption of plaintext x under key y), $\mathsf{inv}(.)$ ($\mathsf{inv}(\mathsf{x})$ is used to model the other half of the asymmetric keypair $\mathsf{x}, \mathsf{inv}(\mathsf{x})$) and a set of function symbols of arity zero, which we will call constants, $\mathsf{ki}, \mathsf{sign1}, \ldots$. This includes the Boolean constants true ($\top$) and false ($\bot$).

The semantics of the model is defined in terms of a transition system. Each state in the model consists of a set of terms in the intruder's knowledge, and a set of global state predicates. The intruder's knowledge increases monotonically with each transition, but the global state is non-monotonic. The transition system may contain loops in the global state.

Example 1 *As a first simple example, we give the transition rules for the intruder's abilities to encrypt and decrypt*

$$\begin{array}{rcll} x, y & \rightarrow & \{\!|\mathsf{x}|\!\}_{\mathsf{y}} & \textit{encryption} \\ \{\!|\mathsf{x}|\!\}_{\mathsf{y}}, y & \rightarrow & x & \textit{decryption} \end{array}$$

Note that there are no global state facts in these rules - the intruder can carry out these operations no matter what the state of the device.

We use the global state facts to model the key storage of the TRD. For each key stored, the global state contains a keyrec fact with the following format:

$$\mathsf{keyrec}(\mathsf{ID}, \mathsf{Value}, \mathsf{decrypt}, \mathsf{sign}, \mathsf{verify}, \mathsf{removable}, \mathsf{under\_update})$$

The fields ID and Value are of type $\mathsf{message}$, i.e. they range over all the terms in the model. The other fields are Boolean: $\mathsf{decrypt}, \mathsf{sign}, \mathsf{verify}$ indicate the permissions set on the key, $\mathsf{removable}$ indicates whether a key is a long term (non-removable) key or a short term (removable) one, and the $\mathsf{under\_update}$ field indicates whether a particular key in undergoing update (in the case of the root signing keys, it models the fact that the key has been revoked).

Example 2 *To see how the global state facts work, consider the rule below, which models the decrypt command. The rule fires only when there is a global state fact matching the fact on the left hand side of the rule. We use a semi-colon to separate the intruder knowledge part from the global state part in our rules.*

$$\mathsf{Decrypt}: \quad \{\!|\mathsf{x}|\!\}_{\mathsf{y}}; \mathsf{keyrec}(\mathsf{z1}, \mathsf{y}, \top, \bot, \bot, \mathsf{z2}, \bot) \quad \rightarrow \quad \mathsf{x}$$

Some rules in the model allow the TRD to generate fresh keys. These are modelled by a finite pool of available fresh terms. Our model also contains a global counter time that is used to count the number of fresh keys that have been generated. This allows us to select a fresh name each time using the predicate fresh, as illustrated in the example below.

Example 3

$$\begin{array}{lcl} \mathsf{GenerateSign} \quad ; \mathsf{time}(\mathsf{x}), \mathsf{fresh}(\mathsf{x}, \mathsf{y}) & \rightarrow & \mathsf{y}; \mathsf{keyrec}(\mathsf{s}(\mathsf{x}), \mathsf{inv}(\mathsf{y}), \bot, \top, \bot, \top, \bot), \mathsf{time}(\mathsf{s}(\mathsf{x})), \\ & & \neg\mathsf{fresh}(\mathsf{x}, \mathsf{y}) \end{array}$$

Note that the global time is advanced using the successor function $\mathsf{s}()$.

The goals of section 2.7 are specified in the model as reachability *queries*.

Definition 1 *A* query *is a pair (T, L) where T is a set of terms and L a set of global facts, either of which may contain variables.*

Intuitively, a query (T, L) is satisfied if there exists a substitution θ such that we can reach a state where the adversary knows all terms in $T\theta$ and all literals in $L\theta$ are evaluated to true. For example, to model the property **P1**, we use the query $(\mathsf{x}, \mathsf{keyrec}(\mathsf{y}, \mathsf{x}, \mathsf{z1}, \mathsf{z2}, \perp, \perp, \mathsf{z4})$, i.e. we demand whether it is possible for the intruder to know the value of a private key which is stored on the device.

In Figure 1 we give the full set of rules we used to model the API. To model the storage of a candidate key during long term key update, we use the predicate ready. In addition, our model contains the rule in Figure 2 for modelling the response from the pseudonym provider. We have simplified the protocols in this initial version of our model, removing the nonce handshake from the pseudonym protocol. The PP simply provides a response for any request signed by the TRD's long term key.

4 Results

We now give our results obtained using the SAT-based model checker SATMC [1]. We took each of the properties from section 2.7 and investigated them with a variety of scenarios in terms of intruder knowledge. Our results are summarised in Table 1. All model files are available for download[2]. We will explain each experiment in turn:

Experiment 1 In a scenario where the intruder starts with only his own public-private keypair, we test to see if he is able to obtain the value of a key stored on the device. We limit the number of fresh keypairs the device will generate to 3. After about 5000 seconds, SATMC terminates indicating no attack is possible in this scenario.

Experiment 2 We repeat experiment 1 and additionally give the intruder the private value half of one of the root keys stored on the device. Again no attack is possible.

Experiment 3 We give the intruder the private half of both root keys, and he is finally able to inject his own key onto the device. This is not a worrisome attack, since it is considered 'out of scope' by the designers. They recommend strong steps to prevent this scenario from occurring, e.g. storing the two private halves in different locations in secure TRDs. We include this experiment only to validate our model.

Experiment 4 We attempt to verify property 2, that is that a valid pseudonym certificate indicates that the private half of the key is stored on the device. This turns out no to be true: an intruder can obtain a certificate for an arbitrary public key for which he has the private half, since the sign command allows him to sign anything with the device's long term signing key, making it appear authentic to the PP. Note this is not due to the simplifications to the pseudonym protocol we made for our model (Figure 2) - the result would be the same with the full protocol implemented.

Experiment 5 We propose a change to the API whereby when a short-term key is generated, the device additionally gives an HMAC of the public key and the device's

[2]`http://www.lsv.ens-cachan.fr/~steel/sevecom`

$$\begin{array}{l}
\mathsf{GenerateSign}: \\
;\mathsf{time}(x), \mathsf{fresh}(x, y) \quad \rightarrow \quad y; \mathsf{keyrec}(s(x), \mathsf{inv}(y), \bot, \top, \bot, \top, \bot), \mathsf{time}(s(x)), \neg\mathsf{fresh}(x, y) \\
\mathsf{GenerateDecrypt}: \\
;\mathsf{time}(x), \mathsf{fresh}(x, y) \quad \rightarrow \quad y; \mathsf{keyrec}(s(x), \mathsf{inv}(y), \top, \bot, \bot, \top, \bot), \mathsf{time}(s(x)), \neg\mathsf{fresh}(x, y)
\end{array}$$

$$\begin{array}{lrcl}
\mathsf{Decrypt}: & \{\!|x|\!\}_y; \mathsf{keyrec}(z1, \mathsf{inv}(y), \top, \bot, \bot, z2, \bot) & \rightarrow & x \\
\mathsf{Sign}: & x; \mathsf{keyrec}(z1, \mathsf{inv}(y), \bot, \top, \bot, z2, \bot) & \rightarrow & \{\!|x|\!\}_{\mathsf{inv}(y)}
\end{array}$$

$$\begin{array}{lrcl}
\mathsf{RevokeRootKey1}: & \{\!|x|\!\}_{\mathsf{inv}(x)}; \mathsf{keyrec}(1, x, \bot, \bot, \top, \bot, \bot), & \rightarrow & \mathsf{keyrec}(1, x, \bot, \bot, \top, \bot, \top) \\
 & \mathsf{keyrec}(2, y, \bot, \bot, \top, \bot, \bot) & &
\end{array}$$

$$\begin{array}{lrcl}
\mathsf{RevokeRootKey2}: & \{\!|y|\!\}_{\mathsf{inv}(y)}; \mathsf{keyrec}(1, x, \bot, \bot, \top, \bot, \bot), & \rightarrow & \mathsf{keyrec}(2, x, \bot, \bot, \top, \bot, \top) \\
 & \mathsf{keyrec}(2, y, \bot, \bot, \top, \bot, \bot) & &
\end{array}$$

$$\begin{array}{lrcl}
\mathsf{UpdateRootKey1}: & \{\!|z|\!\}_{\mathsf{inv}(x)}; \mathsf{keyrec}(2, x, \bot, \bot, \top, \bot, \bot), & \rightarrow & \mathsf{keyrec}(1, z, \bot, \bot, \top, \bot, \bot) \\
 & \mathsf{keyrec}(1, y, \bot, \bot, \top, \bot, \top) & &
\end{array}$$

$$\begin{array}{lrcl}
\mathsf{UpdateRootKey2}: & \{\!|z|\!\}_{\mathsf{inv}(x)}; \mathsf{keyrec}(1, x, \bot, \bot, \top, \bot, \bot), & \rightarrow & \mathsf{keyrec}(2, z, \bot, \bot, \top, \bot, \bot) \\
 & \mathsf{keyrec}(2, y, \bot, \bot, \top, \bot, \top) & &
\end{array}$$

$$\begin{array}{rcl}
\mathsf{InitialiseLTKUpdate}: & & \\
\mathsf{time}(xt); \mathsf{fresh}(xk2), & \rightarrow & xk2; \mathsf{ready}(x, \mathsf{inv}(xk2)), \mathsf{time}(s(xt)) \\
\mathsf{keyrec}(x, \mathsf{inv}(xk1), y, x, \bot, \bot, \bot) & & \mathsf{keyrec}(x, \mathsf{inv}(xk1), y, x, \bot, \bot, \top)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{FinaliseLTKUpdate}: & & \\
\{\!|xk2|\!\}_{\mathsf{inv}(xk3)}; \mathsf{keyrec}(x, \mathsf{inv}(xk1), y, z, \bot, \bot, \top) & \rightarrow & \mathsf{keyrec}(x, \mathsf{inv}(xk2), y, z, \bot, \bot, \bot) \\
\mathsf{ready}(x, \mathsf{inv}(xk2)), \mathsf{keyrec}(x1, xk3, \bot, \bot, \top, \bot, \bot) & &
\end{array}$$

Figure 1: Rules modelling the SEVECOM API

$$\begin{array}{rcl}
\mathsf{CertifyPseudonym}: & & \\
\{\!|xk2|\!\}_{\mathsf{inv}(xk1}; \mathsf{keyrec}(x, \mathsf{inv}(xk1), \bot, \top, \bot, \bot, \bot), & \rightarrow & \{\!|xk2|\!\}_{\mathsf{inv}(xk3)} \\
\mathsf{keyrec}(z, xk3, \bot, \bot, \top, \bot, \bot) & &
\end{array}$$

Figure 2: Rule modelling pseudonym certification

Experiment	Property	Scenario	Attack?	Time
1	P1	Intruder has his own keypair	No	5000s
2	P1	Intruder has own keypair and one root private key	No	5000s
3	P1	Intruder has own keypair and both root keys	Yes	5sec
4	P2	Intruder has own keypair	Yes	1.1sec
5	P2	Intruder has own keypair, API patched (see text)	No	68s

Table 1: Summary of results

$$\begin{array}{lcl}
\mathsf{GenerateSign:} & & \\
;\mathsf{time}(\mathsf{x}),\mathsf{fresh}(\mathsf{x},\mathsf{y}) & \rightarrow & \mathsf{y},\mathsf{h}(\mathsf{id},\mathsf{y});\mathsf{keyrec}(\mathsf{s}(\mathsf{x}),\mathsf{inv}(\mathsf{y}),\bot,\top,\bot,\top,\bot),\mathsf{time}(\mathsf{s}(\mathsf{x})) \\
\mathsf{GenerateDecrypt:} & & \\
;\mathsf{time}(\mathsf{x}),\mathsf{fresh}(\mathsf{x},\mathsf{y}) & \rightarrow & \mathsf{y},\mathsf{h}(\mathsf{id},\mathsf{y});\mathsf{keyrec}(\mathsf{s}(\mathsf{x}),\mathsf{inv}(\mathsf{y}),\top,\bot,\bot,\top,\bot),\mathsf{time}(\mathsf{s}(\mathsf{x}))
\end{array}$$

Figure 3: Modified Generation Rules, using hash function $\mathsf{h}()$ and unique device identification number id to model the use of an HMAC.

unique identifier, which was set at initialisation time (see Figure 3). This id is secret, hence hence be used to construct a MAC. We alter the pseudonym protocol so that the PP will only provide certificates for keys which arrive along with a corresponding HMAC (this requires the pseudonym provider to have a list of valid vehicle IDs, which it must keep secret). The property P2 is now verified in our model.

The seriousness of the attack discovered in experiment 4 depends of course on the details of the deployment. The specification states explicitly the properties required of the pseudonym by the beaconing protocol, including the fact that the pseudonym should guarantee that 'the sender is actually a valid participant of the network'. It further mentions that details of the owners of pseudonyms could be turned over e.g. to law enforcement agencies. An attacker might exploit this vulnerability by e.g. obtaining a set of certificates for a stolen vehicle for which he knows the private half of the keys, and then using these pseudonyms in another vehicle, impersonating the first one. It seems interesting at least to consider how to patch the API to avoid this.

The practicality of our particular proposal for a fix depends on the pseudonym providers being able to securely manage the vehicle IDs. In the original design, they only need to store the public keys (and certificates) of valid vehicles, in order to establish an authenticated channel.

5 Conclusions

The first results of our formal analysis of the Sevecom API are already intriguing: the long term key update commands seem to keep the private keys secure, but it is possible for attacker able to introduce malicious code running on to the OBU to obtain certificates for pseudonyms for which he holds the private halves, allowing impersonation or cloning of a vehicle's identity. We have suggested a change to the API that prevents this, but it places heavier demands on the pseudonym provider. We feel that our results validate the approach of using Dolev-Yao style models to analyse proposals for on-vehicle TRD APIs, even if the SEVECOM API design is not taken forward into future projects.

These are preliminary results, and at present our model has many limitations. These will be the focus of our immediate further work. First, we are using an abstract representation of cryptography, which ignores fine details of the cryptographic algorithms, though there has been much recent work in relating these abstract models to real cryptographic algorithms (e.g. [5]). Additionally, we are assuming that only a small number of fresh keys are generated by the device. To be able to conclude that security in this small model implies security in a model with unbounded number of fresh keys will require more theoretical work, though we are hopeful that the approach we developed for RSA PKCS#11 will apply [10]. Our model only accounts for one vehicle, so we cannot investigate a class of attacks where an intruder obtains signals from more than one vehicle. Furthermore, we have investigated only two security properties of the API. More subtle properties based, for example, on the correctness of the timestamps given in the signatures, remain to be investigated. Finally, we need to analyse the remaining protocols for intialisation, disabling, and clock synchronisation, and their associated API commands.

References

[1] A. Armando and L. Compagna. SAT-based model-checking for security protocols analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008. Software available at `http://www.ai-lab.it/satmc`. Currently developed under the AVANTSSAR project, `http://www.avantssar.eu`.

[2] M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, pages 67–75, October 2001.

[3] J. Clulow. On the security of PKCS#11. In *Proceedings of CHES 2003*, pages 411–425, 2003.

[4] V. Cortier, G. Keighren, and G. Steel. Automatic analysis of the security of XOR-based key management schemes. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 4424 in LNCS, pages 538–552, 2007.

[5] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *14th European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 157–171, Edinburgh, UK, 2005. Springer.

[6] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[7] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.

[8] F. Kargl (Ed). Sevecom baseline architecture. Deliverable for EU Project Sevecom, 2009. D2.1-App.A.

[9] Cryptographic hardware for cars. ESCAR Panel Discussion, November 2007.

[10] S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In P. Degano and L. Viganò, editors, *Preliminary Proceedings of the Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, volume 5511 of *Lecture Notes in Computer Science*, pages 92–106, York, UK, 2009. Springer. To appear.

[11] AVISPA Project. Deliverable 2.3: The intermediate format. Available from `http://www.avispa-project.org/delivs/2.3`.