Automated Verification of Security Protocol Implementations

Sagar Chaki and Anupam Datta

January 30, 2008
CMU-CyLab-08-002

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Automated Verification of Security Protocol Implementations

Sagar Chaki[1], Anupam Datta[2],

[1] Software Engineering Institute, CMU, Pittsburgh, USA
[2] CyLab, Carnegie Mellon University, Pittsburgh, USA

Abstract. We present a method that combines software model checking with a standard protocol security model to provide meaningful security analysis of protocol implementations in a completely automated manner. Our approach incorporates a standard *symbolic attacker model* and provides analogous guarantees about protocol implementations as previous work does for protocol specifications. We have implemented our approach and verified authentication and secrecy properties of an industrial strength protocol implementation – OpenSSL – for configurations consisting of up to 3 servers and 3 clients. We have also implemented two distinct methods for reasoning about attacker message derivations and present their comparison in the context of OpenSSL verification.

1 Introduction

Protocols that enable secure communication over an untrusted network constitute a critical part of the current computing infrastructure. Common examples of such protocols are SSL [18], TLS [16], Kerberos [23], and the IPSec [22] and IEEE 802.11i [1] protocol suites. The design and security analysis of such network protocols presents a difficult problem. In several instances, serious security vulnerabilities were uncovered in protocols many years after they were first published or deployed [25, 27, 21, 28, 11]. Over the last three decades, a variety of highly successful methods and tools have been developed for analyzing the security guarantees provided by network protocol *specifications* [10, 15, 4, 3, 24, 26, 35, 32, 25, 29, 34]. All of these approaches use a standard symbolic model of protocol execution and attack, developed from positions taken by Needham-Schroeder [30], Dolev-Yao [17], and subsequent work by others. The use of a formal attacker model is critical for providing meaningful security guarantees about a protocol in realistic operating environments.

Independently, in recent years, there has been tremendous progress in automatically verifying non-trivial properties of software *implementations*. In this context, one of the most successful techniques is *software model checking* – a combination of predicate abstraction [20] and model checking [14] with automated abstraction refinement [13, 6]. In this paper, we present, to the best of our knowledge, the first method that weds software model checking with a standard protocol security model to provide meaningful security analysis of protocol implementations in a completely automated manner. Our method provides

```
void client(int i,int r)
{
  int s = 0;
  while(1) {
    if(s == 0) {
      send_chall(i,r); ++s;
    } else if(s == 1) {
      recv_resp(i,r); ++s
    } else return;
  }
}
```

```
void server(int r)
{
  int i,s = 0;
  while(1) {
    if(s == 0) {
      recv_chall(r,&i); ++s;
    } else if(s == 1) {
      send_resp(r,i); ++s
    } else return;
  }
}
```

Fig. 1. An implementation of a two-party signature-based challenge-response protocol.

analogous guarantees about protocol implementations as previous work does for protocol specifications.

A key distinguishing feature of our method is that the process of constructing models from implementations involves the use of both predicate abstraction and protocol security models. We observe that, in practice, protocol implementations involve two kinds of operations – non-cryptographic control flow steps and cryptographic computation and communication. Furthermore, the cryptographic operations are delegated to standard library routines. Control flow operations are naturally modeled via predicate abstraction, while cryptographic operations are modeled via actions such as send, receive, encryption and decryption. Our language for protocol actions is based on earlier work on verifying protocol specifications [15]. In addition, our abstraction includes the standard attacker model for protocol analysis. Last, but not the least, we define the security properties of interest—authentication and secrecy–in the context of our modeling formalism. Our modeling technique is described in full detail in Section 2.

As mentioned before, an important aspect of our approach is that the *attacker model* corresponds directly to the attacker model used in analyzing protocol specifications. We assume that the attacker controls the network: it can intercept and modify messages as well as inject messages that it can compute. The attacker's capabilities to compute messages based on previously observed messages is captured by a set of axioms modeling the standard *symbolic attacker* [30, 17] (e.g. if the attacker can compute an encrypted message and the corresponding decryption key, it can recover the plaintext message). We use automated theorem proving techniques to reason about this logical theory of attacker computations. Consequently, our tool supports analysis of protocol implementations with a bounded number of concurrent sessions and unbounded message depth in attacker computations. This is exactly the same model that is used in current state-of-the-art model-checking tools for protocol specifications, such as the AVISPA suite of tools [5]. Further details about our analysis framework are presented in Section 3.

An example of the kind of protocol implementations we want to verify is given in Figure 1. The example implements a two-party encryption-based challenge-response protocol. The `client` function implements the role of a protocol initia-

tor with two parameters: (a) `i` is the identity of the thread executing the client role, and (b) `r` is the identity of server with which the client wishes to communicate. Similarly, the `server` function implements the role of a protocol responder with one parameters: `r` is the identity of the thread executing the server role. The routines invoked implement well-defined cryptographic actions. For example, `send_chall` creates a nonce and sends it encrypted with the server's public key over the network. Similarly, `send_resp` decrypts the message received and sends the result back over the network. We use this as a running example to explain our technique in the rest of this paper.

We have implemented a tool based on our method and applied it successfully to establish authentication and secrecy properties of the *OpenSSL* [31] implementation of the SSL protocol. Our tool takes the following inputs: (a) the protocol code in C, (b) the number of concurrent sessions, (c) the attacker model, and (d) a specification of a security (secrecy or authentication) property φ. The tool has two possible outputs: (i) the protocol code satisfies φ, even in the face of an attack, or (ii) the protocol code violates φ along with a counterexample exhibiting a possible violation of φ. The tool incorporates several optimizations that enable it to scale to industrial strength security protocols. In particular, we were able to verify both authentication and secrecy properties of OpenSSL for configurations comprising of up to 3 servers and 3 clients. Further details about our experimental results are presented in Section 4.

Related Work. In [36], the authors check that implementations of protocols such as SSH conform to rule-based specifications capturing the protocol description in the RFC. However, they do not model the attacker. Bhargavan et al [8] develop a method and tool for establishing meaningful security properties of protocols written in *F#* by translating it into the π-calculus and using an automated tool for verifying the resulting translation. However, their method does not apply directly to a C-style imperative language. In [19], the authors use program analysis techniques to extract an abstract representation of the protocol using Horn clauses; secrecy properties are then cast in terms of a satisfiability problem for a set of Horn clauses. The authors apply their method to a small example. The method as presented there does not extend immediately to trace properties such as authentication. In addition, the use of Horn clauses leads to a loss of precision in abstracting programs involving loops.

2 Security Protocol Model and Properties

In this section, we describe the abstract model for security protocols and the attacker. We also precisely define the security properties of interest—authentication and secrecy.

2.1 Protocol Syntax

Messages. Messages are defined by a free term algebra. We assume the following denumerable and mutually disjoint sets: (a) *Var*: variables, (b) *SId*: session ids,

(c) *Name*: principal names, (d) *Nonce*: nonces (globally unique numbers), and (e) K: symmetric keys. The set *Key* of keys is defined in BNF format as follows:

$$Key = K \mid privkey(Name) \mid pubkey(Name)$$

The set *Term* of terms in our term algebra is defined as follows:

$$Term = Var \mid SId \mid Nonce \mid Key \mid \{Term\}_{Key}$$

$$\mid Sig(privkey(Name), Term) \mid Hash_K(Term) \mid (Term, Term)$$

where $\{t\}_k$ denotes the encryption and decryption of t with k, $Sig(privkey(N), t)$ denotes N's signature over the message t, and $Hash_k(t)$ denotes the keyed hash over message t using key k. For any key $k \in Key$, we use the notation k^{-1} to refer to its (unique) reverse key. Also, (t_1, t_2) denotes a pair of messages t_1 and t_2. A message is a ground term (i.e., a term without any variables). The set of all messages is denoted by *Msg*. We assume that terms are implicitly typed (so that a signature is not confused with a nonce, for example).

Actions. Protocol actions include sending and receiving messages, generating nonces, creating messages, decryption, and pattern matching. The set *Act* of actions is defined as follows:

$$Act = \texttt{new}\ Var \mid \texttt{send}\ Term \mid \texttt{recv}\ Var \mid \texttt{match}\ Var/Term$$

$$\mid Var := Msg \mid Var := \texttt{dec}(Var, Var) \mid Act; Act$$

where: (a) `new` v denotes creating a fresh nonce and storing it in v, (b) `send` t denotes sending a message, (c) `recv` v denotes receiving a message and storing it in v, (d) `match` t_1/t_2 denotes matching the term t_1 with the term t_2, (e) $v := m$ denotes assigning the m to v, (f) $v := \texttt{dec}(v', v'')$ denotes decrypting v' with v'' and storing the result in v, and $\alpha_1; \alpha_2$ denotes α_1 followed by α_2.

Statements. Let *Expr* be a set of expressions defined over *Var* using the standard set of numeric (+, -, * etc.), relational ($<, >, =$ etc.), and boolean ($\wedge, \vee, \neg$ etc.) operators. The set *Stmt* of statements is defined as follows:

$$Stmt = Var := Expr \mid \texttt{assume}\ Expr \mid \texttt{skip}$$

The weakest precondition of an expression e with respect to a statement *St*, denoted by $\mathcal{WP}\{e\}[St]$ is defined as follows: (a) $\mathcal{WP}\{e\}[v := e']$ is obtained by replacing every occurrence of v in e with e', (b) $\mathcal{WP}\{e\}[\texttt{assume}\ e'] = e \wedge e'$, and (c) $\mathcal{WP}\{e\}[\texttt{skip}] = e$.

Context. A term reduces to a message in a specific *context*. Formally, a context $\nu : Var \hookrightarrow Msg$ is a partial mapping from variables to messages. For any context ν, and any term t, we write $\nu[t]$ to mean the message obtained by replacing each variable v in t with $\nu(v)$. If t contains a variable v such that $\nu(v)$ is undefined, then $\nu[t]$ is also undefined (written as $\nu[t] = \perp$). The set of all contexts is denoted

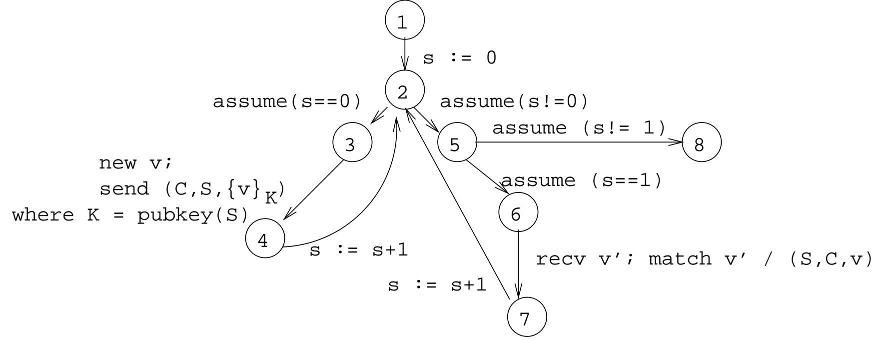

Fig. 2. The role corresponding to `client` in Figure 1.

by *Context*. We write $\{v = m\}$ to denote the singleton context that maps v to m. For any contexts ν_1 and ν_2, we write $\nu_1 \bowtie \nu_2$ to mean the context ν such that:

$$(Dom(\nu) = Dom(\nu_1) \cup Dom(\nu_2)) \bigwedge (\forall v \in Dom(\nu_1) \setminus Dom(\nu_2) \centerdot \nu(v) = \nu_1(v)) \bigwedge$$

$$(\forall v \in Dom(\nu_2) \centerdot \nu(v) = \nu_2(v))$$

Protocol. A *role* is a 4-tuple (S, I, P, T) where: (i) S is a finite set of nodes, (ii) $I \in S$ is an initial node, (iii) $P \subseteq Var$ is a set of input parameters, and (iv) $T \subseteq S \times (Act \cup Stmt) \times S$ is a transition relation where each transition is labeled with an action or a statement. A *protocol* is a set of roles.

A *thread* is an instance of a role executed by a principal. Each thread is identified by a principal name and a unique thread id, and an assignment of values to the role parameters. Formally, a thread is a quadruple (Id, ν, S, I, T) where: (i) $Id = (\eta, N)$ is the thread identifier comprising of a session id η and a name N, (ii) ν is a context, and (iii) $(S, I, Dom(\nu), T)$ is the role being instantiated by the thread.

Example 1. Recall that our example protocol (see Figure 1) consists of two roles: client and server. A thread instantiating the client role is shown pictorially in Figure 2. The nodes of the role are derived from the control flow graph of the `client` procedure, and the transitions are labeled accordingly. The transitions corresponding to cryptographic routines are labeled by actions abstracting the behavior of these routines. The mapping between cryptographic routines and actions is supplied externally. The input parameters `i` and `r` are instantiated with names `C` and `S` respectively. The name `C` appears as part of the thread identifier as well. In summary the thread is (Id, ν, S, I, T) where: (i) $Id = (\eta, \mathtt{C})$, (ii) $\nu = \{i = \mathtt{C}, r = \mathtt{S}\}$, (iii) $S = \{1, 2, \ldots, 8\}$, (iv) $I = 1$, and (v) T is as shown in Figure 2.

2.2 Protocol Semantics

Attacker Model. We use the standard symbolic (Dolev-Yao) attacker model [17, 30]. The attacker capabilities are represented via the following inference rules (where $S \vdash m$ means that the attacker can compute message m from the set S of messages):

$$\begin{array}{rcl} S \vdash m \wedge S \vdash k \implies S \vdash \{m\}_k & \quad & S \vdash \{m\}_k \wedge S \vdash k^{-1} \implies S \vdash m \\ S \vdash m \wedge S \vdash k' \implies S \vdash \mathit{Sig}(k', m) & & S \vdash m \wedge S \vdash k \implies S \vdash \mathit{Hash}_k(m) \\ S \vdash m_1 \wedge S \vdash m_2 \implies S \vdash (m_1, m_2) & & S \vdash (m_1, m_2) \implies S \vdash m_1 \wedge S \vdash m_2 \end{array}$$

In the above, k' is of the form $\mathit{privkey}(N)$ for some name N. In addition, the attacker can generate nonces. The attacker has complete control over the network: it can intercept every message sent on the network and send messages that it can construct (using the above inference rules) to honest parties.

Environment. An *environment* for a thread $\mathcal{T}$ is a pair $(\nu, \Gamma)^{\mathcal{T}}$ such that $\nu \in \mathit{Context}$ and $\Gamma \subseteq \mathit{Msg}$. The set $\mathit{Context} \times 2^{\mathit{Msg}}$ of all environments is denoted by Env. For any environment $E = (\nu, \Gamma)^{\mathcal{T}}$, we write E_ν and E_Γ to mean ν and Γ respectively. Environments model information available to the thread and the attacker during the thread's execution. Specifically, an environment $(\nu, \Gamma)^{\mathcal{T}}$ means that the thread $\mathcal{T}$'s variable binding is ν, while Γ is the set of all messages sent out on the network (and thus available to the attacker). We omit the superscript when the thread $\mathcal{T}$ is clear from the context.

Environment Transformer. Let $\mathit{Decrypt} : \mathit{Msg} \times \mathit{Msg} \hookrightarrow \mathit{Msg}$ be a function such that $\mathit{Decrypt}(m_1, m_2)$ is the result of successful decryption of message m_1 with m_2. If decryption of m_1 with m_2 fails, then $\mathit{Decrypt}(m_1, m_2)$ is undefined (written as $\mathit{Decrypt}(m_1, m_2) = \bot$). The function $\mathit{match} : \mathit{Msg} \times \mathit{Term} \hookrightarrow \mathit{Context}$) takes a message m and a term t and returns a context ν such that $m = \nu[t]$ and the domain of ν is equal to the set of variables in t. If no such context exists, then $\mathit{match}(m, t)$ is undefined and is denoted by $\bot$.

We view an action as an environment transformer. For any action α, the transformer relation $\mathcal{R}_\alpha \subseteq \mathit{Env} \times \mathit{Env}$ is defined as follows:

$$\mathcal{R}_{\texttt{new}\ v} = \{(E, E') \mid E'_\nu = E_\nu \bowtie \{v = n\} \wedge E'_\Gamma = E_\Gamma\} \text{ where } n \text{ is a fresh nonce}$$

$$\mathcal{R}_{\texttt{send}\ t} = \{(E, E') \mid (E_\nu[t] \neq \bot) \wedge (E'_\nu = E_\nu) \wedge (E'_\Gamma = E_\Gamma \cup \{E_\nu[t]\})\}$$

$$\mathcal{R}_{\texttt{recv}\ v} = \{(E, E') \mid E'_\nu = E_\nu \bowtie \{v = m\} \wedge E_\Gamma \vdash m \wedge E'_\Gamma = E_\Gamma\}$$

$$\mathcal{R}_{\texttt{match}\ v/t} = \{(E, E') \mid E'_\nu = E_\nu \bowtie \mathit{match}(E_\nu(v), t) \wedge E'_\Gamma = E_\Gamma\}$$

$$\mathcal{R}_{v:=m} = \{(E, E') \mid E'_\nu = E_\nu \bowtie \{v = m\} \wedge E'_\Gamma = E_\Gamma\}$$

$$\mathcal{R}_{v:=\texttt{dec}(v', v'')} =$$

$$\{(E, E') \mid E'_\nu = E_\nu \bowtie \{v = m\} \wedge m = \mathit{Decrypt}(E_\nu(v'), E_\nu(v'')) \neq \bot \wedge E'_\Gamma = E_\Gamma\}$$

$$\mathcal{R}_{\alpha_1;\alpha_2} = \mathcal{R}_{\alpha_1} \circ \mathcal{R}_{\alpha_2}$$

Predicate Abstraction. A predicate is an expression. Let $\mathcal{P}$ be a set of predicates. A valuation V of $\mathcal{P}$ is a function from $\mathcal{P}$ to $\{\text{TRUE}, \text{FALSE}\}$. The concretization of V, denoted by $\gamma(V)$, is the expression defined as follows:

$$\gamma(V) = \bigwedge_{p \in \mathcal{P}} \gamma_V(p)$$

where $\gamma_V(p) = p$ if $V(p) = \text{TRUE}$ and $\gamma_V(p) = \neg p$ if $V(p) = \text{FALSE}$. The set of all valuations of $\mathcal{P}$ is denoted by $\mathcal{V}_\mathcal{P}$. In predicate abstraction [20], every statement is viewed as a predicate valuation transformer. Specifically, for any statement St and set of predicates $\mathcal{P}$, the transformer relation $\mathcal{R}_{St,\mathcal{P}} \subseteq \mathcal{V}_\mathcal{P} \times \mathcal{V}_\mathcal{P}$ is defined as follows:

$$\mathcal{R}_{St,\mathcal{P}} = \{(V, V') \mid \gamma(V) \wedge \mathcal{WP}\{\gamma(V')\}[St] \text{ is satisfiable}\}$$

Labeled Transition System (LTS). An LTS is a 4-tuple $M = (\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$ such that: (i) $\mathbb{S}$ is a set of states, (ii) $\mathbb{I} \subseteq \mathbb{S}$ is the set of initial states, (iii) Σ is an alphabet of events, and (iv) $\mathbb{T} \subseteq \mathbb{S} \times \Sigma \times \mathbb{S}$ is the transition relation. A state $s \in \mathbb{S}$ is reachable in M iff there exists a sequence $\langle s_1, \alpha_1, \ldots, s_{n-1}, \alpha_{n-1}, s_n\rangle$ such that: (i) $s_1 \in \mathbb{I}$, (ii) $s_n = s$ and (iii) for $1 \le i < n \centerdot (s_i, \alpha_i, s_{i+1}) \in \mathbb{T}$.

We now define a notion of a sequence of events being in order at a state. For any LTS $M = (\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$, the predicate $\mathit{InOrder}_M \subseteq \mathbb{S} \times \Sigma^*$ is defined recursively as follows: (i) $\forall s \in \mathbb{S} \centerdot \mathit{InOrder}_M(s, \langle\rangle)$, and (ii) for any state s and any $t = \langle \alpha_1, \ldots \alpha_n\rangle \in \Sigma^*$, $\mathit{InOrder}_M(s, t)$ iff the following holds for every $s' \in \mathbb{S}$ and $\alpha \in \Sigma$ such that $(s', \alpha, s) \in \mathbb{T}$:

$$(\alpha = \alpha_n \wedge \mathit{InOrder}_M(s', \langle \alpha_1, \ldots \alpha_{n-1}\rangle)) \bigvee (\alpha \notin \{\alpha_1, \ldots, \alpha_n\} \wedge \mathit{InOrder}_M(s', t))$$

Thread Model. Let $\mathcal{P}$ be a set of predicates and $\mathcal{T} = (Id, \nu, S, I, T)$ be a thread. Then the model of $\mathcal{T}$ over $\mathcal{P}$ is the LTS $M(\mathcal{T}, \mathcal{P}) = (\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$ such that (i) $\mathbb{S} = S \times \mathcal{V}_\mathcal{P} \times \mathit{Env}$, (ii) $\mathbb{I} = \{I\} \times \mathcal{V}_\mathcal{P} \times \{(\nu, \mathcal{I}nitA)\}$ with $\mathcal{I}nitA$ denoting the attacker's initial knowledge, (iii) $\Sigma = (\mathit{Act} \cup \mathit{Stmt}) \times \{Id\}$, and (iv) $\mathbb{T}$ is defined as follows:

$$\mathbb{T} = \{((s, V, E), (\alpha, Id), (s', V, E')) \mid (s, \alpha, s') \in T \wedge (E, E') \in \mathcal{R}_\alpha\}$$
$$\bigcup \{((s, V, E), (St, Id), (s', V', E)) \mid (s, St, s') \in T \wedge (V, V') \in \mathcal{R}_{St,\mathcal{P}}\}$$

Example 2. Recall, from Figure 2, our example client thread $\mathcal{T} = (Id, \nu, S, I, T)$ where: (i) $Id = (\eta, \mathtt{C})$, (ii) $\nu = \{i = \mathtt{C}, r = \mathtt{S}\}$, (iii) $S = \{1, 2, \ldots, 8\}$, (iv) $I = 1$, and (v) T is as shown in Figure 2. Let $\mathcal{P} = \{p_0, p_1\}$ be a set of predicates such that $p_0 \equiv$ `(s == 0)` and $p_1 \equiv$ `(s == 1)`. Then the thread model $M(\mathcal{T}, \mathcal{P})$ is the LTS some of whose important transitions are:

$$(1, \{\neg p_0, \neg p_1\}, (\nu, \mathcal{I}nitA)) \quad \xrightarrow{(s:=0, Id)} \quad (2, \{p_0, \neg p_1\}, (\nu, \mathcal{I}nitA))$$

$$(2, \{p_0, \neg p_1\}, (\nu, \mathcal{I}nitA)) \quad \xrightarrow{(\mathtt{assume}(s==0), Id)} \quad (3, \{p_0, \neg p_1\}, (\nu, \mathcal{I}nitA))$$

$$(3, \{p_0, \neg p_1\}, (\nu, \mathcal{I}nitA)) \quad \xrightarrow{(\alpha_1, Id)} \quad (4, \{p_0, \neg p_1\}, (\nu \bowtie \{v = n\}, \mathcal{I}nitA \cup \{(\mathtt{C}, \mathtt{S}, \{n\}_k)\}))$$

In the above, n is a nonce, $\alpha_1 = \mathtt{new}\ v; \mathtt{send}(\mathtt{C}, \mathtt{S}, \{v\}_k)$ and $k = \mathit{pubkey}(\mathtt{S})$.

Thread Composition. We assume that threads execute asynchronously and have disjoint sets of variables. We now present the model of the composition of threads. We use two threads for simplicity. Our model generalizes naturally to an arbitrary but finite number of threads. Let $\mathcal{T}_1 = (Id_1, \nu_1, S_1, I_1, T_1)$ and $\mathcal{T}_2 = (Id_2, \nu_2, S_2, I_2, T_2)$ be two threads and let $M(\mathcal{T}_1, \mathcal{P}_1) = (\mathbb{S}_1, \mathbb{I}_1, \Sigma_1, \mathbb{T}_1)$ and $M(\mathcal{T}_2, \mathcal{P}_2) = (\mathbb{S}_2, \mathbb{I}_2, \Sigma_2, \mathbb{T}_2)$ be their models over two sets of predicates $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively. Then the composed model of the two threads $M(\mathcal{T}_1, \mathcal{P}_1) \parallel M(\mathcal{T}_2, \mathcal{P}_2)$ is the LTS $(\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$ such that: (i) $\mathbb{S} = S_1 \times \mathcal{V}_{\mathcal{P}_1} \times S_2 \times \mathcal{V}_{\mathcal{P}_2} \times Env$, (ii) $\mathbb{I} = \{I_1\} \times \mathcal{V}_{\mathcal{P}_1} \times \{I_2\} \times \mathcal{V}_{\mathcal{P}_2} \times \{(\nu_1 \bowtie \nu_2, \mathcal{I}nitA)\}$, (iii) $\Sigma = \Sigma_1 \cup \Sigma_2$, and (iv) $\mathbb{T}$ is defined as follows:

$$\mathbb{T} = \{((s_1, V_1, s_2, V_2, E), X, (s_1', V_1', s_2', V_2', E'))\}$$

such that for $i \in \{1, 2\}$, the following holds: if $X \in \Sigma_i$ then $((s_i, V_i, E), X, (s_i', V_i', E')) \in \mathbb{T}_i$, otherwise $(s_i = s_i') \wedge (V_i = V_i')$.

We note that this model is similar to the abstract model used for protocol analysis. The presentation of the model is different because we want to align our model with those obtained by predicate abstraction for software model checking. Section 3 describes concretely how we construct the abstract model starting from C code.

2.3 Security Properties and Satisfaction

We deal with two types of security properties: authentication and secrecy. An *authentication* property φ is specified by a finite sequence of events $\langle \alpha_1, \ldots, \alpha_n \rangle$. A model $M = (\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$ satisfies φ iff whenever there exists two reachable states s and s' of M such that $(s', \alpha_n, s) \in \mathbb{T}$, $InOrder_M(s, \langle \alpha_1, \ldots, \alpha_n \rangle)$ holds. This formalization of authentication is based on the concept of *matching conversations* [7].

In our example, a possible authentication property is specified by the event sequence $\langle (\alpha_1, Id_1), (\alpha_2, Id_2), (\alpha_3, Id_1), (\alpha_4, Id_2) \rangle$ where Id_1 is the identifier of a thread executing the client role, Id_2 is the identifier of a thread executing the server role, and α_1, α_2, α_3 and α_4 are actions abstracting the library routines `send_chall`, `recv_chall`, `send_resp` and `recv_resp`.

A *secrecy* property ψ is specified by the inability of the attacker to compute a message m. Thus, a model violates ψ iff there exists a reachable state (s_1, V_1, s_2, V_2, E) of M such that $E_\Gamma \vdash m$. Note that both authentication and secrecy as formulated here are *safety* properties.

Property Satisfaction. The composition of threads $\mathcal{T}_1, \ldots, \mathcal{T}_n$ satisfies a property φ iff there exists sets of predicates $\mathcal{P}_1, \ldots, \mathcal{P}_n$ such that $M(\mathcal{T}_1, \mathcal{P}_1) \parallel \cdots \parallel M(\mathcal{T}_n, \mathcal{P}_n)$ satisfies φ. Let $\mathcal{Q}$ be a protocol with n roles. A *configuration* $\mathcal{C}$ of $\mathcal{Q}$ is a function from $\{1, \ldots, n\}$ to integers. Intuitively, $\mathcal{C}(i)$ is the number of threads instantiating the ith role. Then $\mathcal{Q}$ satisfies a property φ under $\mathcal{C}$ iff every composition (obtained by instantiating the input parameters) of $\sum_i \mathcal{C}(i)$ threads (with the first $\mathcal{C}(1)$ threads instantiating the first role, the second $\mathcal{C}(2)$ threads instantiating the second role, and so on) satisfies φ.

3 Protocol Analysis

In this section, we describe our framework for verifying security properties of network protocol implementations. The input to our analysis engine consists of: (i) the source code for the implementation of a protocol $\mathcal{Q}$, (ii) the authentication or secrecy property φ to be verified, (iii) a configuration $\mathcal{C}$ of $\mathcal{Q}$, and (iv) a mapping κ from cryptographic libraries to actions. The output of our analysis is either TRUE, indicating that the property holds for the specified number of protocol threads, or FALSE, indicating the existence of an attack. In the latter case, our analysis also emits a *counterexample* trace that exhibits a possible attack.

Our analysis considers each possible instantiation of the input parameters of the roles separately. For each such instantiation, it follows the counterexample guided abstraction refinement paradigm and proceeds as follows:

1. **Abstraction:** Create an abstract model M as defined in Section 2 for the specified number of threads using predicate abstraction and abstractions of cryptographic libraries. Proceed to Step 2.
2. **Verification:** Use model checking to verify that M satisfies φ. If model checking succeeds, we return TRUE. If model checking fails, we obtain a counterexample CE and proceed to Step 3.
3. **Validation:** Check whether CE is a real counterexample, i.e., it exhibits a valid attack. If so, return FALSE along with CE. Otherwise, CE is spurious. Proceed to Step 4.
4. **Refinement:** Use CE to update the set of predicates. Repeat from Step 1.

The focus of this paper is the abstraction step that lets us obtain abstract protocol models from source code. In future work, we plan to investigate specialized methods for the other steps as well. In the rest of this section, we present our abstraction methodology in more detail.

The threads are obtained from the source code by constructing the control flow graph and instantiating the input parameters of the different roles. The abstract model is then computed in accordance with the definitions specified in Section 2. Automated theorem provers are used to compute the two kinds of transformers involved in the abstract model: (a) predicate valuation transformers, and (b) environment transformers. While step (a) is carried out using standard predicate abstraction techniques, step (b) uses protocol-specific reasoning involving the theory of attacker message derivations. We have implemented and experimented with two ways to achieve this. First, we formalize the attacker model (presented earlier) as a logical theory and then reduce the problem of deciding whether a message can be derived by the attacker to a validity problem. The second approach involves a decision procedure embodying the idea that the message derivation problem can be solved by a process of deconstruction followed by construction [33].

Abstraction-related Issues. Our approach assumes that programs consist logically of two types of independent operations – control flow and cryptographic computation and communication. We use predicate abstraction to model the control flow in a finite manner, while we use security concepts (actions, attacker model etc.) to model the cryptographic operations. We assume that the control-flow affects the cryptographic operations only by controlling which actions are invoked, and has no bearing on the actual data sent, received and manipulated by the cryptographic operations. Similarly, a cryptographic operation only influences whether control flows or not (for example an action involving a pattern-matching "blocks" if the match fails), and not the direction of control flow. Thus, our approach is unable to detect attacks that rely on cryptographic operations affecting program control flow (such as key values affecting the number of times a loop iterates [9], and hence the running time of a thread execution).

The soundness of our abstract model relies on three observations (assumptions): (a) control flow is soundly abstracted by predicate abstraction, a sound and well-understood technique, (b) cryptographic operations are precisely captured by our model definition (see Section 2) using standard security and cryptographic concepts, and (c) the control flow and cryptographic operations affect each other only in a very restricted manner as described in the previous paragraph.

In addition, we handle restricted C programs involving conditionals and assignments over integer variables. Detecting low-level security issues, such as buffer overflow, require orthogonal verification methods [37]. More generally, we assume that the control flow graph accurately represents the possible executions of the program. Some preliminary work on discharging this assumption using other techniques are being investigated [2].

Simulation conformance of OpenSSL code to the SSL RFC specification in the *absence* of an attacker has been verified previously [12]. The key contribution of the present research is the inclusion of a formal and explicit attacker in the abstract model extracted from the code, leading to a significantly more meaningful security analysis.

4 Experimental Results

We implemented our approach on top of the COPPER tool and experimented with the C source code of OpenSSL version 0.9.6c that implements the initial handshake protocol between a server and a client. For each configuration, we verified the following three security properties: (i) **AuthSrvr** – the protocol ensures that every server is always correctly authenticated to a client, (ii) **AuthClnt** – the protocol ensures that every client is always correctly authenticated to a server, and (iii) **Secrecy** – the protocol ensures that a client's secret can never be derived by the attacker. All our experiments were carried out on a 2.4 GHz machine with 4 GB of RAM.

Due to symmetry, it sufficed to verify **AuthSrvr** only for the first server, and **AuthClnt** and **Secrecy** only for the first client. Also, each property was verified independently for every possible instantiation of the input parameters of

	AuthSrvr			AuthClnt			Secrecy		
	Inst	**Time**	**Mem**	**Inst**	**Time**	**Mem**	**Inst**	**Time**	**Mem**
DeconsCons	15	14090	182	15	30797	273	15	33274	273
Simplify	15	19700	181	15	42854	272	15	45732	272

Fig. 3. Comparison between two types of attacker message derivations. **DeconsCons** = deconstruction-construction; **Simplify** = using Simplify; **Inst** = no. of input parameter instantiations verified; **Time** = time in seconds; **Mem** = memory in MB.

C#,S#	AuthSrvr			AuthClnt			Secrecy		
	Inst	**Time**	**Mem**	**Inst**	**Time**	**Mem**	**Inst**	**Time**	**Mem**
2,2	15	14090	182	15	30797	273	15	33274	273
3,3	150	322110	624	139*	334063	635	135*	313912	634
4,4	10*	292459	887	-	-	-	-	-	-
5,5	2*	257945	812	-	-	-	-	-	-

Fig. 4. Experimental results for different client and server configurations. **C#** = no. of clients; **S#** = no. of servers; a * indicates that not all possible instantiations were verified due to lack of time; a - indicates that no experiments were done.

the client roles. Since these parameters represented principal names, they needed to be assigned a finite number of different values. Hence the total number of possible instantiation of the input parameters was also finite.

First, we compared between the use of a general purpose theorem prover (Simplify) and the decision procedure based on deconstruction followed by construction to compute possible message derivations by the attacker. Our results, for a 2-client-2-server configuration, are summarized in Figure 3. They indicate the algorithm based on deconstruction and construction is about 40% faster than using Simplify.

Next, we varied the number of server and client threads and verified the three properties for each configuration. Figure 4 summarizes our results. Due to lack of time not all possible parameter instantiations were verified for configurations with more than 3 servers and 3 clients. However, memory requirements are quite modest, indicating that our strategy of checking instantiations independently is a sound space-time tradeoff. We believe that complete verification of large configurations is possible on clusters or distributed computers with multi-core CPUs.

5 Conclusion

In this paper, we presented a novel method that combines software model checking with a standard protocol execution and attacker model to provide meaningful security analysis of protocol implementations in a completely automated manner. We have implemented our approach and verified authentication and

secrecy properties of OpenSSL for configurations consisting of up to 3 servers and 3 clients. We have also implemented two distinct methods for reasoning about attacker message derivations (based on general purpose theorem prover and deconstruction followed by construction) and present their comparison in the context of OpenSSL verification. While we focus on security protocol implementations in this paper, we believe that our approach is quite general and applies to other secure software systems as well.

References

1. IEEE Standard 802.11-1999. Local and metropolitan area networks - specific requirements - part 11: Wireless LAN Medium Access Control and Physical Layer specifications., 2004.
2. M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, pages 340–353, 2005.
3. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.
4. M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.
5. A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. 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*, volume 3576 of *LNCS*, pages 281–285, 2005.
6. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, Toronto, Canada, May 19–20, 2001. New York, NY, May 2001. Springer-Verlag.
7. M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryprtology - Crypto '93 Proceedings*, pages 232–249, 1994.
8. K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW*, pages 139–152, 2006.
9. D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.
10. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
11. I. Cervesato, A. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key kerberos. In *Proceedings of the 11-th Annual Asian Computing Science Conference*, 2006.
12. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.
13. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. CAV '00*, volume 1855 of *LNCS*, pages 154–169, July 2000.
14. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.

15. A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (PCL). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.
16. T. Dierks and C. Allen. The TLS protocol version 1.0, 1999. RFC 2246.
17. D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.
18. A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol version 3.0, 1996. Internet Draft.
19. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI*, pages 363–379, 2005.
20. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proc. CAV '97*, volume 1254 of *LNCS*, pages 72–83, June 1997.
21. C. He and J. C. Mitchell. Security analysis and improvements for IEEE 802.11i. In *Proc. NDSS*, 2005.
22. S. Kent and R. Atkinson. Security architecture for the internet protocol, 1998. RFC 2401.
23. J. Kohl and B. Neuman. The Kerberos network authentication service (version 5). IETF RFC 1510, September 1993.
24. P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security protocols. In *Formal Methods World Congress, vol. I*, number 1708 in LNCS, pages 776–793, 1999.
25. G. Lowe. Some new attacks upon security protocols. In *Proc. CSFW*, pages 162–169. IEEE, 1996.
26. C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
27. C. Meadows. Analysis of the Internet Key Exchange protocol using the NRL protocol analyzer. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 1998.
28. C. Meadows and D. Pavlovic. Deriving, attacking and defending the GDOI protocol. In *Proc. ESORICS 2004*, volume 3193 of *LNCS*, pages 53–72, 2004.
29. J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur-phi. In *IEEE Symposium on Security and Privacy*, pages 141–151, 1997.
30. R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
31. OpenSSL website. `http://www.openssl.org`.
32. L. Paulson. Proving properties of security protocols by induction. In *Proc. CSFW*, pages 70–83, 1997.
33. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is np-complete. In *CSFW*, pages 174–, 2001.
34. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols*. Addison-Wesley, 2001.
35. D. Song. Athena: a new efficient automatic checker for security protocol analysis. In *Proc. CSFW*, pages 192–202. IEEE, 1999.
36. O. Udrea, C. Lumezanu, and J. S. Foster. Rule-based static analysis of network protocol implementations. In *Usenix Security*, pages 193–208, 2006.
37. D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. NDSS '00*, 2000.