Security Protocols, Properties, and their Monitoring

Andreas Bauer
Computer Sciences Laboratory
Australian National University
baueran@rsise.anu.edu.au

Jan Jürjens
Department of Computing
The Open University, UK
http://www.jurjens.de/jan/

ABSTRACT

This paper examines the suitability and use of runtime verification as means for monitoring security protocols and their properties. In particular, we employ the runtime verification framework introduced in [5] to monitor complex, history-based security-properties of the SSL-protocol. We give a detailed account of the methodology, compare its formal expressiveness to prior art, and describe its application to an open-source Java-implementation of the SSL-protocol. In particular, we show how one can make use of runtime verification to dynamically enforce that assumptions on the crypto-protocol implementations (that are commonly made when statically verifying crypto-protocol specifications against security requirements) are actually satisfied in a given protocol implementation at runtime. Our analysis of these properties shows that some important runtime correctness properties of the SSL-protocol exceed the commonly used class of safety properties, and as such also the expressiveness of other monitoring frameworks.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, formal methods, model checking, validation*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls, cryptographic controls, verification*

General Terms

Security

Keywords

Security protocols, monitoring history-based properties, security automata, language-based security, runtime verification, temporal logic

1. INTRODUCTION

With respect to crypto-based software (such as crypto-protocols), a lot of successful work has been done to formally analyse abstract specifications of these protocols for security design weaknesses. What is still largely missing is an approach which provides assurance for implementations of crypto-based systems against security weaknesses. This is necessitated by the fact that so far, crypto-based software is 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 crypto-algorithms.

In this paper, we show how this kind of assurance can be provided at runtime by making use of a methodology called *runtime verification* for monitoring systems. We make use of a framework for runtime verification which allows one to monitor both *safety* and *co-safety* properties [5] (which goes beyond previous work on monitoring security properties such as [24]). Being able to use co-safety properties allows us to specify and monitor properties such as that a certain security-providing action has been performed before another security-sensitive action is executed (for example, the authentication has been successfully performed before the session key is exchanged). By combining this runtime verification for the protocol implementation with a static Dolev-Yao type verification of the protocol specification (which can be given, e. g., as a UML model which can be verified with tools such as [17, 28]), the combination of the two approaches allows us to ensure that Dolev-Yao type security properties will be enforced at runtime. We explain our approach at the hand of JESSIE, an open-source implementation of the Java Secure Socket Extension (JSSE).

Moreover, unlike other, more ad-hoc approaches on systems monitoring and in particular on security monitoring, runtime verification has a rigorous formal semantics, which is based upon temporal logic. Many systems, especially protocols, are nowadays specified using temporal logic to enable formal reasoning about their properties. For this purpose, the designer of a protocol specifies not only a model of the protocol itself, but also parts of its intended functionality and valid behaviour in terms of temporal logic formulae, which can then be verified w. r. t. the protocol's model. This procedure is common practice today, and many successful examples can be found in academia and industry alike (cf. [16, 8, 9, 7]). In particular, the *linear-time temporal logic* (LTL, [21]) has found wide acceptance in industrial practice, and a derivation of it, called *Property Specification Language* (PSL), has recently been standardised by the IEEE under IEEE1850 [12]. Hence, by employing runtime verification, temporal logic specifications become useful not only to statically reason about a system, but also to directly check for violations of properties at runtime, when the system operates in nondeterministic environments, or with users in security and otherwise mission critical environments. As such, expert knowledge used to build systems can be reused also to verify them at runtime. Even collections of formal patterns of LTL formulae nowadays exist [9] that further facilitate specification of frequently reoccurring system requirements, similar to *security or security monitoring patterns* [27, 25].

Plan for this paper.

In the next section, we provide a brief overview over the fundamentals of runtime verification and on temporal logic as a formalism to specify system properties. In Sec. 3, we discuss how the runtime verification framework we use to verify an open source implementation of the SSL-protocol, exceeds other formal monitoring approaches, and discuss its expressiveness, in general. As it turns out, many of the properties we need to monitor in our application are so-called co-safety properties, or an intersection of several different classes of properties, which previous approaches to security monitoring (cf. [24]) did not cover. In Sec. 4, we discuss our application in more detail, state some important properties about it, and outline the realisation of runtime verification in this scenario. The remaining two sections discuss related work and conclusions of this paper.

2. RUNTIME VERIFICATION

Monitoring of systems is by no means a new technique, and many applications in different domains exist. In practice, monitoring is often reduced to performing "sanity checks" at runtime while a system executes, such as building residuals, check-sums, or other single-state based assertions. That is, either some system action leads to a "bad state", or not, irregardless of the history of actions.

One can consider a wide spectrum of approaches, ranging from such simple predicate assertions stating properties about single states at single system locations, to more complex, temporal, or *history-based* assertions, stating properties about temporally separated states or events at multiple program locations. From an implementation point of view, the latter is clearly inspired by works of *model checking* [8] temporal specifications and, at the same time, the predominant approach to formal *runtime verification* as encountered in the literature (cf. [13, 15, 5, 6]).

In runtime verification, we are given a correctness property φ about a system, and the system as a "black-box", meaning that no explicit system model is required. Then, from φ a so-called *monitor* is automatically generated, which observes the executions of the running system. If the observed system behaviour does not satisfy φ, an alarm is raised, signalling to the operator or user of the system that some undesired system state has been reached.

Examples of φ include properties stating that a certain series of system operations is prohibited in a specific system mode (i. e., access control), or that certain resources are available to certain users (i. e., availability), and so forth.

In order to being able to generate monitors automatically from a property, it needs to be defined formally. A predominant formalism used in the literature is temporal logic, and for monitoring, *linear-time temporal logic* (LTL) as originally introduced by Pnueli [21].

Hence, let us first briefly recall some basic definitions about LTL. Let AP be a nonempty set of propositions, and $\Sigma = 2^{AP}$ be an *alphabet*. 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 $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$.

DEFINITION 1 (LTL SYNTAX AND SEMANTICS). *The syntax of LTL over Σ, written LTL(Σ), is inductively defined as follows.*

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

Let $\varphi \in LTL(\Sigma)$ be an LTL formula, and $i \in \mathbb{N}$ denote a position. The semantics of LTL *formulae is then defined inductively over infinite strings $w \in \Sigma^\omega$ as follows:*

$$\begin{array}{lcl} w, i \models true & & \\ w, i \models \neg\varphi & \Leftrightarrow & w, i \not\models \varphi \\ w, i \models p \in AP & \Leftrightarrow & p \in w(i) \\ w, i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & w, i \models \varphi_1 \vee w, i \models \varphi_2 \\ w, i \models \varphi_1 \boldsymbol{U} \varphi_2 & \Leftrightarrow & \exists k \geq i.\, w, k \models \varphi_2 \wedge \\ & & \forall i \leq l < k.\, w, l \models \varphi_1 \\ w, i \models \boldsymbol{X}\varphi & \Leftrightarrow & w, i+1 \models \varphi \end{array}$$

We write $w \models \varphi$, if and only if $w, 0 \models \varphi$, and use $w(i)$ to denote the ith element in w.

Further, as is also common, we use $\mathbf{F}\varphi$ short for $true\mathbf{U}\varphi$ ("eventually φ"), $\mathbf{G}\varphi$ short for $\neg\mathbf{F}\neg\varphi$ ("globally φ"), and $\varphi_1 \mathbf{W} \varphi_2$ short for $\mathbf{G}\varphi_1 \vee (\varphi_1 \mathbf{U} \varphi_2)$ (weak-until). For brevity, whenever Σ is clear from the context, we also 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.

EXAMPLE 1. *Let us give some intuitive yet abstract examples of LTL specifications, and let $\varphi_i \in LTL$, and $p \in AP$. For instance, $\boldsymbol{GF}p$ asserts that always p will eventually occur. $\varphi_1 \boldsymbol{U} \varphi_2$ states that φ_1 holds until φ_2 holds, and φ_2 will eventually hold. On the other hand, $\boldsymbol{G}p$ asserts that always p is true on a given trace.*

For each formula $\varphi \in$ LTL, we can construct a so-called *Büchi automaton $\mathcal{A}^\varphi$*, such that the accepted language of the automaton, $\mathcal{L}(\mathcal{A}^\varphi)$, consists of all the models of φ, i. e., $\mathcal{L}(\varphi)$. This construction is known to be exponential in the size of the formula (cf. [14]).

Büchi automata are structurally equivalent to standard finite automata, but accept infinite words, which are also the models for LTL formulae. Moreover, Büchi automata are at the heart of many analyses that are peformed on LTL specifications, such as model checking; that is, language-emptiness checks and language-inclusion are well-understood and decidable problems. Notice, however, that nondeterministic Büchi automata are strictly more expressive than deterministic ones. Hence, we cannot, in general, convert the nondeterministic ones into deterministic ones for all formulae in LTL alike. This is one of the problems, which has to be addressed in runtime verification of LTL formulae.

We can map a system's behaviour to individual *actions* that are formally captured by the alphabet Σ. A series of actions then corresponds to a string or word w of a formal language $L \subseteq \Sigma^\infty$, written $w \in L$.

Moreover, given $\varphi \in$ LTL, we can construct a monitor M^φ which reads a prefix of a possibly infinite word $w \in \Sigma^\omega$, written $u \prec w$, where $u \in \Sigma^*$, telling whether $uv \models \varphi$, $uv \not\models \varphi$, or neither. Notice, $v \in \Sigma^\omega$ is an infinite extension to u, respectively.

The monitor device itself, which can also be specified in terms of a function $M^\varphi : \Sigma^* \to \{\top, \bot, ?\}$, is derived from the Büchi automaton of φ, but with additional modifications to its structure and acceptance condition to cater for the finite-trace interpretation of LTL. As such, a monitor is, but a finite state machine (FSM) interpreting whether or not the finite prefixes of some infinite word are sufficient to deduce the violation ($\bot$), or satisfaction of φ ($\top$), or neither (?).

In fact, closing, or at least narrowing the gap between temporal logic models, which are infinite structures, and finite traces of system observations is a major challenge that gets addressed in almost all runtime verification approaches alike. We omit the technical details at this point, but refer the reader to [5, 6, 10] instead. What is essential for our purpose is to know that there exist finite trace

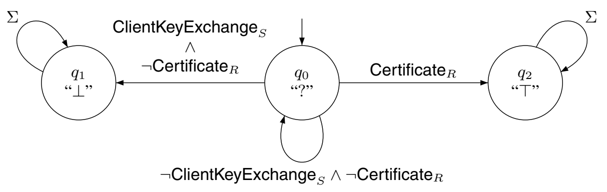

Figure 1: FSM ¬ClientKeyExchange$_S$UCertificate$_R$.

intepretations for LTL, such that monitors can be built according to the above sketched ideas. Moreover, many of the monitor constructions known from the literature do, in fact, rely on the LTL-to-Büchi conversion as a first step towards a getting a FSM, i. e., monitor.

EXAMPLE 2. *In case of our SSL-application, one security property to monitor is that the client will not send out the* ***ClientKeyExchange*** *message before it has received the* ***Certificate*** *message from the server, has performed the validity check for the certificate as specified in Fig. 3, and this check turned out to be positive (see Sec. 4). Hence, we have a set of propositions*

$$AP = \{\mathit{ClientKeyExchange}_S, \mathit{Certificate}_R\},$$

a specification $\varphi = \neg\mathit{ClientKeyExchange}_S \mathbf{U} \mathit{Certificate}_R$*, and obtain indirectly via the Büchi automaton construction, the FSM depicted in Fig. 1. (For brevity, the intermediate steps in the construction are omitted at this point.) Each reachable state has an output symbol associated, stating whether* φ *is satisfied, violated, or neither. Labels on transitions indicate which symbols in the monitor's input trigger a transition.* Σ *on a loop means that any input triggers the corresponding transition. For instance, an observed trace* $u = \emptyset\emptyset\ldots\{\mathit{Certificate}_R\}$ *would lead to satisfaction of* φ*, whereas* $u' = \{\mathit{ClientKeyExchange}_S\}$ *would lead to violatation of* φ*.*

Although the property to be monitored is also rather "simple", the example highlights and summarises the general ideas behind runtime verification. Notice that Schneider's *Security Automata* [24] are a special instance of runtime verification, and are discussed in greater detail in the next section. As is the case in runtime verification, security automata are directly derived from the Büchi automaton of a temporal logic formula.

3. EXPRESSIVENESS

In his seminal paper [24], Schneider introduced the concept of *Execution Monitoring (EM) Enforceability* and that of *Security Automata* for monitoring security policies. He also gives a classification of the properties that can be monitored using security automata, and puts forward the application areas of *access control* and *availability*, which we briefly mentioned in the previous section. The reason for Schneider's choices lies in that properties describing access control or availability violations correspond to, essentially, *safety languages*, a concept which we are going to introduce and review in this section from a temporal-logic point of view.

DEFINITION 2 (SAFETY, BAD PREFIX). *Let* $L \subseteq \Sigma^\omega$ *be a language on infinite words over alphabet* Σ*. A prefix* $u \in \Sigma^*$ *is a* bad prefix *for L, if for every* $w' \in \Sigma^\omega$ *the following holds:* $uw' \notin L$*.*

If $\forall w \in \Sigma^\omega \backslash L.\ \exists u \in \Sigma^*$ *such that* $w = uw'$ *for some* $w' \in \Sigma^\omega$*, and* $\forall w'' \in \Sigma^\omega.\ uw'' \notin L$*, L is a* safety language *(also called a safety property).*

Clearly, L can be expressed, for instance, by any formula in LTL, φ. In other words, if all models of φ, captured by the set $\mathcal{L}(\varphi)$, span a safety language, then all infinite words that are not part of the language, i. e., which violate φ, must have a finite bad prefix. After reading this prefix, it is clear that any (infinite) extension to this prefix must also violate φ. Hence, safety properties are usually considered to being suitable for monitoring, where only finite prefixes of system behaviour can be observed. Notice, we adopt the notation $u \prec w$ to say that u is a prefix of w.

In [22], the notion of a *monitorable* property is introduced which, basically, subsumes all properties that can be violated or satisfied upon reading a finite prefix. Let us adopt this notion for our purposes, and let us point out that a more detailed account of this concept is also available in [6].

Moreover, since security automata can be understood as a special kind of runtime verification (runtime verification subsumes other types of languages, but safety as well), we are going to review non-safety languages for their suitability to express properties of security-critical applications (see also Lemma 1).

DEFINITION 3 (LIVENESS). *Let* $L \subseteq \Sigma^\omega$ *be called a* liveness language*, if for all prefixes* $u \in \Sigma^*$ *the following holds:* $\exists w \in \Sigma^\omega : uw \in L$*.*

In an early work, Lamport pointed out that all "interesting properties" about systems could be expressed using safety and liveness languages [20]. For static verification techniques, such as model checking, this does not impose any problems, but for monitoring, liveness properties can impose a problem: they may require that both the models as well as the counterexamples of a property are infinite structures without bad, respectively good (see also Definition 4), prefixes.

However, there are interesting language classes "in between" the safety and liveness hierarchy, which either have finite models or finite counterexamples, or even both. Arguably, the most important class is that of a *co-safety language*.

DEFINITION 4 (CO-SAFETY, GOOD PREFIX). *Let* $L \subseteq \Sigma^\omega$*. A prefix* $u \in \Sigma^*$ *is a* good prefix *for L, if for all* $w' \in \Sigma^\omega$ *the following holds:* $uw' \in L$*.*

If for all $w \in L$*, there exists a good prefix* $u \in \Sigma^*$*, such that* $w = uw'$ *with* $w' \in \Sigma^\omega$*, then L is called a* co-safety language*.*

Moreover, there exists a set-theoretic categorisation of languages in the so-called *Cantor space* or topology (cf. [3]), which explains the tight connection between safety and co-safety languages. In a nutshell, in this topology, safety languages correspond to closed sets. Consequently, the negation of a safety property corresponds to an open set, which in turn, describes a co-safety property. Again, in informal terms, if L is a safety language, then $\overline{L}$, the complement of L, is a co-safety language.

The practical implication of this is that although liveness languages are generally unsuitable for monitoring, there exist subclasses which are, indeed, monitorable as captured by the following proposition.

PROPOSITION 1. *There are liveness properties which have a finite good prefix. These properties are co-safety properties. Not all co-safety properties are liveness properties.*

PROOF. (Sketch) Recall, liveness of L asserts $\forall u \in \Sigma^*.\ \exists w \in \Sigma^\omega.\ uw \in L$. Now, consider the property $\varphi = \mathbf{F}p$, and let $\Sigma^* p \Sigma^\omega$ be a regular expression denoting the set of all words that begin with arbitrary but finitely many symbols from Σ, followed by p, and then followed again by infinitely many symbols from Σ. Clearly,

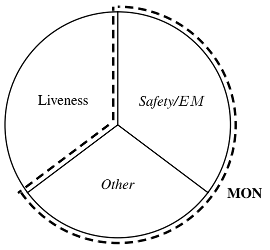

Figure 2: Different language classes

$\mathcal{L}(\varphi) = \mathcal{L}(\Sigma^* p \Sigma^\omega)$. Moreover, the regular expression intersects with the definitions of both liveness and co-safety: The $\forall u \in \Sigma^*$ in the liveness definition matches the initial Σ^* of the regular expression, and $p\Sigma^\omega$ matches the $\exists w \in \Sigma^\omega .\ uw \in L$. Co-safety is covered, because all infinite words do, indeed, have a finite good prefix expressed as $\Sigma^* p$, after which "anything" can be added.

For the second part of the proposition, let us consider a property $\varphi' = a\mathbf{U}b$. Again, it is easy to see that this is a co-safety property since all finite words containing a's until a b occurs can be arbitrarily extended, similarly as above. However, φ' is not a liveness property. Consider the following trace: $u' = \emptyset$, i. e., neither a nor b hold. Clearly, u' cannot be extended to anything satisfying φ', thus violating the definition of liveness. □

Schneider called the safety languages, the enforcable properties, which can be monitored/enforced using security automata as can be seen from the following contraposition:

PROPOSITION 2 ([24]). *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.*

Obviously, the rationale for this is that violations of safety properties can be detected after reading a *finite* stream of system events. This, for example, is not possible when trying to enforce the liveness property $\mathbf{GF}p$ as is pointed out above—its models as well as counterexamples are both infinite words without distinguishing prefixes. We leave the substantiation of this claim as an excercise to the reader.

The additional expressiveness as compared to Prop. 2 that we gain from using runtime verification is captured in the following lemma as well as in Fig. 2, which exemplifies our findings. Notice that in the figure, the area labelled "liveness" represents the strict liveness properties, i. e., those not including co-safety, and vice versa for "safety".

LEMMA 1. *Let EM be the enforceable properties, and MON be the monitorable properties, then $EM \subset MON$.*

PROOF. (Sketch) Above, we have set MON to be the properties which can be either violated, satisfied, or both, using a finite prefix. Clearly, the lemma holds, since EM corresponds to safety, and we have seen, for example, that co-safety can be satisfied using a finite prefix. □

The security properties of the SSL-protocol that we are considering in the remainder are within the class MON of properties.

4. RUNTIME SECURITY PROPERTIES OF THE SSL-PROTOCOL

We apply the approach sketched above to the implementation of the Internet security protocol SSL in the project JESSIE, which is an open-source implementation of the Java Secure Sockets Extension (JSSE). The whole JESSIE project currently consists of about 5 MB of code, but the part directly relevant to SSL consists of less than 700 KB in about 70 classes.

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

Figure 4: Data for the Handshake message

4.1 The SSL Protocol

SSL is the de facto standard for securing http connections, which however has been the source of several significant security vulnerabilities in the past [1] and is therefore an interesting target for a security analysis. In this paper, we concentrate on the fragment of SSL that uses RSA as the cryptographic algorithm and provides server authentication (cf. Fig. 3).

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 cryptographic algorithms. 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 formalized 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 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().

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 Fig. 3. 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 Fig. 4.

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

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

Figure 5: Constructor for random

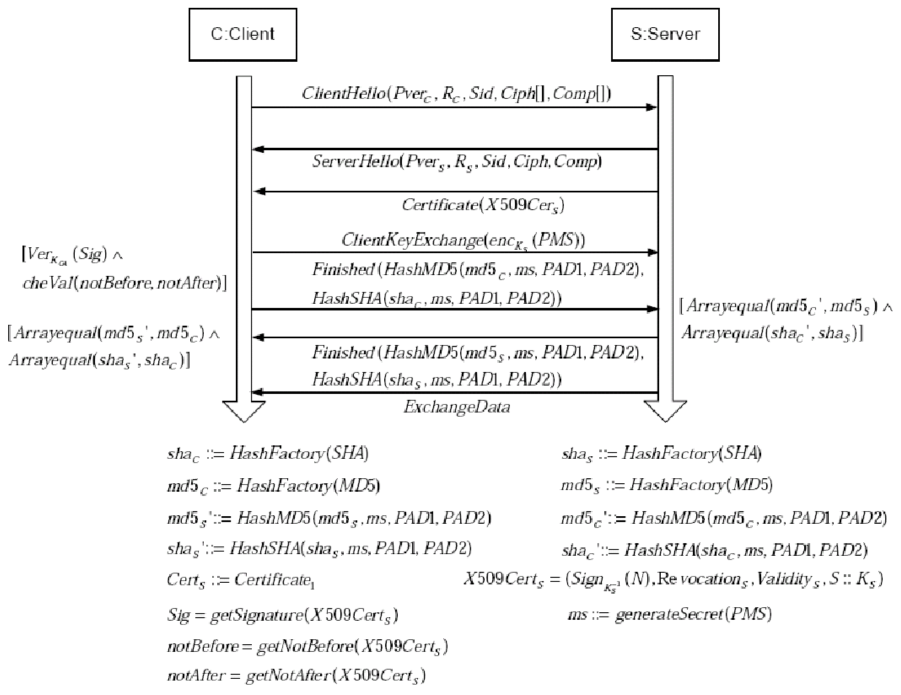
C:Client
S:Server
$ClientHello(Pver_C, R_C, Sid, Ciph[], Comp[])$
$ServerHello(Pver_S, R_S, Sid, Ciph, Comp)$
$Certificate(X509Cer_S)$
$ClientKeyExchange(enc_{K_S}(PMS))$
$[Ver_{K_{CA}}(Sig) \wedge cheVal(notBefore, notAfter)]$
$Finished(HashMD5(md5_C, ms, PAD1, PAD2), HashSHA(sha_C, ms, PAD1, PAD2))$
$[Arrayequal(md5_C', md5_S) \wedge Arrayequal(sha_C', sha_S)]$
$[Arrayequal(md5_S', md5_C) \wedge Arrayequal(sha_S', sha_C)]$
$Finished(HashMD5(md5_S, ms, PAD1, PAD2), HashSHA(sha_S, ms, PAD1, PAD2))$
$ExchangeData$
$sha_C ::= HashFactory(SHA)$
$md5_C ::= HashFactory(MD5)$
$md5_S' ::= HashMD5(md5_S, ms, PAD1, PAD2)$
$sha_S' ::= HashSHA(sha_S, ms, PAD1, PAD2)$
$Cert_S ::= Certificate_1$
$Sig = getSignature(X509Cert_S)$
$notBefore = getNotBefore(X509Cert_S)$
$notAfter = getNotAfter(X509Cert_S)$
$sha_S ::= HashFactory(SHA)$
$md5_S ::= HashFactory(MD5)$
$md5_C' ::= HashMD5(md5_C, ms, PAD1, PAD2)$
$sha_C' ::= HashSHA(sha_C, ms, PAD1, PAD2)$
$X509Cert_S = (Sign_{K_S^{-1}}(N), Revocation_S, Validity_S, S :: K_S)$
$ms ::= generateSecret(PMS)$

Figure 3: The cryptographic protocol implemented in SSLSocket.java

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

Figure 6: Data in ClientHello message

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 Fig. 5).

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), all data that is sent and received must be investigated. In Fig. 6, 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.

4.2 Monitoring security properties

A crypto-protocol like the one specified in Fig. 3 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 [17] 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 his 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.

One can then use runtime verification to make sure that the implementation correctly and securely implements the specification, and in particular implements the security properties previously demonstrated at the specification level.

For this we first need to determine how important elements at the model level are implemented at the implementation level. 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, it first needs to be identified at which points in the implementation messages are received and sent out, and which messages exactly. To be able to do this, we exploit the fact that in many implementations of crypto-protocols, message communication is implemented in a standardized way (which can be used to recognise 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 (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 in storing it into variables of the appropriate types. 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, the messages themselves are usually represented by message classes that offer write and read methods and in which the write and read methods from the java.io are called.

According to the information that is contained in a sequence diagram specification of a crypto-protocol, the runtime verification needs to keep track of the following information: 1. *Which data is sent out?* and 2. *Which data is received?*

The runtime checks will enforce that the relevant part of the implementation conforms to the specification in the following sense. 1. *The code should only send out messages that are specified to be sent out according to the specification and in the correct order*, and 2. *these messages should only be sent out if the conditions that have to be checked first according to the specification are met*.

Some examples for such properties in the case of the SSL-protocol specified in Fig. 3 are given by the following requirements that arise from the above discussion:

1. The client will not send out the ClientKeyExchange message until it has received the Certificate message from the server, has performed the validity check for the certificate as specified in Fig. 3, and this check turned out to be positive. If that is the case, the client will indeed send out the ClientKeyExchange message eventually.
2. The server will not send the Finished message to the client before the MD5 hash received from the client in the Finished message 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.
3. The client will not send any transport data to the server before the MD5 hash received from the server in the Finished message has been checked to be equal to the MD5 created by the client, and correspondingly for the SHA hash.

In the concluding example of Sec. 2, we have already discussed a formalisation of the first property, namely,

$$\varphi_1 = \neg \mathsf{ClientKeyExchange}_S \mathbf{U} \mathsf{Certificate}_R$$

where $\{\mathsf{ClientKeyExchange}_S, \mathsf{Certificate}_R\} \subseteq AP$ is the set of atomic propositions that match the sending and receiving events of ClientKeyExchange and Certificate. Recall that the stream of events processed by our monitor consists of elements from 2^{AP}; that is, at each point in time, the application keeps track of both events the sending of ClientKeyExchange and the receiving of Certificate. If none of the events was observed, the according propositions are interpreted as $\bot$, otherwise as $\top$. Moreover, φ_1 is a classical co-safety property. For the actual monitor of this property as well as examples for violating as well as satisfying observed system behaviour, see Sec. 2.

Let us examine the second property as given above. It involves comparison of values and function calls. Since this cannot be modelled using LTL directly, we instead adapt the set of atomic propositions as follows. Let

$$\{\mathsf{Finished}_S, (MD5(\mathsf{Finished}_R) = MD5(\mathsf{Finished}_S)),$$
$$(SHA(\mathsf{Finished}_R) = SHA(\mathsf{Finished}_S))\} \subseteq AP.$$

In other words, we define two propositions that are interpreted as $\top$, if and only if the equality condition holds, which we have to check in the code of our application in terms of comparing the MD5 and SHA hash values. The corresponding property w. r. t. AP is then

$$\varphi_2 = (\neg \mathsf{Finished}_S \mathbf{W} (MD5(\mathsf{Finished}_R) = MD5(\mathsf{Finished}_S)))$$
$$\wedge (\mathbf{F}(MD5(\mathsf{Finished}_R) = MD5(\mathsf{Finished}_S)) \Rightarrow \mathbf{F}\mathsf{Finished}_S),$$

and we assume some φ_2' where all occurrences of the proposition $(MD5(\mathsf{Finished}_R) = \ldots)$ are replaced by the proposition $(SHA(\mathsf{Finished}_R) = \ldots)$ from AP, respectively. Hence, we create two monitors for this security requirement: one for φ_2 and another one for φ_2'. Notice, φ_2 and φ_2' are neither strictly safety nor strictly co-safety properties. For instance, consider a trace $u = \emptyset\{\mathsf{Finished}_S\}$, which violates the first part of our conjunction since $\mathsf{Finished}_S = \top$, but $(MD5(\mathsf{Finished}_R) = \ldots) = \bot$. On the other hand, the trace $v = \emptyset\{(MD5(\mathsf{Finished}_R) = \ldots)\}$ is a model for φ, since our second observation in v shows that the MD5 checksum was successfully compared, and until then, $\mathsf{Finished}_S = \bot$ held. Recall $\emptyset$ means that all propositions are interpreted as $\bot$. φ_2 is not co-safety since there exists the infinite model $v' = \emptyset\emptyset\ldots$ without a good prefix, i. e., $\mathsf{Finished}_S$ never holds. Moreover, it is not safety since, there exists the infinite counterexample $u' = \{(MD5(\mathsf{Finished}_R) = \ldots)\}\emptyset\emptyset\ldots$ without a bad prefix.

Finally, requirement 3. can be formalised as follows.

$$\varphi_3 = \neg Data \mathbf{W}((MD5(\mathsf{Finished}_R) = MD5(\mathsf{Finished}_S)),$$

where AP is as in the previous example, but additionally contains an action, indicating the sending of data, $Data$. Now we have a safety property since all traces of violating behaviour for φ_3 are finite, or in other words: there exist no infinite counterexamples that cannot be recognised with a bad prefix. 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 Finished 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.

Notice that monitoring strict safety properties, such as $\mathbf{G}p$, 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 finite models, hence the corresponding monitor can output all three values of $\{\top, \bot, ?\}$, depending on the observed system behaviour. The same holds for φ_1, although it is strictly a co-safety property.

Code instrumentation & realisation.

Technically, our Java implementation of the SSL-protocol needs to set or unset the atomic propositions as system events are created, e. g., by the sending and receiving of messages, or by the outcomes of comparing actual values with reference values, and so forth. Our monitors then process the resulting stream of actions, in that the setting or the unsetting of propositions creates a new action, each time this occurs. As such, the notion of a next-state as asserted by the LTL-operator **X** is somewhat misleading in our application, since it does not operate synchronously to some global discrete clock. Hence, we can actually chose the so-called "next-free" fragment of LTL, and point out that our properties are then closed under "stuttering" (cf. [11]). But this is a minor technical detail.

Once the monitors are generated for all relevant specifications, the only code that needs to be added to the main application is the code to set or unset propositions, and the code for handling communication between the monitors and the application itself.

Notice that other comprehensive Java programs are often developed in parallel using event logging libraries such as log4j, which can then directly be used for capturing all relevant system events that the monitors require. If no such library is used, as in our example, then the set of relevant events needs to be identified first, and all occurrences in the code be instrumented, accordingly.

5. FURTHER RELATED WORK

We have already pointed out the significance of security automata as described in [24], and their relation to other works on formal verification and languages. In [19] Krukow et al. describe the use of runtime verification for *access control* in *reputation systems*, such as realised in parts by ebay.com. One of their main contributions is a quantification extension to standard LTL, and to show means to enforce properties expressed in this logic. However, unlike in our framework, they employ a different fragment of LTL, which allows one to "look into the past" rather than the future and present. Although both logics are semantically equivalent, we would like to point out that our fragment is much closer to already existing and standardised temporal logic verification tools than the one by Krukow et al. A similar although less technical approach to access control is also described in [23]. Moreover, there exists a large body of work in using runtime verification in combination with *aspect-oriented programming*, and many such applications employ the Java programming language as a means for implementation (cf. [2]). Since our case study is at the moment at a prototypical stage, we believe that our ad-hoc approach to code instrumentation could well benefit from some of the ideas presented in this area. Work on model-based runtime checking of security permissions is reported in [18]. That work does not use LTL but uses UML sequence diagrams and statecharts as the specification notation (and the emphasis of that work is on security permissions rather than crypto-protocols).

6. CONCLUSIONS

Runtime verification allows us to monitor even complex, history-dependent specifications as they arise e.g. 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 fall into the class MON of properties that can be monitored by our approach but not, e.g., in the class EM of properties that can be monitored using Schneider's security automata.

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, due to [4, 26] it is known that given some formula $\varphi \in$ LTL, deciding whether $\mathcal{L}(\varphi)$ is safety (co-safety) is a PSPACE-complete problem. Hence, there are practical 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.

Once the monitors are generated, then the resulting overhead from using monitors is minimal. The particular approach described in [5], in fact, creates monitors with *optimal* space complexity w. r. t. 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 affect runtime efficiency.) 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 lightweight fashion.

7. REFERENCES

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

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, pages 345–364. ACM, 2005.

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

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

[5] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *FSTTCS*, volume 4337 of *LNCS*. Springer-Verlag, Dec. 2006.

[6] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. Technical Report TUM-I0724, Institut für Informatik, Technische Universität München, Dec. 2007.

[7] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Advances in Computers*, chapter Bounded model checking. Academic Press, 2003.

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

[9] 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.

[10] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on

truncated paths. In W. A. H. Jr. and F. Somenzi, editors, *CAV*, volume 2725 of *LNCS*, pages 27–39. Springer-Verlag, 2003.

[11] K. Etessami. Stutter-invariant languages, omega-automata, and temporal logic. In *CAV*, volume 1633 of *LNCS*, pages 236–248. Springer-Verlag, 1999.

[12] H. Foster, E. Marschner, and Y. Wolfsthal. IEEE 1850 PSL: The next generation. In *DVCon*, 2005.

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

[14] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th IFIP WG6.1 Intl. Symp. Protocol Specification, Testing and Verification*, pages 3–18. Chapman & Hall, 1996.

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

[16] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, 1991.

[17] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *ICSE*, 2005.

[18] J. Jürjens. Model-based run-time checking of security permissions using guarded objects. In *Run-time Verification*, Lecture Notes in Computer Science, 2008.

[19] 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.

[20] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

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

[22] A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In *FM*, volume 4085 of *LNCS*, pages 573–586. Springer, 2006.

[23] A. Pretschner, M. Hilty, and D. A. Basin. Distributed usage control. *Commun. ACM*, 49(9):39–44, 2006.

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

[25] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2006.

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

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

[28] Umlsec tool, 2001-08. http://computing-research.open.ac.uk/jj/ umlsectool.