Runtime Verification of Cryptographic Protocols

Andreas Bauer [a] and Jan Jürjens [b,*]
[a] *NICTA, and Australian National University*
[b] *Department of Computer Science, TU Dortmund (Germany)*

Abstract

There has been a significant amount of work devoted to the static verification of security protocol designs. Virtually all of these results, when applied to an actual implementation of a security protocol, rely on certain implicit assumptions on the implementation (for example, that the cryptographic checks that according to the design have to be performed by the protocol participants are carried out correctly). So far there seems to be no approach that would enforce these implicit assumptions for a given implementation of a security protocol (in particular regarding legacy implementations which have not been developed with formal verification in mind).

In this paper, we use a code assurance technique called "runtime verification" to solve this open problem. Runtime verification determines whether or not the behaviour observed during the execution of a system matches a given formal specification of a "reference behaviour". By applying runtime verification to an implementation of any of the participants of a security protocol, we can make sure during the execution of that implementation that the implicit assumptions that had to be made to ensure the security of the overall protocol will be fulfilled. The overall assurance process then proceeds in two steps: First, a design model of the security protocol in UML is verified against security properties such as secrecy of data. Second, the implicit assumptions on the protocol participants are derived from the design model, formalised in linear-time temporal logic, and the validity of these formulae at runtime is monitored using runtime verification. The aim is to increase one's confidence that statically verified properties are satisfied not only by a model of the system, but also by the actual running system itself. We demonstrate the approach at the hand of the open source implementation Jessie of the de-facto Internet security protocol standard SSL. We also briefly explain how to transfer the results to the SSL-implementation within the Java Secure Sockets Extension (JSSE) recently made open source by Sun Microsystems.

Key words: Security protocols, SSL, Java, temporal logic, static verification, runtime verification, security automata.

1. Introduction

With respect to cryptographic protocols (or short, crypto-protocols), a lot of successful work has been done to formally analyse abstract specifications of these protocols for security design weaknesses [26, 19, 40, 1, 14, 47, 7, 6]. What is still largely missing is an approach which provides assurance for *implementations* of crypto-protocols against security weaknesses. This is necessitated by the fact that so far, crypto-protocols are usually not generated automatically from formal specifications. So even where the corresponding specifications are formally verified, the implementations may still contain vulnerabilities related to the insecure use of cryptographic algorithms, or by some underlying assumptions in the abstract models that are invalidated by reality.

For example, recently a security bug was discovered in OpenSSL where several functions inside OpenSSL incorrectly checked the result from calling a signature verification function, thereby allowing a malformed signature to be (wrongly) treated as a good signature [1]. This was a serious vulnerability because it would enable an attacker in a "man in the middle" attack scenario (which is generally realistic on the Internet, which is why security protocols are needed in the first place) to present a malformed SSL/TLS signature to a client suffering from this vulnerability, who would wrongly accept it as valid.

The above vulnerability thus gives a motivating example for a scenario where checks performed at runtime can be indispensable for the secure functioning of a given crypto-protocol implementation. The goal of our work is to offer a formally based approach which is integrated with existing design models of the system (e. g., in UML), and which would allow the runtime assurance of properties supporting general security requirements (such as secrecy and authenticity), rather than being

* Corresponding author. Partially supported by the EU project and SecureChange (ICT-FET-231101).

Email addresses: baueran@rsise.anu.edu.au (Andreas Bauer), http://jurjens.de/jan/ (Jan Jürjens).

[1] Cf. http://www.openssl.org/news/secadv_20090107.txt.

a quick fix for a particular known vulnerability. More specifically, the approach is based on the combination of:

– static security verification of UML specifications of crypto-protocols against security requirements using the UMLsec tool-support [48], which implements a Dolev-Yao style security analysis [26] [2] and
– a technique called *runtime verification* (cf. [25, 13]) which monitors assumptions that had to be made during the model-level analysis, on the implementation level at runtime, using monitors that can be automatically generated from linear-time temporal logic (LTL, [42]) formulae (cf. Section 4) using the open source LTL3TOOLS [49].

The combination of the two verification approaches thus allows us to ensure that Dolev-Yao type security properties will be enforced by the implementation at runtime.

We discuss runtime verification in detail in Section 4, but in a nutshell it works like this: we are given a formal specification of desired or undesired system behaviour. From this, a so called *monitor* is automatically generated, which is a software component that, at runtime, compares an observed behaviour of a system with the specified reference behaviour. If a mismatch is detected, the monitor signals an alarm. If at a certain point it becomes clear that the observed behaviour from then on will always satisfy the given reference behaviour, the monitor signals confirmation. Otherwise, it continues monitoring the system until one of the two situations mentioned above occurs (if ever).

Note that we are not concerned with the correct implementation of low-level crypto-algorithms (such as key generation or encryption). Instead, we would like to make sure that certain assumptions on the correct use of these algorithms within a protocol (e. g., that the validity of a signature is indeed checked at a certain point in the protocol), that have to be made when performing a static security analysis at the model level, are satisfied at runtime at the implementation level. In particular, our aim is not to verify the implementation against low-level weaknesses that cannot be detected using Dolev-Yao style verification (such as buffer overflows), for which other tools already exist that can be applied in addition to ours.

The goal of the proposed process is thus to ensure that the *implementation* of a crypto-protocol is conformant to its specification as far as the Dolev-Yao style security properties are concerned which have been verified against some security properties at the specification-level. To check conformance with respect to the security properties of the implementation against the specification, we have to ensure in particular that the implementation will only send out those (security-relevant) message parts that are permitted by the protocol specification, and only *whenever* they are permitted by the specification (e. g. after a certain signature check has been performed). Our approach allows us to enforce this conformance separately for each of the distributed participants in a protocol (such as client and server), by generating a security monitor for each of the parts that should be monitored in a distributed way. In particular, it allows the user to apply our runtime assurance approach only to *some* participants in the protocol: For example, those for which one has access to the source code, which is the level at which we will apply the approach in this paper (although in principle run-time verification can also be applied to system components available as a black box only, without access to the code).

For example, our approach can in particular enforce that the secret is only sent out on the network whenever specified by the protocol design, and only after encrypting it appropriately. As another example, the protocol specification may require that a session key may only be sent out encrypted under a public key that was received before, and *after* checking that the certificate for that public key is valid. In this paper, we will focus on examples of the latter kind that concern the kind of cryptographic checks that according to the protocol specification need to have been performed correctly before the next message in a protocol can be sent out.

It is interesting to note that our approach bears some similarities with the work of Schneider [43], who introduced *security automata* to detect violations of so called *safety properties* at runtime at the implementation level. Note, however, that safety properties form only a strict subset of those properties relevant to runtime verification of crypto-protocols, as demonstrated by our case study of the widely used SSL-protocol for Internet communication encryption. In particular, our approach to runtime verification strictly exceeds Schneider's security automata, as we will demonstrate in this paper, and allows one to generate monitors for properties that go beyond the "safety spectrum", as was necessary in our application to SSL. Also, this work seems to be the first application of runtime verification to crypto-protocol implementations.

Specifically, we explain our approach at the hand of JESSIE, an open source implementation of the *Java Secure Socket Extension* (JSSE), which includes the SSL-protocol. We first generate finite state machines as acceptors for the relevant properties defined at the specification-level, and then generate Java code from these state machines. The outcome then constitutes the executable monitor which watches over our implementation of Jessie. In addition to that, we also briefly explain how to transfer these results to Sun's own implementation of the JSSE, which was recently made open source.

Note that it is not in scope of the current paper to explain how the LTL formulas used in the run-time verification could be generated automatically from a UMLsec specification.

Outline. This article is a significantly extended version of our previous paper [11]. Additions in comparison to that paper include:

– the first part of the methodology, which performs the automated, static security verification on the model level, in a way that is tightly integrated with the later runtime verification part,

[2] A Dolev-Yao style security analysis of a cryptographic protocol is a security analysis of the interaction of the protocol participants with a man-in-the-middle attacker, at a relatively high level of abstraction which does not consider low-level properties (such as bit-level properties of cryptographic algorithms, or traffic analysis) but focusses on the correctness of the use of cryptography in the protocol.

– the integration of the overall approach including the two phases (static model verification and runtime implementation verification)
– the implementation of automated, formally based tool-support for both phases of the approach and detailed description of this tool-support, and
– an explanation of how the application of our approach was transferred to other libraries such as Sun's own JSSE implementation.

The remaining parts of this article are structured as follows. In the next section, we first outline related work. In Section 3, we introduce the SSL-protocol and identify relevant security properties. We then explain how protocol models in the security extension UMLsec of the Unified Modeling Language (UML) can be automatically verified against these security properties. In Section 4, we provide more details on runtime verification, and discuss how the approach we employ in our work exceeds Schneider's security automata in its formal expressiveness. Section 5 then identifies what we call *runtime security properties*, which we have derived from our specification used for static verification. We formalise these properties in the widely used temporal logic LTL [42], and discuss how to automatically derive a monitor from these formalised properties. Specifically, we provide details on the finite state machines that are generated first, and subsequently on the generated code. Moreover, we state briefly how our results developed at the hand of an application to the open source library Jessie, can be transferred to Sun's own SSL-implementation as part of the JSSE, which was recently made open source as well. Finally, in Section 6, we conclude.

2. Related work

2.1. *Monitoring, runtime verification & security automata*

Monitoring systems is a widely used means to verify that the behaviour of a running system, i. e., a stream of isolated *events*, adheres to its intended behaviour. Many examples are described in different areas much older than the emerging field of runtime verification; for instance, [33, 51] describe the use of synchronous observers to dynamically verify control software, and one may even count classic debugging as a form of monitoring (where the system is a "glass-box system" as compared to a "black-box system", where only the outside visible behaviour can be observed). However, such approaches are typically less rigorous, and less structured than runtime verification, which is formally based on temporal logics, or other forms of regular languages to specify the properties one is interested in, formally. As a scientific discipline, runtime verification was pioneered by works of Havelund and Rosu [34] (see also [35, 36]), who described how to obtain efficient monitors for specifications given in (past-time) linear-time temporal logic (LTL, [42]). Note that for different "flavours" of temporal logic, different approaches to runtime verification exist (cf. [12, 31, 41, 13, 9] for an overview).

Moreover, via the *Property Specification Language* (PSL), there exists nowadays an IEEE industry standard, IEEE1850, for temporal logic which subsumes LTL as well. There exists a considerable amount of tool support for creating and verifying PSL specifications, in particular, with respect to chip design and integrated circuits. We believe that the adoption of PSL and temporal logic by industry is also beneficial for the adoption of our approach in security-critical environments in general.

The techniques used in runtime verification also bear a resemblance with the well-known *security automata* as introduced by Schneider [43], and already mentioned in the introduction. Formally, Schneider's work is based on temporal logic as well, however, imposes restrictions on the types of specifications which can be monitored (or "enforced" to put it in Schneider's own terms). Security automata are restricted to the so-called *safety fragment* (of LTL). For a formula which is from the safety fragment, the corresponding (possibly empty) set of words satisfying the formula (the so-called language) is of such a form that any word *not* in this set can be recognised by an automaton using a prefix of that formula only. Note that this is not always possible for LTL formulae in general, e. g., there exist formulae formalising so-called liveness properties, or co-safety properties, that are not safety properties. Because the properties we consider go beyond the pure safety fragment of LTL, our approach is strictly more expressive than Schneider's original work, and this also explains why in our case study (see Section 5), we could not simply use security automata in the first place.

There also exists work by Clarkson and Schneider [24], in which the scope of properties from describing merely a set of words to sets of sets of words is extended, i.e., to so-called *hyperproperties*. Although Clarkson and Schneider have been able to describe some important security policies using hyperproperties that cannot be described using the types of properties used in this paper, it is not clear to us whether hyperproperties can be also be operationalized in the same efficient way as non-hyperproperties. We therefore do not use hyperproperties as a specification formalism for our method in this paper, although this could indeed be interesting future work.

Another application of monitoring to security was presented in [50]. The paper proposes a caller-side rewriting algorithm for the byte-code of the .NET virtual machine where security checks are inserted around calls to security-relevant methods. The work is different from ours in that it has not been applied to the security verification of cryptographic protocols, which pose specific challenges (such as the correct use of cryptographic functions and checks). In another approach, [45] proposes to use formal patterns of LTL formulae that formalise frequently reoccurring system requirements as *security monitoring patterns*. Again, this does not seem to have been applied to cryptographic protocols so far.

2.2. *Code-level security hardening*

Approaches for code-level security hardening based on approaches other than runtime verification exist as well, including the following examples. Again, they differ from the work

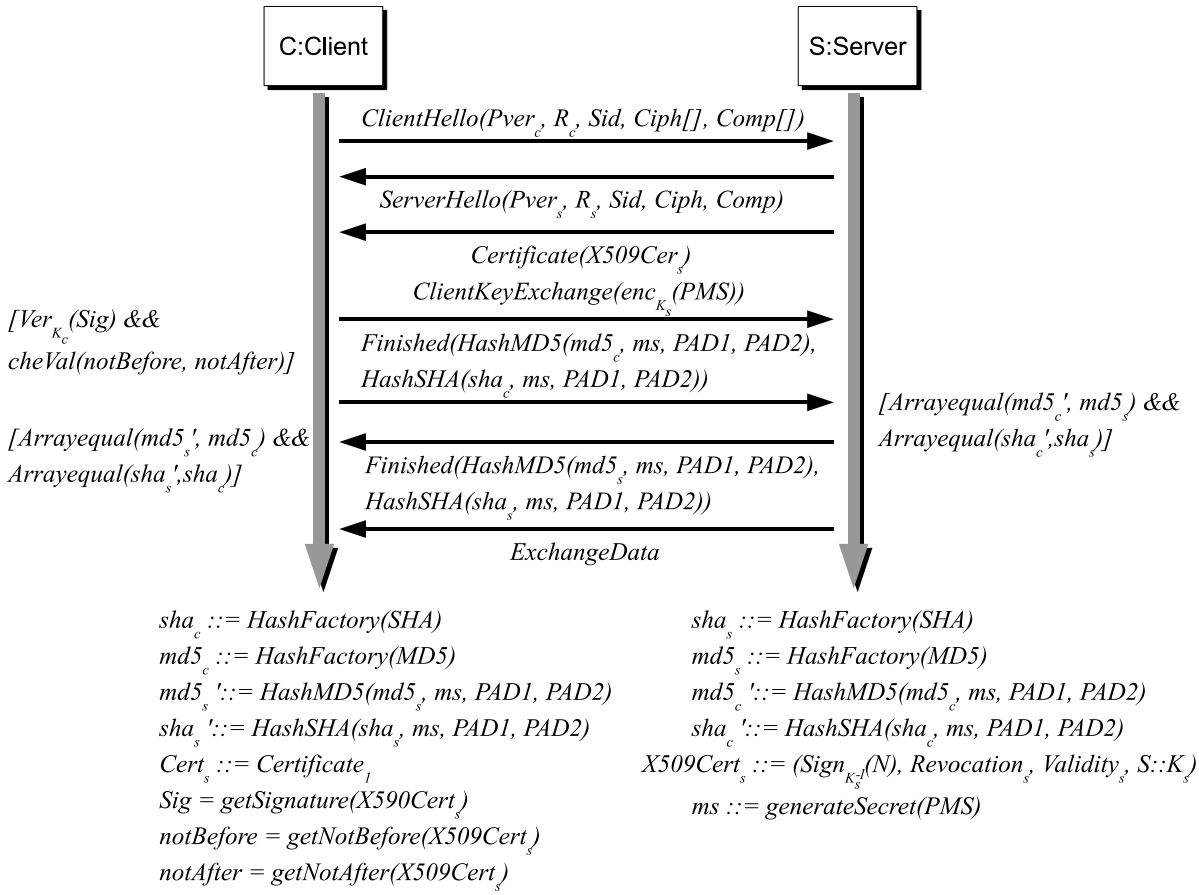

Fig. 1. The cryptographic protocol implemented in SSLSocket.java

we present here in that they have not been applied to crypto protocol implementations and their specific requirements. [28] describes an approach for retrofitting legacy code with security functionality, specifically applied to authorisation policy enforcement. It can be used to identify security-sensitive locations in legacy servers in order to place reference monitor calls to mediate these locations. [53] shows how to apply aspect-oriented programming to implement security functionality such as access control and logging on the method level.

More recently, there has been a growing interest in formally verifying implementations of crypto-protocols against high-level security requirements such as secrecy with respect to Dolev-Yao attacker models (cf. [39, 32, 38, 15]). These works so far have aimed to verify implementations which were constructed with verification in mind (and in particular fulfil significant expectations on the way they are programmed) [32, 15], or deal only with simplified versions of legacy implementations [39, 38]. Our use of runtime verification is motivated by the observation that so far it has not seemed to be feasible to statically and formally verify legacy implementations of practically relevant complexity against high-level security requirements such as secrecy.

Other work on security verification on the code level includes [22, 18, 29, 8, 17, 20, 21]. Again, these approaches so far do not seem to have been applied to crypto-protocol implementations.

Other approaches to the model-based development of security-critical software include [10, 3, 54, 30, 37, 16]. These do not seem to have been used in connection with runtime verification so far.

3. Security properties of the SSL-protocol

SSL is the de-facto standard for securing http-connections and is therefore an interesting target for a security analysis. It may be interesting to note that early versions of SSL (before becoming a “standard” renamed as TLS in RFC 2246) had been the source of several significant security vulnerabilities in the past [2]. In this paper, we concentrate on the fragment of SSL that uses RSA as the cryptographic algorithm and provides server authentication (cf. Figure 1).

As usual in the formal analysis of crypto-based software, the crypto-algorithms are viewed as abstract functions. In our application, these abstract functions represent the implementations from the *Java Cryptography Architecture* (JCA). The messages that can be created from these algorithms are then as usual formally defined as a term algebra generated from ground data, such as variables, keys, nonces, and other data using symbolic operations. These symbolic operations are the abstract versions of the crypto-algorithms.

We assume a set $\mathbf{Keys}$ of encryption keys disjointly partitioned in sets of *symmetric* and *asymmetric* keys. We fix a set $\mathbf{Var}$ of *variables* and a set $\mathbf{Data}$ of *data values* (which may

include *nonces* and other secrets). The *algebra of expressions* $\mathbf{Exp}$ is the term algebra generated from the set $\mathbf{Var} \cup \mathbf{Keys} \cup \mathbf{Data}$ with the operations given in Figure 2. There, the symbols E, E', and E'' denote terms inductively constructed in this way. Note that, as syntactic sugar, encryption enc(E, E') is often written more shortly as $\{E\}_{E'}$, sign(E, E') as $\mathcal{S}ign_{E'}(E)$, conc(E, E') as $E :: E'$, and inv(E) as E^{-1}. In that term algebra, one defines the equations dec(enc(E,K),inv(K))=E and ver(sign(E,inv(K)),K,E)=true for all terms E,K, and the usual laws regarding concatenation, head(), and tail().

enc(E, E')	(encryption)
dec(E, E')	(decryption)
hash(E)	(hashing)
sign(E,E')	(signing)
ver(E,E',E")	(verification of signature)
kgen(E)	(key generation)
inv(E)	(inverse key)
conc(E,E')	(concatenation)
head(E) and tail(E)	(head and tail of concat.)

Fig. 2. Abstract Cryptographic Operations

Note that the cryptographic functions in the JCA are implemented as several methods, including an object creation and possibly initialisation. Relevant for our analysis are the actual cryptographic computations performed by the digest(), sign(), verify(), generatePublic(), generatePrivate(), nextBytes(), and doFinal() methods (together with the arguments that are given beforehand, possibly using the update() method), so the others are essentially abstracted away. Note also that the key and random generation methods generatePublic(), generatePrivate(), and nextBytes() are not part of the crypto-term-algebra but are formalised implicitly in the logical formula by introducing new constants representing the keys and random values (and making use of the inv(E) operation in the case of generateKeyPair()).

In our particular protocol, setting up the connection is done by two methods: doClientHandshake() on the client side and doServerHandshake() on the server side, which are part of the SSL socket class in jessie-1.0.1/org/metastatic/jessie/provider. After some initialisations and parameter checking, both methods perform the interaction between client and server that is specified in Figure 1. Each of the messages is implemented by a class, whose main methods are called by the doClientHandshake() rp. doServerHandshake() methods. The associated data is given in Figure 3.

Message name	Class of Message Type	Message Type
`ClientHello`	`ClientHello`	`CLIENT_HELLO`
`ServerHello`	`ServerHello`	`SERVER_HELLO`
`Certificate*`	`Certificate`	`CERTIFICATE`
`ClientKeyExchange`	`ClientKeyExchange`	`CLIENT_KEY_EXCHANGE`
`Finished`	`Finished`	`FINISHED`

Fig. 3. Data for the Handshake message

We must now determine for the individual data how it is implemented on the code level, to then be able to verify that this is done correctly. We explain this exemplarily for the variable randomBytes written by the method ClientHello to the message buffer. By inspecting the location at which the variable is written (the method write(randomBytes) in the class Random), we can see that the value of randomBytes is determined by the second parameter of the constructor of this class (see Figure 4).

```
1  Random(int gmtUnixTime, byte[] randomBytes)
   {
3    this.gmtUnixTime = gmtUnixTime;
     this.randomBytes = (byte[])randomBytes.clone();
5  }
```

Fig. 4. Constructor for random

in Model	Send: ClientHello	by Outputstream.write in
	type.getValue()	Handshake.write
	(bout.size() >>> 16 & 0xFF)	Handshake.write
	(bout.size() >>> 8 & 0xFF)	Handshake.write
	(bout.size() & 0xFF)	Handshake.write
Pver	major	ProtocolVersion.write
	minor	ProtocolVersion.write
	((gmtUnixTime >>> 24) & 0xFF)	Random.write
	((gmtUnixTime >>> 16) & 0xFF)	Random.write
	((gmtUnixTime >>> 8) & 0xFF)	Random.write
	(gmtUnixTime & 0xFF)	Random.write
R_C	randomBytes	ClientHello.write
	sessionId.length	ClientHello.write
Sid	sessionId	ClientHello.write
	((suites.size() << 1) >>> 8 & 0xFF)	ClientHello.write
	((suites.size() << 1) & 0xFF)	ClientHello.write
Ciph[]	id[]	CipherSuite.write
	comp.size()	ClientHello.write
Comp[]	comp[2]	ClientHello.write

Fig. 5. Data in ClientHello message

Therefore the contents of the variable depends on the initialisation of the current random object and thus also on the program state. Thus we need to trace back the initialisation of the object. In the current program state, the random object was passed on to the ClientHello object by the constructor. This again was delivered at the initialisation of the Handshake object in SSLSocket. doClientHandshake() to the constructor of Handshake. Here (within doClientHandshake()), we can find the initialisation of the Random object that was passed on. The second parameter is generateSeed() of the class SecureRandom from the package java.security. This call determines the value of randomBytes in the current program state. Thus the value randomBytes is mapped to the model element R_C in the message ClientHello on the model level. For this, java.security.SecureRandom.generateSeed() must be correctly implemented. To increase our confidence in this assumption of an agreement of the implementation with the model (although a full formal verification is not the goal of this paper),

$$\forall E_1, E_2.\ \big(\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\ \Rightarrow\ \mathsf{knows}(E_1 :: E_2) \wedge \mathsf{knows}(\{E_1\}_{E_2}) \wedge\ \mathsf{knows}(\mathcal{S}ign_{E_2}(E_1))\big)$$
$$\wedge\big(\mathsf{knows}(E_1 :: E_2)\ \Rightarrow\ \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\big)$$
$$\wedge\big(\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1})\ \Rightarrow\ \mathsf{knows}(E_1)\big)$$
$$\wedge\big(\mathsf{knows}(\mathcal{S}ign_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2)\ \Rightarrow\ \mathsf{knows}(E_1)\big)$$

Fig. 6. Structural formulae

all data that is sent and received must be investigated. In Figure 5, the elements of the message ClientHello of the model are listed. Here it is shown which data elements of the first message communication are assigned to which elements in the doClientHandshake() method.

A crypto-protocol like the one specified in Figure 1 can then be verified at the specification level for the relevant security requirement such as secrecy and authenticity. This can be done using one of the tools available for this purpose, such as the UMLsec tool [48], which is based on the well-known Dolev-Yao adversary model for security analysis. The idea is here that an adversary can read messages sent over the network and collect them in her knowledge set. The adversary can merge and extract messages in the knowledge set and can delete or insert messages on the communication links. The security requirements can then be formalised using this adversary model. For example, a data value remains secret from the adversary if it never appears in the knowledge set of the adversary.

We explain our translation from crypto-protocols specified as UML sequence diagrams to first-order logic formulae which can be processed by the automated theorem prover e-SETHEO [46]. The formalisation automatically derives an upper bound for the set of knowledge the adversary can gain.

The idea is to use a predicate $\mathsf{knows}(E)$ meaning that the adversary may get to know E during the execution of the protocol. For any data value s supposed to remain secret as specified in the UMLsec model, one thus has to check whether one can derive $\mathsf{knows}(s)$. The set of predicates defined to hold for a given UMLsec specification is defined as follows.

For each publicly known expression E, one defines $\mathsf{knows}(E)$ to hold. The fact that the adversary may enlarge her set of knowledge by constructing new expressions from the ones she knows (including the use of encryption and decryption) is captured by the formula in Figure 6.

For each object O a given sequence diagram, our analysis defines a predicate $\mathsf{PRED}(O)$ which captures the behaviour of the object O as relevant from the point of view of the attacker. Thus, for our purposes, a sequence diagram provides, for each object O, a sequence of command schemata of the form *await event e – check condition g – output event e'* represented as *connections* in the sequence diagrams. Connections are the arrows from the life-line of the source object O to the life-line of a target object which are labelled with a message to be sent from the source to the target and a guard condition that has to be fulfilled.

Suppose we are given an object O in the sequence diagram and a connection $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ with:

- $\mathsf{source}(l) = O$,
- $\mathsf{guard}(l) \equiv cond(arg_1, \ldots, arg_n)$, and
- $\mathsf{msg}(l) \equiv exp(arg_1, \ldots, arg_n)$,

where the parameters arg_i of the guard and the message are variables which store the data values exchanged during the course of the protocol. Suppose that the connection l' is the next connection in the sequence diagram with $\mathsf{source}(l') = O$ (i.e. sent out by the same object O as the message l). For each such connection l, we define a predicate $\mathsf{PRED}(l)$ as in Figure 7. If such a connection l' does not exist, $\mathsf{PRED}(l)$ is defined by substituting $\mathsf{PRED}(l')$ with $true$ in this formula.

$$\mathsf{PRED}(l) =$$
$$\forall exp_1, \ldots, exp_n.\big(\mathsf{knows}(exp_1) \wedge \ldots \wedge \mathsf{knows}(exp_n)$$
$$\wedge\ cond(exp_1, \ldots, exp_n)$$
$$\Rightarrow\ \mathsf{knows}(exp(exp_1, \ldots, exp_n)$$
$$\wedge\ \mathsf{PRED}(l'))\big)$$

Fig. 7. Connection predicate

The formula formalises the fact that, if the adversary knows expressions $exp_1, \ldots, exp_n$ validating the condition $cond(exp_1, \ldots, exp_n)$, then she can send them to one of the protocol participants to receive the message $exp(exp_1, \ldots, exp_n)$ in exchange, and then the protocol continues. With this formalisation, a data value s is said to be kept secret if it is not possible to derive $\mathsf{knows}(s)$ from the formulae defined by a protocol. This way, the adversary knowledge set is approximated from above (because one abstracts away for example from the message sender and receiver identities and the message order). This means, that one will find all possible attacks, but one may also encounter "false positives", although this has not happened yet with any real examples. The advantage is that this approach is rather efficient.

For each object O in the sequence diagram, this gives a predicate $\mathsf{PRED}(O) = \mathsf{PRED}(l)$ where l is the first connection in the sequence diagram with $\mathsf{source}(l) = O$. The axioms in the overall first-order logic formula for a given sequence diagram are then the conjunction of the formulae representing the publicly known expressions, the formula in Figure 6, and the conjunction of the formulae $\mathsf{PRED}(O)$ for each object O in the diagram. The conjecture, for which the automated theorem prover (ATP) will check whether it is derivable from the axioms, depends on the security requirements contained in the class diagram. For the requirement that the data value s is to be kept secret, the conjecture is $\mathsf{knows}(s)$.

4. Runtime verification for systems monitoring

Verification on the specification level checks whether or not an abstract model satisfies predetermined security properties. Monitoring, on the other hand, checks whether or not the *implementation* of this model correctly and securely realises the specification. More generally, monitoring systems subsumes all techniques that help verify that the behaviour of a running system, i. e., a stream of isolated *events*, adheres to its intended behaviour. The approaches to monitoring range from ad-hoc to formal methods based on inference and logic (cf. Section 2). The latter are often referred to as runtime verification. In runtime verification, a reference behaviour is specified, typically in terms of a temporal logic language, such as linear time temporal logic (LTL, [42]), and then a so called monitor is generated which compares a system's observable behaviour against this specification. As such, it operates in parallel to the system and is intended not to influence its behaviour.

4.1. *Notions and notation*

In this section, we briefly recall some basic definitions regarding LTL and runtime verification thereof, and introduce the necessary notation. First, let AP be a non-empty set of *atomic propositions*, and $\Sigma = 2^{AP}$ be an *alphabet*. Then infinite words over Σ are elements from Σ^ω and are abbreviated usually as $w, w', \ldots$. Finite words over Σ are elements from Σ^* and are usually abbreviated as $u, u', \ldots$. As is common, we set $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$ as the set of all infinite and finite words over Σ.

We will adopt the following terminology with respect to monitoring LTL formulae. We will use the propositions in AP to represent atomic system *actions*, which is what will be directly observed by the monitors introduced further below. Note that, by making use of dedicated actions that notify the monitor of changes in the system state, one can also indirectly use them to monitor whether properties of the system state hold. Thus, we can use the terms "action occurring" and "proposition holding" synonymously. An *event* will denote a set of propositions and a *word* will denote a *sequence of events* (i. e., a system's behaviour over time). The idea is that a monitor observes a stream of system events and that multiple actions can occur simultaneously.

To specify a system's behaviour (in order to define a monitor), we employ the temporal logic LTL which is defined as follows.

Definition 1 (LTL syntax and semantics, [42]) *The set of LTL formulae over Σ, written LTL(Σ), is inductively defined by the following grammar:*

$$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \, \mathbf{U} \, \varphi \mid \boldsymbol{X} \, \varphi, \quad p \in AP.$$

The semantics *of LTL formulae is defined inductively over its syntax via the satisfaction relation "$\models$" as follows. Let $\varphi, \varphi_1, \varphi_2 \in$ LTL(Σ) be LTL formulae, $p \in AP$ an atomic proposition, $w \in \Sigma^\omega$ an infinite word, and $i \in \mathbb{N}$ a position in w. Let $w(i)$ denote the ith element in w (which is a set of propositions):*

$$\begin{aligned}
&w, i \models true \\
&w, i \models \neg\varphi && \Leftrightarrow w, i \not\models \varphi \\
&w, i \models p && \Leftrightarrow p \in w(i) \\
&w, i \models \varphi_1 \vee \varphi_2 && \Leftrightarrow w, i \models \varphi_1 \vee w, i \models \varphi_2 \\
&w, i \models \varphi_1 \, \boldsymbol{U} \, \varphi_2 && \Leftrightarrow \exists k \geq i.\ w, k \models \varphi_2 \wedge \\
& && \qquad \forall i \leq l < k.\ w, l \models \varphi_1 \\
& && \qquad \text{(“}\varphi_1 \text{ until } \varphi_2\text{”)} \\
&w, i \models \boldsymbol{X} \, \varphi && \Leftrightarrow w, i+1 \models \varphi \quad \text{(“next } \varphi\text{”)}
\end{aligned}$$

where w, i denotes the ith position of w. We use $w(i)$ to denote the ith element in w which is a set of propositions. Notice the difference between w, i and $w(i)$, i.e., the former marks a position in the trace, whereas the latter denotes a set.

When $w \models \varphi$ holds, we also say that w is a *model* for the formula φ in the logical sense of the word, , meaning that w is a word which satisfies the formula. Intuitively, the statement $w, i \models \varphi$ is supposed to formalise the situation that the event sequence w satisfies the formula φ at the point when the first i events in the event sequence w have happened. In particular, defining $w, i \models true$ for all w and i means that $true$ holds at any point of any sequence of events. We also write $w \models \varphi$, if and only if $w, 0 \models \varphi$.

Further, as is common, we use $\mathbf{F}\ \varphi$ as short notation for $true\ \mathbf{U}\ \varphi$ (intuitively interpreted as "eventually φ"), $\mathbf{G}\ \varphi$ short for $\neg\mathbf{F}\ \neg\varphi$ ("always φ"), and $\varphi_1\ \mathbf{W}\ \varphi_2$ short for $\mathbf{G}\ \varphi_1 \vee (\varphi_1\ \mathbf{U}\ \varphi_2)$, which is thus a weaker version of the $\mathbf{U}$ -operator. For brevity, whenever Σ is clear from the context or whenever a concrete alphabet is of no importance, we will use LTL instead of LTL(Σ). Moreover, we make use of the standard Boolean operators $\Rightarrow, \wedge, \ldots$ that can easily be defined via the above set of operators.

Example 1 *We give some examples of LTL specifications. Let $p \in AP$ be an action (formally represented as a proposition). Then $\mathbf{G}\,\mathbf{F}\ p$ asserts that at each point of the execution of any of the event sequences produced by the system, p will afterwards eventually occur. In particular, it will occur infinitely often in any infinite system run.*

For another example, let $\varphi_1, \varphi_2 \in$ LTL be formulae. Then the formula $\varphi_1\ \mathbf{U}\ \varphi_2$ states that φ_1 holds until φ_2 holds and, moreover, that φ_2 will eventually hold. On the other hand, $\mathbf{G}\ p$ asserts that the proposition p always holds on a given trace (or, depending on the interpretation of this formula, that the corresponding action occurs at each system update).

It is worth emphasising the point that the "Until-operator" as defined above can be somewhat counterintuitive to the way that the word "until" is used in natural language; that is, the formula $a\ \mathbf{U}\ b$ is satisfied if and only if b happens at some point in time in the future, and a held until then. In the case of the weaker until, $\mathbf{W}$, we do not demand occurrence of b, and the formula would be satisfied also by observing an infinitely recurring action a. Hence, when formalising natural language specifications in terms of LTL, the word "until" in the natural language will very often be translated to "weak until" in LTL.

4.2. *Monitorable languages*

An LTL formula gives rise to a set of infinite words, $L \subseteq \Sigma^\omega$, i. e., a language whose elements satisfy the formula according to the entailment relation of Definition 1. L also naturally gives rise to another set, defined as $\Sigma^\omega - L$, i. e., the set of all words not contained in L. Let us now examine some properties of the languages (i.e. sets of words) that can be specified this way. We first introduce the notion of *bad* and *good* prefixes.

Definition 2 *Let $L \subseteq \Sigma^\omega$ be a language of infinite words over alphabet Σ. A finite word $u \in \Sigma^*$ is called a* bad prefix *(with respect to L) iff there exists no infinite extension $w \in \Sigma^\omega$, such that $uw \in L$ (where uw is the concatenation of the words u and w). In contrast, the finite word u is called a* good prefix *(with respect to L) iff for any infinite extension $w \in \Sigma^\omega$ it holds that $uw \in L$.*

We say that u is a good (resp. bad) prefix of w wrt. a language L if u is a prefix of w and if u is a good (resp. bad) prefix wrt. L as defined above. Note that a word w may have neither a good nor a bad prefix with respect to a language L associated with a given formula φ. In this case, when monitoring this sequence of events, it will at no finite point in time be clear whether all extensions of the sequence of events monitored so far will satisfy the formula φ, or not.

Definition 3 *L is called a* safety language *iff all words $w \in \Sigma^\omega - L$ have a bad prefix.*

When we refer to the formula $\varphi \in LTL$ giving rise to a safety language, we will also use the term *safety property* to say that the logical models of φ adhere to Definition 3.

Definition 4 *L is called a* co-safety language *iff all words $w \in L$ have a good prefix.*

Note that the complements of safety languages (i. e. the language $\Sigma^\omega - L$ where L is a safety language) are exactly the co-safety languages. The proof of this fact is straightforward, by making use of the central observation that a word w is a bad prefix with respect to the language $\Sigma^\omega - L$ exactly if it is a good prefix with respect to the language L, which follows directly from the definitions for good resp. bad prefixes.

Note that there exist languages which are both safety and co-safety languages, such as the empty set $\emptyset$, and the set of all infinite words, Σ^ω. Similarly, there also exist languages that are neither safety nor co-safety languages. In fact, some such languages will play a crucial role for our application of monitoring security properties of the SSL-protocol, which cannot be specified using only safety or co-safety properties alone.

Some typical languages that are neither safety nor co-safety languages fall into a third class of languages, the liveness languages, introduced as follows.

Definition 5 *L is called a* liveness language *iff for every prefix $u \in \Sigma^*$ there exists an infinite extension $w \in \Sigma^\omega$, such that $uw \in L$.*

Note however that not all languages outside safety and co-safety are liveness languages, and that there are liveness languages that are at the same time co-safety languages. We revisit the examples given above to shed more light on the relationship between these three different classes of formal properties. Note further that there are other definitions of liveness found in the literature. We have adopted here the one given originally by [4].

Example 2 *The formula **G F** p from above (for a given $p \in AP$) asserts that at each point of a given event sequence produced by the system, p will afterwards eventually occur. It does not assert a frequency of the occurrence but demands that p occurs infinitely often. In particular, every observed behaviour, even if it does not contain an observation of p yet, can be extended such that **G F** p is satisfied. This means in particular that **G F** p is not a safety property: there exist words that are not in the language L defined by **G F** p, which do not have a bad prefix, because every of their prefixes can be extended to a word in L. In fact, this even holds for all words not in L, and indeed also for all words in L as well, which proves that **G F** p is a liveness property. It is not a co-safety property, because there exist words in L which do not have a good prefix (in fact, none of the words in L has a good prefix).*

*For given formulae $\varphi_1, \varphi_2 \in LTL$, the formula φ_1 **U** φ_2 is a co-safety property as every word that satisfies this formula has a good prefix, i. e., can be detected via a finite word. This word is of a form such that it satisfies φ_1 finitely often, and then it indeed satisfies φ_2. The formula is not a safety property: One infinite counterexample is the word that satisfies φ_1, but not φ_2. Hence, not all counterexamples have a bad prefix as required by Definition 3. φ_1 **U** φ_2 is also not a liveness property: a word that does not satisfy φ_1 at its first position cannot be extended to a word that satisfies φ_1 **U** φ_2.*

*Finally, for a given $p \in AP$, the formula **G** p is a safety property as all counterexamples have a bad prefix. It is not a co-safety property: Intuitively speaking, one can never be sure that a given event sequence that satisfies p up to a given finite point in time will always satisfy p. It is also not a liveness property because again, a word that does not satisfy p at its first position cannot be extended to a word that satisfies **G** p.*

The above three examples demonstrate that each of the classes of safety, co-safety resp. liveness properties contain properties not contained in any of the other two classes. One should also note that there exist properties that are not contained in any of the three classes.

In particular, there are properties which we believe to be security-relevant and which cannot be monitored using Schneider's security automata [43], which however can be monitored using our approach, as we will discuss in Section 4.3. For example, these can be properties of the form "an event will eventually happen" (as in the second example above). For this kind of property, the monitor cannot only just confirm that the relevant event has happened when it has happened. It can potentially also confirm at a given point in the execution of the system, that the event will not eventually happen (i.e. it will never happen from this point onwards). There are indeed security-relevant uses for such properties. One particularly important class is denial-of-service properties. Although this paper is not specifically concerned with denial of service, this kind of property does become relevant in our application here (cf. Section 3), where some properties aim to establish that a certain message in the protocol will, under certain conditions, eventually be sent

out. If an adversary could detect conditions under which this is not the case, this might enable him to launch a denial of service attack against the protocol.

We will now briefly explain the approach for runtime verification used in this paper. Let $\varphi \in$ LTL be the specification we wish to monitor. The monitors generated by our approach monitor this formula by interpreting it using a 3-valued semantics that is defined with respect to a finite event sequence as follows.

Definition 6 *Let $\varphi \in LTL$, and $u \in \Sigma^*$. Then, a monitor for φ realises the following* entailment relation*:*

$$[u \models \varphi] := \begin{cases} \top & \text{if } u \text{ is a good prefix wrt. } L(\varphi) \\ \bot & \text{if } u \text{ is a bad prefix wrt. } L(\varphi) \\ ? & \text{otherwise.} \end{cases}$$

The $[\cdot]$ is used to separate the 3-valued monitor semantics for φ from the classical, 2-valued LTL semantics introduced in Definition 1.

Note that monitoring only makes sense if there is hope that a conclusive answer (i. e., $\neq ?$) is obtainable at all. If the language does not allow for such an answer, then it makes little sense to monitor it, as all verdicts by such a monitor would yield $?$. Therefore, we define the set of languages that are *monitorable* using our approach as follows:

Definition 7 *Let MON be the set of all languages over an alphabet Σ, for which there exists some $u \in \Sigma^*$, such that $[u \models \varphi] \neq ?$.*

Thus, MON is the set of all languages for which there exists a good or bad prefix. Note that not all words need to be recognisable for such a language via a good, resp. bad prefix—just some, such that the monitor can detect such a prefix should it occur. For such languages, our approach to runtime verification will produce a monitor that outputs $\top$ in case of a good prefix and $\bot$ in case of a bad prefix observed. While neither is observed, such a monitor will output $?$, meaning that the prefix seen so far does not allow a more conclusive answer as to whether the monitored language will be satisfied or violated when extending the prefix.

Notably, the generated monitors are able to detect good resp. bad prefixes of the monitored property φ, should they exist and occur, as early as possible. That is, if the observed trace $u \in \Sigma^*$ is a good resp. bad prefix for φ, then the monitor will not return $?$ as verdict. As such it detects minimal good resp. bad prefixes.

To provide a more precise comparison with Schneider's security automata in the next subsection, we also define the following notion of *syntactically monitorable* properties.

Definition 8 *Let MON_{syn} be the set of all languages that correspond to properties defined as conjunctions or disjunctions of safety and co-safety properties (i.e. properties $\phi_1 \wedge \ldots \wedge \phi_n$ or $\phi_1 \vee \ldots \vee \phi_n$ where the ϕ_i are safety or co-safety properties).* Note that MON_{syn} includes all safety and co-safety properties because $true$ is both a safety and a co-safety property, and recall that co-safety properties are exactly the negations of safety properties.

Proposition 9 *We have $MON_{syn} \subseteq MON$.*

PROOF. [44, Th. 3.1] shows that the class of safety properties is closed under combination using $\vee$, and $\wedge$ (and by duality the same holds for co-safety properties). It is therefore sufficient to consider the case $\phi \wedge \psi$ where ϕ is a safety property and ψ is a co-safety property (the case for $\vee$ again works by duality). By definition of safety properties, we know that all words that do not satisfy ϕ have a bad prefix. If there is no word that does not satisfy ϕ, then $\phi \wedge \psi = \psi$. Otherwise, there is a word with a bad prefix for ϕ, which is also a bad prefix for $\phi \wedge \psi$. □

4.3. *Expressiveness in comparison to security automata*

Schneider's security automata [43] which he proposes for the "enforcement" of system properties, accept exactly elements of the safety fragment of ω-regular languages. ω-regular languages, in contrast to regular languages (i.e., the languages which can be defined via commonly used regular expressions or finite automata), are those which can be defined via infinite automata, such as nondeterministic Büchi automata (cf. [23]). Moreover, the languages definable in LTL are ω-regular, although not every ω-regular language is definable in LTL. From the discussion in the previous subsection it then follows that, using complementation of a given system property (specified in LTL), security automata could also be used to "enforce" co-safety properties. That is, if $\varphi \in LTL$ is a safety property corresponding to desired system behaviour, then its negation, $\overline{\varphi}$, is a co-safety property corresponding to undesired system behaviour. The result of the automaton/monitor would then have to be inverted, accordingly.

As we have seen in the previous subsection, however, sometimes we may want to specify properties which may not fall into either category of safety or co-safety languages (cf. also our application in Section 5.1). Notably as such they also exceed the expressiveness of security automata, although we have managed to successfully create monitors for those. Let us therefore formally establish in what sense our approach to runtime verification exceeds the expressiveness of security automata. Schneider abbreviates the set of languages accepted by security automata as EM, which as pointed out above, coincides with the safety fragment of ω-languages. For EM he observes:

Proposition 10 ([43]) *If the set of executions for a security policy φ is not a safety language, then an enforcement mechanism from EM does not exist for φ.*

Let us now relate this class MON of properties which are monitorable in our approach with the class EM of properties "enforceable" by security automata:

Theorem 11 *It holds that $EM \subset MON_{syn} \subseteq MON$.*

PROOF. The statement $EM \subseteq MON_{syn}$ follows from the facts that EM is contained in the class of safety properties (which was shown in Proposition 10 above), and that MON_{syn} includes the class of safety properties, as noted above. (Note that the statement is still true if we use the "trick" explained above which would allow us to enforce co-safety properties using EM, since MON_{syn} also includes the class of co-safety properties.)

To show that MON_{syn} is strictly larger than EM, it suffices to exhibit a language in MON_{syn} which is demonstrably not contained in EM. Consider the property $\neg\mathbf{F}\ a \vee \mathbf{F}\ b$ where a and b are distinct atomic propositions. The corresponding language is in MON_{syn} since $\neg\mathbf{F}\ a$ is a safety property and $\mathbf{F}\ b$ a co-safety property. $\neg\mathbf{F}\ a \vee \mathbf{F}\ b$ is not in EM because it is not a safety property (nor a co-safety property): Infinite event sequences containing a but not b do not satisfy $\neg\mathbf{F}\ a \vee \mathbf{F}\ b$, but do not have a bad prefix (and infinite event sequences containing neither a nor b satisfy $\neg\mathbf{F}\ a \vee \mathbf{F}\ b$, but do not have a good prefix). □

Note that Property 2 of Section 5 is an example which is contained in MON but not in EM: it is neither safety, nor co-safety as there exists a model without good prefix as well as a counterexample without bad prefix for it (as explained in detail there). On the other hand, using our approach, there exists a monitor for it, which is depicted in Figure 10. Therefore, this property is indeed in MON.

5. Monitoring runtime security properties of SSL

In order to monitor properties of the SSL-protocol, we first need to determine how important elements at the model level are implemented at the implementation level. Basically, this can be done in the following three steps:
– Step 1: Identification of the data transmitted in the sending and receiving procedures at the implementation level.
– Step 2: Interpretation of the data that is transferred and comparison with the sequence diagram.
– Step 3: Identification and analysis of the cryptographic guards at the implementation level.

In Step 1, the communication at the implementation level is examined and it is determined how the data that is sent and received can be identified in the source code. Afterwards, in Step 2, a meaning is assigned to this data. The interpreted data elements of the individual messages are then compared with the appropriate elements in the model. In Step 3, it is described how one can identify the guards from the model in the source code.

To this aim, we first identify where in the implementation messages are received and sent out, and which messages exactly. In doing so, we exploit the fact that in most implementations of cryptographic protocols, message communication is implemented in a standardised way (which can be used to recognise exactly where messages are sent and received). The common implementation of sending and receiving messages in cryptographic protocols is through *message buffers*, by writing the data into type-free streams (i. e., ordered byte sequences), which are sent across the communication link, and which can be read at the receiving end. The receiver is responsible for reading out the messages from the buffer in the correct order and storing it into variables of the appropriate types. Accordingly, in case of the Java implementation JESSIE of the SSL-protocol, this is done by using the methods write() from the class java.io.OutputStream to write the data to be sent into the buffer and the method read() from the class java.io.InputStream to read out the received data from the buffer. Also note that the messages themselves are usually represented by message classes that offer custom write and read methods, and in which the write and read methods from the java.io are called, subsequently.

Moreover, according to the information that is contained in a sequence diagram specification of a cryptographic protocol, the monitor which is generated for performing runtime verification needs to keep track of the following information:
– *Which data is sent out?*, and
– *Which data is received?*

This, in turn, depends on where the monitor will be placed in the actual and possibly distributed implementation; that is, certain properties (of the SSL-protocol) are to be monitored at the client-side while others are to be monitored at the server-side. While our approach is not restricted to either point of view, server or client, the users have to make this decision based on the properties they would like to monitor, the available system resources, accessibility, and so forth.

The monitors, once in place, will then generally enforce that the relevant part of the implementation conforms to the specification in the following sense:
– The code should only send out messages that are specified to be sent out according to the specification and in the correct order, and
– these messages should only be sent out if the conditions that have to be checked first according to the specification are met.

In the next section, we give some example properties which highlight these two points wrt. the SSL-protocol. Note that it may depend on a given application and on the requirements to be monitored whether or not it may be possible to find monitorable properties whose violation at runtime indeed prevents the leaking of data (i.e. the detection occurs before the undesirable situation occurs). While this should be generally possible in many real-world applications where runtime monitoring is applicable, there may also be undesirable events that cannot be detected prior to their occurrence, i.e. there does not exist a suitable property that could be specified by the user or monitored by the system, such that its violation would help anticipate or prevent the undesirable event from happening.

5.1. *Example runtime security properties*

In this section, we consider the examples listed below for properties that should be enforced using runtime verification in the case of the SSL-protocol specified in Figure 1 in more detail. Each of these properties enforces on the implementation level the implicit assumption that had to be made for the model level security analysis that any of the messages in the protocol specified in Figure 1 is only sent out after the relevant protocol participant has satisfactorily performed the required checks on the message that was received just before.

(i) ClientKeyExchange$(enc_K, (PMS))$ is not sent by the client until it has received the Certificate$(X509Cer_s)$

message from the server, has performed the validity check for the certificate as specified in Figure 1, and this check turned out to be positive.

(ii) Finished$(HashMD5(md5_s, ms, PAD1, PAD2))$ is not sent by the server to the client before the MD5 hash received from the client in the message Finished$(HashMD5(md5_c, ms, PAD1, PAD2))$ has been checked to be equal to the MD5 created by the server, and correspondingly for the SHA hash, but will send it out eventually after that has been established.

(iii) The client will not send any transport data to the server before the MD5 hash received from the server in the Finished$(HashMD5(md5_s, ms, PAD1, PAD2))$ message has been checked to be equal to the MD5 created by the client, and correspondingly for the SHA hash.

Below we consider each of the three properties in detail.

Property 1. Step 1 and 2 of the above procedure, yield the two abstract messages ClientKeyExchange and Certificate. Moreover, once we have identified in the source code where these messages are sent respectively received and evaluated, we can add at this point custom code that sets or unsets two "flags": the flag ClientKeyExchange$(enc_K, (PMS))$ is set by the code if and only if the message ClientKeyExchange is sent by the client, and the flag Certificate$(X509Cer_S)$ is set if and only if the certificate was received and positively checked. Otherwise, both flags are unset. Coming back to our formal model of LTL runtime verification (see Section 4), this yields the following set of atomic propositions:

$$AP = \{\text{ClientKeyExchange}(enc_K, (PMS)), \\ \text{Certificate}(X509Cer_S)\},$$

whose names correlate with the ones displayed in Figure 1. Notice that LTL as introduced in Section 4 does not cater for parameters, and parameters in an action's name are therefore not a semantic concept.

Based on AP we can now formalise the required property in LTL as follows, using the "weak until" operator, which in particular allows for the fact that if the certificate is never received, then the formula is satisfied if in turn the message ClientKeyExchange$(enc_K, (PMS))$ is never sent.

$$\varphi_1 = \neg\text{ClientKeyExchange}(enc_K, (PMS)) \\ \mathbf{W}\ \text{Certificate}(X509Cer_S).$$

This meets our intuitive interpretation of the "until" in the natural language requirement because if, for example, a man-in-the-middle attacker deletes any certificate message sent by the server, we cannot possibly demand that ClientKeyExchange$(enc_K, (PMS))$ should be eventually sent by the client.

We can then use the approach to runtime verification described in detail in [13] and realised in the open source tool LTL3TOOLS to automatically generate a finite state machine for monitoring this formula. From this finite state machine, we subsequently generate Java code, i. e., the executable monitor (for details of this process, see Section 5.2), which watches over the protocol implementation while our client participates in an SSL-session. The executable monitor signals the value $\top$ ("property satisfied") once the certificate was received and checked, $\bot$ ("property violated") if the client sends the key without successful check, and it will signal the value ? ("inconclusive") as long as neither of the two conditions holds. Recall that the stream of events that is processed by the monitor consists of elements from 2^{AP} (i. e., the powerset of all possible system actions). That is, at each point in time, the monitor keeps track of *both* events: the sending of ClientKeyExchange$(enc_K, (PMS))$ and the receiving of Certificate$(X509Cer_S)$. Hence, as long as none of the events is observed, the monitor basically processes the empty event. Moreover, φ_1 is a classical safety property, because all counterexamples have a finite bad prefix, i. e., can be recognised as such after finitely many observations. As such, this property is also recognisable by a security automaton.

Property 2. In order to monitor the second requirement as given above, we now take the point of view of the server rather than that of the client. Let

$$AP = \{\text{Finished}(HashMD5(md5_s, ms, PAD1, PAD2)), \\ \text{Arrayequal}(md5_s, md5_c)\}$$

Notice, how we have not added Arrayequal(sha_s, sha_c) to AP. The reason for this is that we will be actually creating two monitors both operating over their own respective alphabets. We can now formalise the corresponding LTL property with respect to AP as we did before:

$$\varphi_2 = (\neg\text{Finished}(HashMD5(md5_s, ms, PAD1, PAD2)) \\ \mathbf{W}\ \text{Arrayequal}(md5_s, md5_c)) \\ \wedge(\mathbf{F}\ \text{Arrayequal}(md5_s, md5_c) \\ \Rightarrow \mathbf{F}\ \text{Finished}(HashMD5(md5_s, ms, \ldots))).$$

To monitor the analogous statement for the SHA rather than the MD5, we define an additional formula, φ_2', where all occurrences of the proposition Arrayequal$(md5_s, md5_c)$ are replaced by the proposition Arrayequal(sha_s, sha_c), respectively. Neither of the two formulae are actually safety or co-safety properties, although there exist finite traces which violate, respectively, satisfy the formula: For instance, consider a trace $u = \emptyset; \{\text{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))\}$, which violates the first part of our conjunction since this implies Finished$(HashMD5(md5_s, ms, PAD1, PAD2)) = \top$, but Arrayequal$(md5_s, md5_c)) = \bot$. On the other hand, the trace $v = \emptyset; \{\text{Arrayequal}(md5_s, md5_c)\}$ is a model for φ_2, since our second observation in v shows that the MD5 checksum was successfully compared, and until then, Finished$(HashMD5(md5_s, ms, PAD1, PAD2))$ did not hold. Note that $\emptyset$ means that all propositions in AP are interpreted as $\bot$ (i.e. that none of the monitored events occurs at that point in time), and we use the symbol ";" to

denote the concatenation of events. However, as pointed out above, φ_2 is not a co-safety property since, besides the trace v which has a good prefix, there also exists the infinite model $v' = \emptyset;\emptyset;\ldots$ without a good prefix, i. e., $\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))$ never holds. Recall, the concept of co-safety asserts that *all* models possess a finite good prefix (Definition 4). Moreover, it is not a safety property since there exists the infinite counterexample $u' = \{\mathsf{Arrayequal}(md5_s, md5_c))\};\emptyset;\ \emptyset;\ldots$ without a bad prefix. Recall, the concept of safety asserts that *all* counterexamples can be recognised via a finite bad prefix (Definition 3), and u', if infinitely extended with $\emptyset$ is, indeed, a counterexample, although it cannot be recognised as such after finitely many observations. Note that a monitor for this property is given in Section 5.2, although there exists no corresponding Schneider security automaton for it.

Property 3. Lastly, the above requirement 3. can be formalised as follows:

$$\varphi_3 = \neg\mathsf{Data}\ \mathbf{W}\ \mathsf{Arrayequal}(md5_s, md5_c),$$

where AP is similar as in the previous example, but contains a proposition indicating the sending of data, Data. Here we have a real safety property again since all traces of violating behaviour for φ_3 are recognisable after finitely many observations. It is not co-safety since the infinite trace $w = \emptyset;\emptyset;\ldots$ satisfies φ_3, which would be the case if an intruder has intercepted and kept the $\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))$ message, such that it is never received at the server-side. Again, as in the previous example, we create a second monitor to check the outcome of the SHA-comparison.

Monitoring a property like $\mathbf{G}\,p$ means that the corresponding monitor would output ? as long as no violation occurred, but never $\top$, since all models are infinite traces without good prefixes. However, φ_2 and φ_3 are such that they do have models with good prefixes, hence the corresponding monitor can output all three values of $\{\top, \bot, ?\}$, depending on the observed system behaviour. The same holds for φ_1, which is actually a safety property. Note that, had we used the "strong until" operator $\mathbf{U}$ in φ_1 and φ_2, then both properties would be classical co-safety properties. Co-safety properties would be monitorable using security automata, however, only via the detour of complementing them and checking that the complements do *not* hold (i. e., one has to "invert" the semantics). Using LTL3TOOLS, this detour is not necessary. We can directly generate a monitor for φ_1 and φ_2 (also for the variants using the "strong until" operator).

5.2. *Implementation*

Once we have formalised the natural language requirements in terms of LTL formulae as above, we can then use our LTL3TOOLS [49] to automatically generate finite state machines (FSM) from which we derive the actual (Java) monitor code. The FSMs obtained from LTL3TOOLS are of type Moore, which means that, in each state that is reached, they output a symbol (i. e., ?, $\top$ (TOP), or $\bot$ (BOT)). States are changed as new system actions become visible to the monitor. The FSM generated for the runtime security property φ_1 is given in Figure 9. The initial state is $(0, 0)$ whose output is ?. If event $\{cert\}$ occurs, short for $\{\mathsf{Certificate}(X509Cer_S)\}$, then the monitor takes a transition into state $(1, -1)$ and outputs $\top$ to indicate that the property is satisfied. On the other hand, if neither $cert$ nor cke, short for $\mathsf{ClientKeyExchange}(enc_K, (PMS))$, occurs, then the automaton remains in $(0, 0)$ and outputs ?, indicating that so far φ_1 has not been violated, but also not been satisfied. A violation would be the reaching of $(-1, 1)$, if event $\{cke\}$ occurs (before $cert$), such that the monitor would output $\bot$. Generating code from this state machine is a straightforward exercise, which we only outline by giving a code example of the generated code (see Figure 8). This code in turn is embedded into an event loop, such that new events can enforce the triggering of transitions, and changing of states. The current state information is stored inside the variable m_state, whereas the events are abbreviated by definitions.

```
1  class MonitorPhi1
   {
3      ...

5      void process_event(Event e)
       {
7          if ( m_state.equals("(0,␣0)") )
           {
9              System.err.println("?");

11             if ( type_ID(e) == EMPTY )
               {
13                 m_state = "(0,␣0)";
                   return;
15             }
               else if ( type_ID(e) == CERT )
17             {
                   m_state = "(1,␣-1)";
19                 return;
               }
21             ...  // process remaining events
           }
23         else if ( m_state.equals("(1,␣-1)") )
           {
25             System.err.println("TOP");
               ...  // process remaining states
27         }
       }
29 }
```

Fig. 8. Fragment of the generated monitor code for φ_1

In order to determine which event has occurred, we use a set of program flags (i. e., Boolean variables) which are initially set to $false$. We set these to $true$ as soon as an identified action has taken place. The combination of flags then determines the current event to process by the monitor. Hence, once the monitor code is generated, the only code that needs to be added to the main application is the code to set the flags, and the code to communicate them in terms of events to the monitor class. According to our experiences, an LTL property of the size that usually occurs in these kinds of application results into a monitor of 150 LOC in Java, including communication code between application and monitor.

The remaining two properties, φ_2 and φ_3 are implemented in the same manner. For brevity, we do not discuss their internals in detail, but give the automatically generated FSMs in Figures 10 and 11, respectively. Note that these monitors are good examples to demonstrate that it is, indeed, feasible to monitor properties beyond the "safety spectrum" that Schneider's security automata capture: All FSMs cover all the three different truth values, and not all of them are safety (or co-safety) formulae (as discussed above in detail).

Notice that other comprehensive Java programs but Jessie are often developed in parallel with an event logging library such as log4j [3], which can then directly be used for capturing all relevant system events that the monitors require. If no such library is used, as in the case of JESSIE, then the set of relevant events needs to be identified first, and all occurrences in the code instrumented accordingly.

5.3. *Application to Sun's JSSE implementation*

Let us also briefly explain how one can apply the runtime verification approach which we discussed above in terms of the open source implementation Jessie of the JSSE to Sun's very own implementation as a library in the standard JDK (since version 1.4), which was recently made open source. The source code of this library (after version 1.6) can be checked out from the OpenJDK repository [4]. To perform the runtime verification as explained above on this implementation, one only needs to modify the mappings between the specification elements and their implementations that provide the traceability of the model to the implementation level. For example, in the Jessie implementation, the doHandshake protocol is mainly implemented in the class SSLSocket of the Jessie (v. 1.0.1) library, whereas in the library implementation in the OpenJDK 1.6 (hereafter called JSSE 1.6), the protocol is mainly implemented in the class sun.security.ssl.HandshakeMessage. Nevertheless, the naming of the symbols can be traced to the implementation. Table 1 lists some mapping from the symbols in Figure 1 to their naming in the JSSE 1.6 library.

Table 1
Traceability mappings in JSSE 1.6

Symbols	JSSE 1.6
1. C	HandshakeMessage.ClientHello
2. S	HandshakeMessage.ServerHello
3. P_{ver}	protocolVersion
4. R_C	clnt_random
R_S	svr_random
5. S_{id}	sessionId
6. Ciph[ ]	cipherSuites
7. Comp[ ]	compression_methods
8. Veri	CertificateVerify.verify()
9. $D_{\text{notBefore}}$	cert.getNotBefore()
D_{notAfter}	cert.getNotAfter()

6. Conclusions

In this article, we successfully address the open problem of how to enforce that certain assumptions implicit in the Dolev-Yao style verification of crypto-protocol specifications are satisfied on the implementation level. Towards this goal, runtime verification allows us to monitor even complex, history-dependent specifications as they arise for security protocols. We have seen, in particular, that some crucial runtime correctness properties of our SSL-implementation could not be monitored using prior formal approaches to monitoring security-critical systems, since they exceed the expressiveness of the safety fragment (of LTL), which is the fragment monitorable by Schneider's security automata. As an example for a security weakness that can be found using our approach, the approach helped us detecting a missing signature check in the Jessie implementation of the SSL protocol.

However, as can also be seen by some of our properties, it is often difficult to decide whether or not a formula is a safety property, whether it is co-safety, or neither—even for the trained eye. In fact, it is known that given some formula $\varphi \in$ LTL, deciding whether $\mathcal{L}(\varphi)$ is safety (co-safety) is a PSPACE-complete problem [5, 44]. Hence, there are theoretical limitations to our approach, whenever it is not clear to the designer of a system whether or not some imposed security property is monitorable at all. However, due to the completeness result, we cannot hope to provide a more efficient or convenient way for performing this check. Also, from the practical point of view, the user can simply attempt to generate the monitor using the LTL3TOOLS. If that should fail, then the user has the option to syntactically simplify the input formula, or perhaps to generate multiple smaller monitors for the subformulae of the original input that will later operate in parallel.

Once the monitors are generated, then the resulting overhead from using monitors is minimal, in the sense that one can show that other approaches for generating monitors from LTL formulae cannot be any more efficient: For example, the particular approach described in [13] and realised by the LTL3TOOLS [49] creates monitors with (near to) *optimal* space complexity with respect to the property to be monitored. It should be pointed out, however, that the intermediate steps in generating a monitor involve a double exponential "blow up" in the length of the specification, but this does not necessarily affect runtime efficiency, when the monitors are minimised as a final step. On the other hand, sophisticated event logging libraries such as log4j can create a considerable space and time overhead, which is another reason why we have chosen to instrument our application manually in a more lightweight fashion by setting or unsetting a number of flags.

Although comprehensive performance tests in terms of the exact overhead of the instrumentation are yet to be carried

[3] http://logging.apache.org/
[4] https://openjdk.dev.java.net/svn/openjdk/jdk/trunk/j2se/src/share/classes/sun/security/ssl

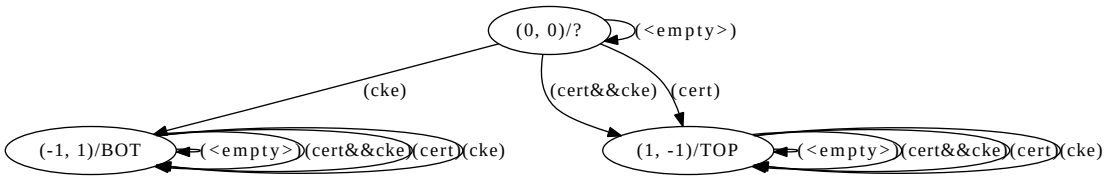

Fig. 9. Automatically generated FSM for $\varphi_1 = \neg$cke **W** cert

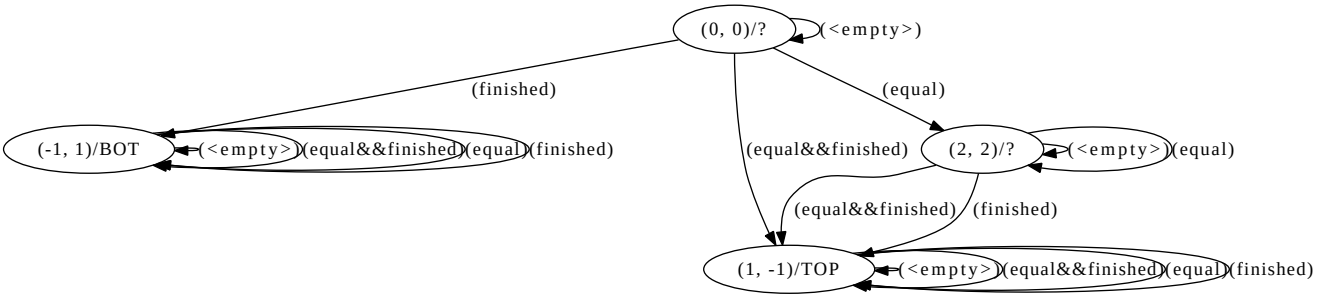

Fig. 10. Automatically generated FSM for $\varphi_2 = (\neg$finished **W** equal $\wedge$ (**F** equal $\Rightarrow$ **F** finished$))$

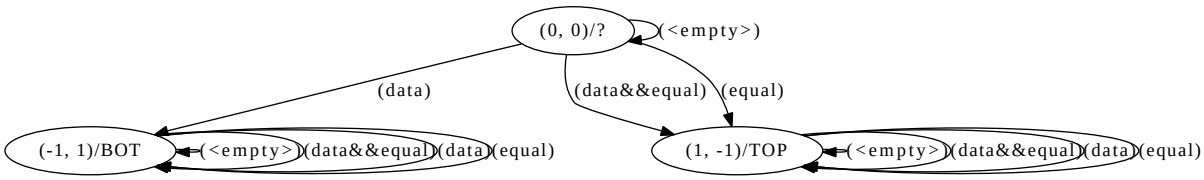

Fig. 11. Automatically generated FSM for $\varphi_3 =$ data **W** equal

out, we have collected some preliminary performance indicators using the comprehensive collection of LTL formulae [5] that is underlying Dwyer et al.'s commonly used *Software Specification Patterns* (cf. [27]). This collection of formulae, taken from real-world applications that employ formal specification using temporal logic, consists of a total of 447 formal specifications, of which 97 were given in terms of LTL. Although the 97 formulae were between 2 and 44 token in length, none of the generated (and minimised) monitors consisted of more than 6 states, which is at least an indication that the expected performance overhead for practical specifications, such as represented by Dwyer et al.'s formulae, is manageable, or even negligible as it was the case with our example properties.

Moreover, for the purposes of this paper, we could only demonstrate how to monitor a selected set of properties. Many more properties would be important to monitor, such as, for example, monitoring that the data stream in an SSL transaction is indeed encrypted.

Acknowledgements We thank the reviewers for constructive feedback which led to a significant improvement of this paper.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi Calculus. In *Fourth ACM Conference on Computer and Communications Security (CCS 1997)*, pages 36–47, 1997.

[2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.

[3] M. Alam, M. Hafner, and R. Breu. Model-driven security engineering for trust management in SECTET. *Journal of Software*, 2(1), Feb. 2007.

[4] B. Alpern and F. B. Schneider. Defining liveness. Technical report, Ithaca, NY, USA, 1984.

[5] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[6] S. Andova, C. J. F. Cremers, K. Gjøsteen, S. Mauw, S. F. Mjølsnes, and S. Radomirovic. A framework for compositional verification of security protocols. *Inf. Comput.*, 206(2-4):425–459, 2008.

[7] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer-Verlag, 2005.

[8] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Pri-*

[5] cf. http://patterns.projects.cis.ksu.edu/documentation/specifications/ALL.raw

vacy, pages 387–401. IEEE Computer Society, 2008.

[9] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In M. Aiello, M. Aoyama, F. Curbera, and M. P. Papazoglou, editors, *ICSOC*, pages 193–202. ACM, 2004.

[10] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006.

[11] A. Bauer and J. Jürjens. Security protocols, properties, and their monitoring. In B. D. Win, S.-W. Lee, and M. Monga, editors, *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems (SESS)*, pages 33–40, New York, NY, May 2008. ACM Press.

[12] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*. Accepted for publication.

[13] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 4337 of *LNCS*. Springer, Dec. 2006.

[14] G. Bella, L. C. Paulson, and F. Massacci. The verification of an industrial payment protocol: the set purchase phase. In V. Atluri, editor, *ACM Conference on Computer and Communications Security*, pages 12–20. ACM, 2002.

[15] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW*, pages 139–152. IEEE Computer Society, 2006.

[16] S. Braghin, A. Coen-Porisini, P. Colombo, S. Sicari, and A. Trombetta. Introducing privacy in a hospital information system. In Win et al. [52], pages 9–16.

[17] B. Braun. Save: static analysis on versioning entities. In Win et al. [52], pages 25–32.

[18] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In R. Büschkes and P. Laskov, editors, *DIMVA*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2006.

[19] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society, Series A*, 426(1871):233–271, December 1989. Also appeared as SRC Research Report 39 and, in a shortened form, in *ACM Transactions on Computer Systems 8*, 1:18–36 (February 1990).

[20] L. Cavallaro, A. Lanzi, L. Mayer, and M. Monga. Lisabeth: automated content-based signature generator for zero-day polymorphic worms. In Win et al. [52], pages 41–48.

[21] I. Chowdhury, B. Chan, and M. Zulkernine. Security metrics for source code structures. In Win et al. [52], pages 57–64.

[22] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46. IEEE Computer Society, 2005.

[23] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[24] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, Washington, DC, USA, 2008. IEEE Computer Society.

[25] S. Colin and L. Mariani. Run-time verification. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer, 2004.

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

[27] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proc. 2nd WS. on Formal methods in software practice*, pages 7–15. ACM, 1998.

[28] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *IEEE Symposium on Security and Privacy*, pages 214–229. IEEE Computer Society, 2006.

[29] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *ICSE*, pages 458–467. IEEE Computer Society, 2007.

[30] M. Gegick and L. Williams. On the design of more secure software-intensive systems by use of attack patterns. *Information & Software Technology*, 49(4):381–397, 2007.

[31] M. Geilen. On the construction of monitors for temporal logic properties. *ENTCS*, 55(2), 2001.

[32] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI'05*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[33] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology*, pages 83–96, London, UK, 1994. Springer-Verlag.

[34] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[35] K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

[36] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Journal on Software Tools for Technology Transfer*, 2004.

[37] V. Horvath and T. Dörges. From security patterns to implementation using petri nets. In Win et al. [52], pages 17–24.

[38] J. Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In S. Easterbrook and S. Uchitel, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. ACM, 2006.

[39] J. Jürjens and M. Yampolskiy. Code security analysis with assertions. In D. Redmiles, T. Ellman, and A. Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages

392–395. ACM, 2005.

[40] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, Spring 1994.

[41] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems with applications to history-based access control. In *CCS*, pages 260–269. ACM, 2005.

[42] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[43] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[44] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.

[45] G. Spanoudakis, C. Kloukinas, and K. Androutsopoulos. Towards security monitoring patterns. In *SAC*, pages 1518–1525. ACM, 2007.

[46] G. Stenz and A. Wolf. E-SETHEO: An automated^3 theorem prover. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000)*, volume 1847 of *Lecture Notes in Computer Science*, pages 436–440. Springer-Verlag, 2000.

[47] S. G. Stubblebine and R. N. Wright. An authentication logic with formal semantics supporting synchronization, revocation, and recency. *IEEE Trans. Software Eng.*, 28(3):256–285, 2002.

[48] UMLsec tool, 2001-08. http://mcs.open.ac.uk/jj2924/umlsectool.

[49] LTL_3 Tools, 2008. http://ltl3tools.SourceForge.Net/.

[50] D. Vanoverberghe and F. Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In G. Barthe and F. S. de Boer, editors, *FMOODS*, volume 5051 of *Lecture Notes in Computer Science*, pages 240–258. Springer, 2008.

[51] M. Westhead and S. Nadjm-Tehrani. Verification of embedded systems using synchronous observers. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 405–419, London, UK, 1996. Springer-Verlag.

[52] B. D. Win, S.-W. Lee, and M. Monga, editors. *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems, SESS 2008, Leipzig, Germany, May 17-18, 2008*. ACM, 2008.

[53] B. D. Win, B. Vanhaute, and B. D. Decker. How aspect-oriented programming can help to build secure software. *Informatica (Slovenia)*, 26(2), 2002.

[54] L. Yu, R. B. France, I. Ray, and K. Lano. A light-weight static approach to analyzing UML behavioral properties. In *ICECCS*, pages 56–63. IEEE Computer Society, 2007.