

ASPIER: An Automated Framework for Verifying Security Protocol Implementations^{*}

Sagar Chaki Anupam Datta
chaki@sei.cmu.edu danupam@cmu.edu

Carnegie Mellon University

Abstract. We present ASPIER – the first framework that combines software model checking with a standard protocol security model to analyze authentication and secrecy properties of protocol implementations in an automated manner. ASPIER incorporates a standard *symbolic attacker model* and provides analogous guarantees about protocol implementations as previous work does for protocol specifications. We have implemented ASPIER and used it to verify authentication and secrecy properties of a part of an industrial strength protocol implementation – the handshake in OpenSSL 0.9.6c – 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 evaluated them in the context of OpenSSL verification. Finally, ASPIER detected the “version-rollback” vulnerability in OpenSSL 0.9.6c source code.

1 Introduction

Network protocols such as SSL [21], TLS [19], Kerberos [30], IPSec [29], and IEEE 802.11i [1] are designed to enable secure communication over untrusted networks. However, they are notoriously difficult to get right; the literature is replete with serious security flaws uncovered in protocols many years after they were first published [32, 34, 26, 35, 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* [2, 4, 31, 33, 45, 42, 32, 38, 44, 18]. Independently, in recent years, there has been significant 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 [25] and model checking [16] with automated abstraction refinement [15, 5].

^{*} This research was partially supported by the PACC Initiative at the Software Engineering Institute, Pittsburgh, USA, the NSF Cybertrust grant “Realizing Verifiable Security Properties on Untrusted Computing Platforms”, the NSF Science and Technology Center TRUST and the U.S. Army Research Office contract titled “Perpetually Available and Secure Information Systems” (DAAD19-02-1-0389) to CMU’s CyLab.

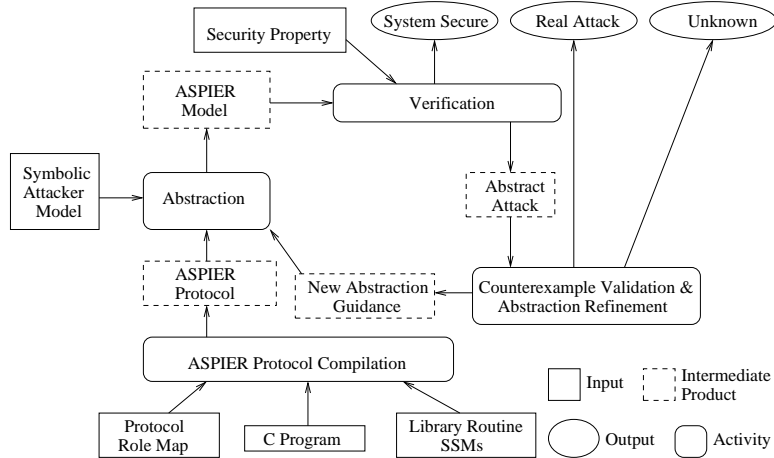


Fig. 1. Overall security verification framework. SSMs = specification state machines.

In this paper, we present ASPIER¹—the first framework that weds software model checking with a standard protocol security model to analyze *authentication* and *secrecy* properties of protocol implementations in an automated manner. We have implemented the method in the ASPIER tool and applied it successfully to establish authentication and secrecy properties of the OpenSSL [41] implementation of the SSL handshake protocol. We elaborate below on the three central contributions of this paper – the verification method, the tool, and the application to OpenSSL – and our underlying assumptions.

Method. The overall ASPIER framework is depicted in Figure 1. A protocol is defined by a set of programs, e.g. OpenSSL consists of a client program and a server program.

ASPIER PROTOCOL COMPILER. We begin with the C programs for the protocol and translate them into a program in the ASPIER protocol language (presented in Section 3). This process involves two main steps. First, calls to library routines are replaced by their corresponding specification state machines (SSMs) via *specification inlining*. The SSMs used for specification inlining are provided as input to ASPIER. For example, a call to an encryption function is replaced by a SSM performing the corresponding encryption action in the ASPIER programming language. Second, a control flow graph (CFG) is extracted by an automated syntactic analysis of each C program. The edges of the CFG are labeled by appropriate statements from the ASPIER programming language.

The design of the ASPIER protocol language is a key contribution of this paper, and is driven by the observation that, in practice, protocol implementations

¹ ASPIER is an acronym for “Automated Security Protocol Implementation verIER”

involve two kinds of operations – numeric and cryptographic. The ASPIER protocol language is designed carefully to separate these two aspects while allowing interaction between these two kinds of operations. Specifically, there are two disjoint sets of variables: message variables and numeric variables. Cryptographic actions, such as generating new random numbers, sending and receiving messages, pattern matching, decryption and signature verification operate on terms (including message variables) of a free term algebra, as in security protocol specifications. Numeric operations, including assignments to numeric variables, and conditionals where the value of a numeric expression determines which branch is taken, are just like in C.

Since numeric and cryptographic statements appear as labels on CFG edges, we are able to model interaction between them. This interaction is crucial for representing realistic protocols (e.g., OpenSSL), where different program statements are executed by an agent based on the value (e.g., version number received) of a message variable. However, assignments of numeric expressions to message variables (and vice-versa) are disallowed. We believe that this is a reasonable restriction since messages are constructed by calls to library routines for encryption, signature generation etc. These design choices reflect tradeoffs among expressivity of the language, and tractability and soundness of ASPIER.

CEGAR IN ASPIER. Once the C programs are translated into ASPIER’s programming language, our verification method follows the counterexample guided abstraction refinement (CEGAR) paradigm [15, 5]. Starting with an initial set of predicates, we proceed iteratively as follows:

1. **Abstraction:** This step automatically extracts an abstract model M of the concrete protocol via ASPIER’s predicate abstraction. We address several key challenges in this step. First, we develop a uniform model that not only combines standard predicate abstraction for C-style code with security protocol actions, but also incorporates a *symbolic attacker model*. In particular, 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* [40, 20] (e.g. if the attacker can compute an encrypted message and the corresponding decryption key, it can recover the plaintext message). Our attacker model is the same as the one used in current state-of-the-art model-checking tools for protocol specifications, such as OFMC [7]. Second, we use automated deduction techniques to reason about the theory of attacker computations induced by our attacker model. Consequently, ASPIER enables automated analysis of protocol implementations with a bounded number of concurrent sessions and unbounded message depth in attacker computations. Finally, we express the security properties of our interest—authentication and secrecy—as reachability properties in the context of our modeling formalism. Our main technical result (Theorem 1) is that ASPIER’s abstraction is sound.

Specifically, for a security property φ , if M satisfies φ , then the concrete system also satisfies φ . Section 4 explains this step in detail.

2. **Verification:** We use reachability analysis to verify that $M \models \varphi$. ASPIER’s verification covers all possible *connection topologies*, i.e., functions mapping each client session C to the server with which C initiates a connection. To improve scalability, ASPIER verifies each possible connection topology separately, thereby trading off time for space. In addition, we use symmetry reduction to reduce the number of connection topologies to be verified, without losing coverage. Section 6 has more details on these optimizations. If verification succeeds, since M is a sound abstraction, we exit with “System Secure”. Otherwise, we obtain a counterexample CE and proceed to Step 3.
3. **Counterexample Validation:** We check whether CE is a real counterexample, i.e., it concretizes to a real attack. If so, we exit with “Real Attack” and a concretization of CE . If we find that CE does not concretize to any real attack, i.e., CE is spurious, we proceed to Step 4. Otherwise, we are unable to determine whether CE concretizes to a real attack or not, and we exit with “Unknown”. ASPIER’s counterexample validation requires an extension to the standard CEGAR approach to account for protocol actions in addition to standard program statements. The technical difference shows up in the definition of weakest preconditions, where protocol actions are treated like NOPs. This is because protocol actions do not directly affect the values of numeric variables in our model. Section 5.1 explains this step in detail.
4. **Refinement:** We use the spurious CE to update the abstraction information and repeat from Step 1. The main property ensured by the refinement process is that CE does not arise as a counterexample in the refined model. This step is standard from previous work. Section 5.2 explains this step in detail.

Tool. We have implemented the ASPIER tool, which accepts the following inputs: (a) the protocol code in C, (b) protocol-specific information (the number of concurrent sessions and the mapping of protocol roles to functions in the C code), (c) the attacker model, (d) library routine SSMs, and (e) a specification of a security (secrecy or authentication) property φ . The tool has three possible outputs: (i) “System Secure”: the protocol code satisfies φ , even in the face of an attack, (ii) “Real Attack”: the protocol code violates φ along with a counterexample exhibiting a possible violation of φ , or (iii) “Unknown”: neither of the previous two conclusions could be derived. The ASPIER implementation is presented in Section 6.

Application. We have used ASPIER to establish authentication and secrecy properties of the *OpenSSL* [41] implementation of the SSL handshake protocol. Our experimental results fall in two categories. First, we consider the situation when clients and servers implement only SSL 3.0. In this scenario, the ASPIER tool verifies important authentication and secrecy properties of OpenSSL for configurations comprising of up to 3 servers and 3 clients. Second, we consider the case where clients and servers implement both SSL 2.0 and SSL 3.0 and negotiate the version during handshake. In this case, the tool detects the “version rollback”

attack whereby an intruder can force a server and a client to use SSL 2.0 even when both intend to use SSL 3.0. To our knowledge, our results are the first of its kind concerning OpenSSL. Further details about our OpenSSL case study and our experimental results are presented in Section 2 and Section 7 respectively.

Assumptions. The current ASPIER implementation treats floating point data as integers, and bit-wise operations as uninterpreted functions. Pointers are also not treated soundly. These limitations arise from COPPER, the underlying model checker. Some of them (e.g., treatment of pointers) are addressable by porting ASPIER to a different model checking engine (e.g., SLAM [5] or BLAST). Other limitations (e.g., treatment of floating point) are still being researched actively. We also assume that the control flow graph accurately represents the possible executions of the program. We believe that this assumption can be discharged using orthogonal techniques (see, for example, [3]). Detecting low-level security issues, such as buffer overflows, also require other program analysis methods [47]. It is noteworthy that, despite these restrictions, we found ASPIER to be useful in reasoning about an industrial strength protocol implementation.

2 Case Study: SSL/OpenSSL Handshake

Secure Socket Layer (SSL) is the most widely used protocol for secure communication over the Internet. OpenSSL is one of the most popular implementations of SSL. A key component of SSL/OpenSSL is the initial handshake that enables a client to connect securely with a server. In the rest of this paper, we write SSL (OpenSSL) to mean the handshake component of SSL 3.0 (OpenSSL 0.9.6c). Security analysis of the SSL specification has been performed with considerable success [37]. In this paper, we use ASPIER to prove important authentication and secrecy properties of the OpenSSL implementation.

Broadly, an SSL handshake between a client (initiator) and a server (responder) consists of the following sequence of four message exchanges: (1) `client_hello` from client to server, (2) `server_hello` from server to client, (3) `client_verify` from client to server, and (4) `server_finished` from server to client. Figures 2(a) and 2(b) show *extremely simplified* versions of the OpenSSL client and server code respectively. The `client` function implements the role of a protocol initiator with three parameters: (a) `i` – identity of thread executing client role, (b) `r` – identity server with which client wishes to communicate, and (c) `v` – version of SSL client wishes to use. Similarly, the `server` function implements the role of a protocol responder with two parameters: (a) `r` – identity of the thread executing the server role, and (b) `v` – version of SSL server wishes to use. A program running `client` (or `server`) essentially executes a finite state machine with four states. In each state, the program sends or receives an appropriate message via a library routine call. For example, `s_c_hello` sends a `client_hello` message to the server, while `r_c_hello` receives a `client_hello` message from a client. The sequence of messages sent and received corresponds to the SSL specification.

```

void client(int i,r,v)
{
  int s = 0;
  while(1) {
    if(s == 0) {
      s_c_hello(...);
    } else if(s == 1) {
      v = r_s_hello(...);
    } else if(s == 2) {
      s_c_verify(...);
    } else if(s == 3) {
      r_s_finished(...);
    } else return;
    ++s;
  }
}
(a)

```

```

void server(int r,v)
{
  int i,s = 0;
  while(1) {
    if(s == 0) {
      v = r_c_hello(...);
    } else if(s == 1) {
      s_s_hello(...);
    } else if(s == 2) {
      r_c_verify(...);
    } else if(s == 3) {
      s_s_finished(...);
    } else return;
    ++s;
  }
}
(b)

```

Fig. 2. An implementation of a two-party signature-based challenge-response protocol.

We use the program in Figures 2 as a running example to illustrate our approach. Specifically, we show how ASPIER is able to verify the following three security properties of this program: (i) *Server Authentication*: A `server_finished` message sent by a server is always preceded by an appropriate sequence of messages; (ii) *Client Authentication*: A `server_finished` message received by a client is always preceded by an appropriate sequence of messages; and (iii) *Secrecy*: The secret transmitted by a client (as part of its `client_verify` message) is only revealed to the server. We note that even though we use a simplified version of OpenSSL to illustrate our approach, our experiments were performed on benchmarks derived from actual OpenSSL source code.

3 Concrete Security Protocol Model and Properties

In this section, we describe the syntax and semantics of the ASPIER protocol language, including our attacker model. We also precisely define the security properties of our interest—authentication and secrecy.

3.1 Protocol Language Syntax

We assume the following denumerable and mutually disjoint sets: (a) $MConst, MVar$: message constants and variables, (b) $\mathbb{Z}, NVar$: numeric (integer) constants and variables, (c) Sid : session ids, (d) $Name$: principal names, (e) $Nonce$: nonces (globally unique numbers), and (f) K : symmetric keys.

Messages. Messages are defined by a free term algebra. The set Key of keys is defined in BNF format as follows:

$$Key = K \mid \text{privkey}(Name) \mid \text{pubkey}(Name)$$

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

$$Term = MConst \mid MVar \mid Sid \mid Nonce \mid Key \mid \{Term\}_{Key} \\ \mid Sig(privkey(Name), Term) \mid Hash_K(Term) \mid (Term, Term)$$

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 to avoid confusion, e.g., between a signature and a nonce.

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:

$$Act = \mathbf{new} \ MVar \mid \mathbf{send} \ Term \mid \mathbf{recv} \ MVar \mid MVar := Msg \\ \mid \mathbf{match} \ MVar / Term$$

where: (a) $\mathbf{new} \ v$ denotes creating a fresh nonce and storing it in v , (b) $\mathbf{send} \ t$ denotes sending a message, (c) $\mathbf{recv} \ v$ denotes receiving a message and storing it in v , (d) $v := m$ denotes assigning m to v , and (e) $\mathbf{match} \ v/t$ denotes matching the value of v with t . Note that decryption and signature verification are implemented via pattern matching.

Statements. Let $Expr$ be a set of expressions defined over \mathbb{Z} and $NVar$ 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:

$$Stmt = Act \mid NVar := Expr \mid \mathbf{assume} \ Expr$$

Context. A *context* represents an assignment of messages to message variables. Formally, a context $\nu : MVar \hookrightarrow Msg$ is a partial mapping from message variables to messages. The set of all contexts is denoted by \mathcal{C} . We write $\{v = m\}$ to denote the singleton context that maps v to m . For any contexts ν_1 and ν_2 with domains D_1 and D_2 respectively, we write $\nu_1 \bowtie \nu_2$ to mean the context ν such that the following hold:

$$Dom(\nu) = D_1 \cup D_2 \bigwedge \forall v \in D_1 \setminus D_2 \cdot \nu(v) = \nu_1(v) \bigwedge \forall v \in D_2 \cdot \nu(v) = \nu_2(v)$$

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 $v \notin Dom(\nu)$, then $\nu[t]$ is undefined (written as $\nu[t] = \perp$).

Protocol. A *role* is a 4-tuple (S, I, P, T) where: (i) S is a finite set of control-flow-graph (CFG) nodes, (ii) $I \in S$ is an initial CFG node, (iii) $P \subseteq MVar$ is a

set of input parameters, and (iv) $T \subseteq S \times Stmt \times S$ is a transition relation where each transition is labeled with 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 5-tuple (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.

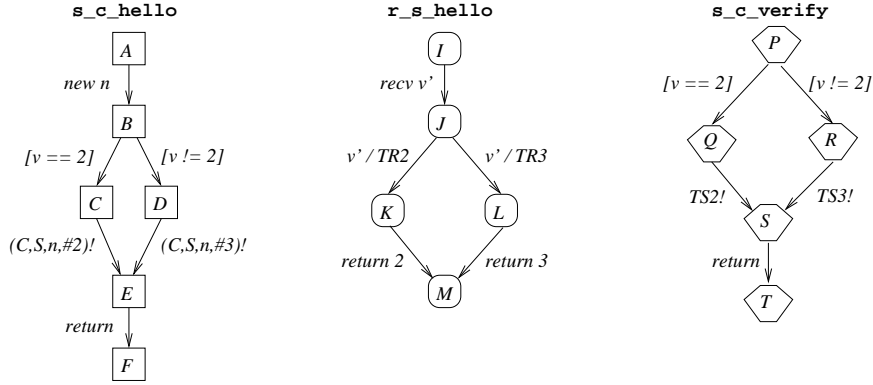


Fig. 3. SSMs for some library routines called by `client` procedure from Fig. 2.

Example 1. Recall, from Section 2, that our running example protocol consists of two roles: client and server. The control flow graph (CFG) of the client role is derived from the `client` procedure shown in Figure 2 via *specification inlining* followed by CFG extraction. Figure 3 shows the SSMs for some of the library routines called by `client`. Figure 4 shows a fragment of the resulting client role CFG. To illustrate specification inlining, each client role CFG node has the same shape and name as its corresponding SSM state.

Figure 4 uses the following convention: (i) $[e] \equiv \mathbf{assume} \ e$; (ii) $t! \equiv \mathbf{send} \ t$; (iii) $\#n \equiv$ message constant n ; (iv) $v/t \equiv \mathbf{match} \ v/t$; (v) $\{t\}t' \equiv \{t\}v$; (vi) $(t_1, \dots, t_n) \equiv (t_1, (\dots(t_{n-1}, t_n) \dots))$; (vii) S is the server's name; (viii) C is the client's name; (ix) CA is the certifying authority's name; (x) n, n' are nonces; (xi) Sec is the client's secret; (xii) $TR2 \equiv (S, C, n', \#2, \mathit{Sig}(\mathit{privkey}(CA), (S, \mathit{pubkey}(S))))$; (xiii) $TR3 \equiv (S, C, n', \#3, \mathit{Sig}(\mathit{privkey}(CA), (S, \mathit{pubkey}(S))))$; (xiv) $TS2 \equiv (C, S, \mathit{Sig}(\mathit{privkey}(CA), (C, \mathit{pubkey}(C))), \mathit{Sig}(\mathit{privkey}(C), HS_2), \{Sec\}_{\mathit{pubkey}(S)}, \mathit{Hash}_{Sec}(HS_2))$ where $HS_2 \equiv$ all previous messages sent and received by the client **excluding** the version number; (xv) $TS3 \equiv (C, S, \mathit{Sig}(\mathit{privkey}(CA), (C, \mathit{pubkey}(C))), \mathit{Sig}(\mathit{privkey}(C), HS_3), \{Sec\}_{\mathit{pubkey}(S)},$

$Hash_{Sec}(HS_3)$) where $HS_3 \equiv$ all previous messages sent and received by the client **including** the version number.

Note the interaction between messages and numeric data in the client. First, the constant (#2 or #3) in the message sent by `s_c_hello` depends on the value of the numeric variable v denoting the version of SSL that the client wishes to use. Next the value (2 or 3) of v is set depending on the constant (#2 or #3) in the message received by `r_s_hello`. Finally, the contents of the message sent by `s_c_verify` depends on the new value of v . The key difference between SSL 2.0 and 3.0 in terms of the message sent by `s_c_verify` is that former does not include the version numbers in encrypted form, while the latter does. This leads to the “version-rollback” attack in SSL 2.0, as we discuss later in Section 7.1.

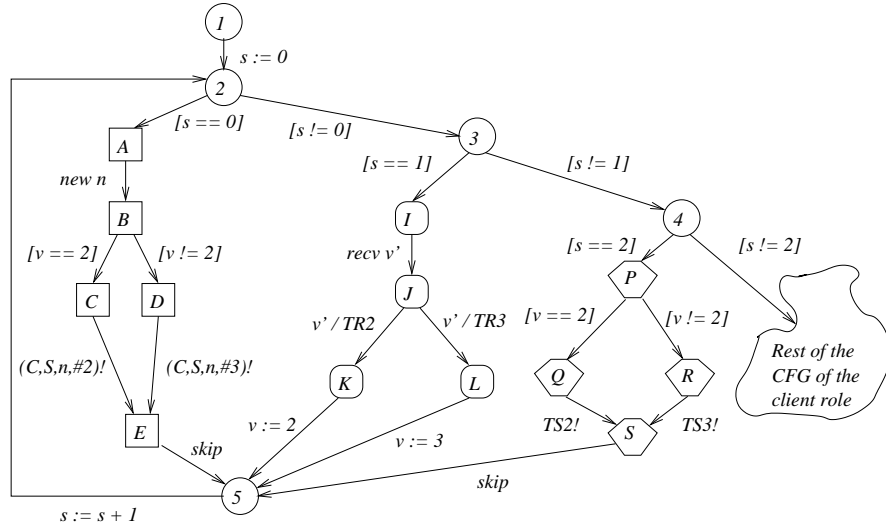


Fig. 4. The role derived from the `client` procedure from Fig. 2 via specification inlining using the library routine SSMs from Figure 3.

The input parameters i and r are instantiated with names C and S of the client and the server 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) $Id = (\eta, C)$, (ii) $\nu = \{i = C, r = S\}$, (iii) $S = \{1, 2, \dots\}$, (iv) $I = 1$, and (v) T is as shown in Figure 4.

3.2 Protocol Language Semantics: Preliminaries

In this section, we present some basic concepts used for describing the semantics of a protocol. We begin with standard concepts from protocol specification analysis: *attacker capabilities* and *knowledge*, and a *context* that describes the

$$\begin{aligned}
\Gamma \vdash m \wedge \Gamma \vdash k &\Longrightarrow \Gamma \vdash \{m\}_k \\
\Gamma \vdash \{m\}_k \wedge \Gamma \vdash k^{-1} &\Longrightarrow \Gamma \vdash m \\
\Gamma \vdash m \wedge \Gamma \vdash k' &\Longrightarrow \Gamma \vdash \text{Sig}(k', m) \\
\Gamma \vdash m \wedge \Gamma \vdash k &\Longrightarrow \Gamma \vdash \text{Hash}_k(m) \\
\Gamma \vdash m_1 \wedge \Gamma \vdash m_2 &\Longrightarrow \Gamma \vdash (m_1, m_2) \\
\Gamma \vdash (m_1, m_2) &\Longrightarrow \Gamma \vdash m_1 \wedge \Gamma \vdash m_2
\end{aligned}$$

Fig. 5. Attacker message derivation rules; $k' = \text{privkey}(N)$ for some name N .

$$\begin{aligned}
\mathcal{R}_{\text{new } v}^{\text{Msg}} &= \{(\nu, \Gamma, \nu', \Gamma') \mid \nu' = \nu \bowtie \{v = n\} \wedge \Gamma' = \Gamma\} \text{ where } n \text{ is a fresh nonce} \\
\mathcal{R}_{\text{send } t}^{\text{Msg}} &= \{(\nu, \Gamma, \nu', \Gamma') \mid (\nu[t] \neq \perp) \wedge (\nu' = \nu) \wedge (\Gamma' = \Gamma \cup \{\nu[t]\})\} \\
\mathcal{R}_{\text{recv } v}^{\text{Msg}} &= \{(\nu, \Gamma, \nu', \Gamma') \mid \nu' = \nu \bowtie \{v = m\} \wedge \Gamma \vdash m \wedge \Gamma' = \Gamma\} \\
\mathcal{R}_{v:=m}^{\text{Msg}} &= \{(\nu, \Gamma, \nu', \Gamma') \mid \nu' = \nu \bowtie \{v = m\} \wedge \Gamma' = \Gamma\} \\
\mathcal{R}_{\text{match } v/t}^{\text{Msg}} &= \{(\nu, \Gamma, \nu', \Gamma') \mid \nu' = \nu \bowtie \text{match}(E_\nu(v), t) \wedge \Gamma' = \Gamma \wedge E'_\sigma = E_\sigma\} \\
\mathcal{R}_{v:=e}^{\text{S}} &= \{(\sigma, \sigma') \mid \sigma' = \sigma \bowtie \{v = \sigma[e]\}\} & \mathcal{R}_{\text{assume } e}^{\text{S}} &= \{(\sigma, \sigma) \mid \sigma[e] \neq 0\}
\end{aligned}$$

Fig. 6. Transformer rules for statements.

values of various message variables in the different role instances as they execute. The execution of the protocol actions updates the context and the attacker knowledge as described formally by the transition rules in Figure 6.

We then augment the formalism with a *store* which keeps track of the mapping of numeric variables to constants. The store is updated by assignment actions (see the last rule in Figure 6); values of numeric variables read from the store affects the control flow of the program via the conditionals that appear in the CFG (as illustrated in the example in Figure 2).

Attacker Model. We use the standard symbolic (Dolev-Yao) attacker model [20, 40]. The attacker capabilities are represented via the inference rules shown in Figure 5, where $\Gamma \vdash m$ means that the attacker can compute message m from the set Γ of messages. In addition, the attacker can generate nonces and message constants. 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. Finally, the attacker has an identity on the network, i.e. a name and corresponding private and public keys. We will use Γ (with decorations) to denote the attacker’s knowledge throughout the paper.

Context and Protocol Actions. Recall that a context maps message variables (that appear in role instances) to messages. Protocol actions such as sending and receiving messages updates the context and the attacker knowledge as depicted in Figure 5. The semantics of most protocol actions is standard. However, pattern

matching (which is used to model operations such as projection, decryption and signature verification) makes use of a function $match$. Specifically, $match : Msg \times Term \mapsto \mathcal{C}$ 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 $match(m, t)$ is undefined and is denoted by \perp .

Store. A store represents an assignment to numeric variables. Formally, a store $\sigma : NVar \mapsto \mathbb{Z}$ is a mapping from numeric variables to constants. The set of all stores is denoted by \mathcal{S} . For any store σ , and any expression e , we write $\sigma[e]$ to mean the integer obtained by replacing each variable v in e with $\sigma(v)$ and then performing standard arithmetic operations. For any $\sigma \in \mathcal{S}$, $v \in NVar$ and $c \in \mathbb{Z}$, $\sigma \bowtie \{v = c\}$ is the store σ' such that:

$$\forall v' \in NVar. (v' = v \wedge \sigma'(v') = c) \vee (v' \neq v \wedge \sigma'(v') = \sigma(v'))$$

Environment. An *environment* for a thread \mathcal{T} is a triple $(\sigma, \nu, \Gamma)^{\mathcal{T}}$ such that $\sigma \in \mathcal{S}$, $\nu \in \mathcal{C}$ and $\Gamma \subseteq Msg$. The set $\mathcal{S} \times \mathcal{C} \times 2^{Msg}$ of all environments is denoted by \mathcal{E} . For any environment $E = (\sigma, \nu, \Gamma)^{\mathcal{T}}$, we write E_σ , E_ν and E_Γ to mean σ , ν and Γ respectively. Environments model concrete information available to the thread and the attacker during the thread's execution. Specifically, an environment $(\sigma, \nu, \Gamma)^{\mathcal{T}}$ means that the thread \mathcal{T} 's numeric and messages variable binding is σ and ν respectively, 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. We view a statement as an environment transformer. However, action statements only transform the ν and Γ components of an environment, while non-action statements transform only the σ component of an environment. Thus, we first present these transformers separately.

1. For any action statement St , the transformer relation $\mathcal{R}_{St}^{Msg} \subseteq \mathcal{C} \times 2^{Msg} \times \mathcal{C} \times 2^{Msg}$ is shown in Figure 6. For any non-action statement St , \mathcal{R}_{St}^{Msg} is the identity relation.
2. For any non-action statement St , the transformer relation $\mathcal{R}_{St}^{\mathcal{S}} \subseteq \mathcal{S} \times \mathcal{S}$ is shown in Figure 6. For any action statement St , $\mathcal{R}_{St}^{\mathcal{S}}$ is the identity relation.
3. Finally, for any statement St , the concrete transformer relation $\mathcal{R}_{St} \subseteq \mathcal{E} \times \mathcal{E}$ is defined as follows:

$$\mathcal{R}_{St} = \{((\sigma, \nu, \Gamma), (\sigma', \nu', \Gamma')) \mid (\sigma, \sigma') \in \mathcal{R}_{St}^{\mathcal{S}} \wedge (\nu, \Gamma, \nu', \Gamma') \in \mathcal{R}_{St}^{Msg}\}$$

3.3 Protocol Language Concrete Semantics

We defined protocols and threads in Section 3.1. We now define how a finite number of threads execute concurrently. This is meant to capture the scenario where, for example, 3 client and 2 server threads of SSL are running concurrently. The formal definition is presented using Labeled Transition Systems (LTSs). Note that this model is similar to those used for protocol analysis. The presentation

of the model is different because we want to align it with models used for software model checking. We begin by defining LTSs.

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.

Concrete Thread Model. Let $\mathcal{T} = (Id, \nu, S, I, T)$ be a thread. Let us write E_I to mean $\mathcal{S} \times \{\nu\} \times \{I_0\}$ where $I_0 \subseteq Msg$ denotes the attacker's initial knowledge. Then the concrete model of \mathcal{T} is the LTS $M(\mathcal{T}) = (\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$ such that (i) $\mathbb{S} = S \times \mathcal{E}$, (ii) $\mathbb{I} = \{I\} \times E_I$, (iii) $\Sigma = (Stmt \times \{Id\}) \cup \{m\# \mid m \in Msg\}$, and:

$$\mathbb{T} = \{((s, E), (St, Id), (s', E')) \mid (s, St, s') \in T \wedge (E, E') \in \mathcal{R}_{St}\} \\ \cup \{((s, E), m\#, (s, E)) \mid s \in S \wedge E \in \mathcal{E} \wedge E_I \vdash m\}$$

The different components of the LTS can be understood as follows: (i) states are represented by the states of the CFG together with an environment that maps variables (message and numeric) to values and keeps track of the attacker knowledge; (ii) the initial states are obtained by combining the initial state of the CFG with an initial environment; (iii) the alphabet of events consists of protocol statements along with the identifier of the thread that executed the statement; (iv) a transition $((s, E), (St, Id), (s', E'))$ exists in the LTS if the edge between s and s' on the CFG is labeled by the statement St ; as a consequence, the environment E is updated to E' following the environment transformer relation. The event $m\#$ indicates that the attacker derived m and sent it out on the network, i.e. the secrecy of m has been violated. This is special transition useful for specifying secrecy properties.

Concrete Thread Composition. We assume that threads execute asynchronously and have disjoint sets of variables. We now present the concrete model of the composition of two threads (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 $M(\mathcal{T}_1) = (\mathbb{S}_1, \mathbb{I}_1, \Sigma_1, \mathbb{T}_1)$ and $M(\mathcal{T}_2) = (\mathbb{S}_2, \mathbb{I}_2, \Sigma_2, \mathbb{T}_2)$ be their respective concrete models. Let $E_I = \mathcal{S} \times \{\nu_1 \bowtie \nu_2\} \times \{I_0\}$. Then the composed model of the two threads $M(\mathcal{T}_1, \mathcal{T}_2)$ is the LTS $(\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$ such that: (i) $\mathbb{S} = S_1 \times S_2 \times \mathcal{E}$, (ii) $\mathbb{I} = \{I_1\} \times \{I_2\} \times E_I$, (iii) $\Sigma = \Sigma_1 \cup \Sigma_2$, and:

$$\mathbb{T} = \{((s_1, s_2, E), X, (s'_1, s'_2, E'))\}$$

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

3.4 Security Properties and Satisfaction

For any sequence w and any set u of events, we write $w \downarrow u$ to mean the subsequence of w obtained by eliding events not in u . For instance, $\langle \alpha, \gamma, \beta, \beta, \gamma, \beta, \alpha \rangle \downarrow$

$\{\alpha, \gamma\} = \langle \alpha, \gamma, \gamma, \alpha \rangle$. We deal with two types of security properties – authentication and secrecy. Let $M = (\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$ be any concrete model.

Authentication. An authentication property φ ensures that certain events always happen in a specific order. It is specified by a finite sequence of events $\langle \alpha_1, \dots, \alpha_n \rangle$. A counterexample to φ is a finite sequence $\langle q_0, b_1, q_1, \dots, b_k, q_k \rangle$ such that: (i) $q_0 \in \mathbb{I}$, (ii) $\forall 1 \leq i \leq k \cdot (q_{i-1}, b_i, q_i) \in \mathbb{T}$, (iii) $b_k = \alpha_n$, and (iv) $\langle b_1, \dots, b_k \rangle \downarrow \{\alpha_1, \dots, \alpha_n\} \neq \langle \alpha_1, \dots, \alpha_n \rangle$.

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 $\langle \alpha_1, \alpha_2, \alpha_3, \alpha_4 \rangle$ is the sequence of messages exchanged between a client and server during a correct run of the SSL handshake.

Secrecy. A secrecy property φ ensures the inability of the attacker to compute a specific message m . A counterexample to φ is a finite sequence $q_0, b_1, q_1, \dots, b_k, q_k$ such that: (i) $q_0 \in \mathbb{I}$, (ii) $\forall 1 \leq i \leq k \cdot (q_{i-1}, b_i, q_i) \in \mathbb{T}$, and (iii) $b_k = m\#$. Note that both authentication and secrecy are instances of reachability properties.

Property Satisfaction. The composition of threads $\mathcal{T}_1, \dots, \mathcal{T}_n$ satisfies a property φ iff $M(\mathcal{T}_1, \dots, \mathcal{T}_n)$ satisfies φ . Let \mathcal{Q} be a protocol with n roles. A *configuration* $Conf$ of \mathcal{Q} is a function from $\{1, \dots, n\}$ to natural numbers. Intuitively, $Conf(i)$ is the number of threads instantiating the i^{th} role of \mathcal{Q} . Then \mathcal{Q} satisfies a property φ under $Conf$ iff every composition (obtained by instantiating the input parameters) of $\sum_i Conf(i)$ threads (with the first $Conf(1)$ threads instantiating the first role, the next $Conf(2)$ threads instantiating the second role, and so on) satisfies φ .

4 Abstract Security Protocol Model and Soundness

Even though authentication and secrecy are reachability properties, they cannot be verified via direct reachability analysis of the concrete protocol since, in general, the concrete model has an infinite set of states. To overcome this problem, SPIER performs reachability analysis on an *abstract protocol model*. In this section, we describe how the abstract model is extracted automatically from the concrete protocol using *predicate abstraction*. In the abstract model, sets of stores are represented in terms of predicates that they satisfy (e.g. $x > 0$). This enables us to represent an infinite set of concrete states (i.e., stores) using a finite set of abstract states (i.e., predicates). The main result of this section is a *soundness theorem* (Theorem 1), which states that if a security property holds in the abstract protocol model, then it also holds in the concrete (real) protocol model².

² Note, however, that the failure of a property in the abstract model does *not* imply its failure in the concrete model. SPIER’s behavior in such situations is the topic of the next section.

4.1 Abstract Protocol Model

We begin our presentation of the abstract protocol model with a description of the concepts behind predicate abstraction.

Predicate. 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 set of all valuations of \mathcal{P} is denoted by $\mathcal{V}_{\mathcal{P}}$. Given a store σ , we write $V_{\mathcal{P}}(\sigma)$ to mean the unique valuation of \mathcal{P} defined as follows:

$$\forall p \in \mathcal{P} . (V_{\mathcal{P}}(\sigma)(p) = \text{TRUE} \iff \sigma[p] \neq 0)$$

Note that we use C-style semantics to convert between expressions and Boolean formulas, i.e., something is TRUE iff it is non-zero.

Concretization. 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 $V(p) \implies \gamma^V(p) = p$ and $\neg V(p) \implies \gamma^V(p) = \neg p$.

Weakest Precondition. 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) $\mathcal{WP}\{e\}[\text{assume } e'] = e \wedge e'$, and (c) $\mathcal{WP}\{e\}[Act] = e$.

Abstract Transformer. In predicate abstraction [25], every statement is viewed as a predicate valuation transformer. Specifically, for any statement St and set of predicates \mathcal{P} , the transformer relation $\mathcal{R}_{St, \mathcal{P}}^{\mathcal{V}} \subseteq \mathcal{V}_{\mathcal{P}} \times \mathcal{V}_{\mathcal{P}}$ is defined as follows:

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

Intuitively, V is related to V' by the abstract transformer $\mathcal{R}_{St, \mathcal{P}}^{\mathcal{V}}$ if there are stores σ and σ' such that $V = V_{\mathcal{P}}(\sigma)$, $V' = V_{\mathcal{P}}(\sigma')$, and there is a concrete transition from σ to σ' . We use an automated theorem prover to check for the satisfiability of $\gamma(V) \wedge \mathcal{WP}\{\gamma(V')\}[St]$. If the theorem prover does not return a definite answer, we assume that $\gamma(V) \wedge \mathcal{WP}\{\gamma(V')\}[St]$ is satisfiable to preserve the soundness of our abstraction. The following fact, which we state without proof, describes the precise correspondence between store transformers and predicate valuation transformers. It is used later on (see Lemma 1) to prove a similar correspondence between concrete and abstract environment transformers.

Fact 1 *For any set of predicates \mathcal{P} and any statement St :*

$$\forall \sigma, \sigma' \in \mathcal{S} . (\sigma, \sigma') \in \mathcal{R}_{St}^{\mathcal{S}} \implies (V_{\mathcal{P}}(\sigma), V_{\mathcal{P}}(\sigma')) \in \mathcal{R}_{St, \mathcal{P}}^{\mathcal{V}}$$

Abstract Environment. An abstract environment corresponds to an environment where all numerical data is represented as predicate valuations. Let \mathcal{P} be

a set of predicates. Then the set of all abstract environments over \mathcal{P} , denoted by $\widehat{\mathcal{E}}_{\mathcal{P}}$, is $\mathcal{V}_{\mathcal{P}} \times \mathcal{C} \times 2^{Msg}$. For any environment $E = (\sigma, \nu, \Gamma)$ we write \widehat{E} to mean the corresponding abstract environment $(V_{\mathcal{P}}(\sigma), \nu, \Gamma)$. The predicate valuation transformer $\mathcal{R}_{St, \mathcal{P}}^{\mathcal{V}}$ described above naturally induces an abstract environment transformer $\mathcal{R}_{St, \mathcal{P}} \subseteq \widehat{\mathcal{E}}_{\mathcal{P}} \times \widehat{\mathcal{E}}_{\mathcal{P}}$ as follows:

$$\mathcal{R}_{St, \mathcal{P}} = \{((V, \nu, \Gamma), (V', \nu', \Gamma')) \mid (V, V') \in \mathcal{R}_{St, \mathcal{P}}^{\mathcal{V}} \wedge (\nu, \Gamma, \nu', \Gamma') \in \mathcal{R}_{St}^{Msg}\}$$

Note that $\mathcal{R}_{St, \mathcal{P}}$ is computable since both $\mathcal{R}_{St, \mathcal{P}}^{\mathcal{V}}$ and \mathcal{R}_{St}^{Msg} are computable. The following lemma describes the precise correspondence between concrete and abstract environment transformers. It is used subsequently (see Lemma 2 and Theorem 1) to prove the critical soundness result about our abstraction.

Lemma 1. *For any set of predicates \mathcal{P} and any statement St :*

$$\forall E, E' \in \mathcal{E} . (E, E') \in \mathcal{R}_{St} \implies (\widehat{E}, \widehat{E}') \in \mathcal{R}_{St, \mathcal{P}}$$

Proof. Let $E = (\sigma, \nu, \Gamma)$ and $E' = (\sigma', \nu', \Gamma')$ such that $(E, E') \in \mathcal{R}_{St}$. From the definition of \mathcal{R}_{St} we know that $(\sigma, \sigma') \in \mathcal{R}_{St}^S$ and $(\nu, \Gamma, \nu', \Gamma') \in \mathcal{R}_{St}^{Msg}$. Then, by Fact 1, we know that $(V_{\mathcal{P}}(\sigma), V_{\mathcal{P}}(\sigma')) \in \mathcal{R}_{St, \mathcal{P}}^{\mathcal{V}}$. Hence, from the definition of $\mathcal{R}_{St, \mathcal{P}}$, we know that $((V_{\mathcal{P}}(\sigma), \nu, \Gamma), (V_{\mathcal{P}}(\sigma'), \nu', \Gamma')) \in \mathcal{R}_{St, \mathcal{P}}$, which is what we want. \square

Abstract Thread Model. Let $\mathcal{T} = (Id, \nu, S, I, T)$ be a thread, and \mathcal{P} be a set of predicates. Let us write \widehat{E}_I to mean $\mathcal{V}_{\mathcal{P}} \times \{\nu\} \times \{I_0\}$ where $I_0 \subseteq Msg$ is the attacker's initial knowledge. 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) $\mathbb{S} = S \times \widehat{\mathcal{E}}_{\mathcal{P}}$, (ii) $\mathbb{I} = \{I\} \times \widehat{E}_I$, (iii) $\Sigma = (Stmt \times \{Id\}) \cup \{m\# \mid m \in Msg\}$, and \mathbb{T} is the following relation:

$$\begin{aligned} & \{((s, \widehat{E}), (St, Id), (s', \widehat{E}')) \mid (s, St, s') \in T \wedge (E, E') \in \mathcal{R}_{St, \mathcal{P}}\} \\ & \cup \{((s, \widehat{E}), m\#, (s, \widehat{E})) \mid s \in S \wedge \widehat{E} \in \widehat{\mathcal{E}} \wedge E_{\Gamma} \vdash m\} \end{aligned}$$

$$\begin{aligned} & (1, \{\neg p_0, \neg p_1, \neg p_2, v_2\}, (\nu, \Gamma_0)) \xrightarrow{(s:=0, Id)} (2, \{p_0, \neg p_1, \neg p_2, v_2\}, (\nu, \Gamma_0)) \\ & (2, \{p_0, \neg p_1, \neg p_2, v_2\}, (\nu, \Gamma_0)) \xrightarrow{(\text{assume}(s==0), Id)} (A, \{p_0, \neg p_1, \neg p_2, v_2\}, (\nu, \Gamma_0)) \\ & (A, \{p_0, \neg p_1, \neg p_2, v_2\}, (\nu, \Gamma_0)) \xrightarrow{(\alpha_1, Id)} (B, \{p_0, \neg p_1, \neg p_2, v_2\}, (\nu \bowtie \{n = n_0\}, \Gamma_0)) \\ & (B, \{p_0, \neg p_1, \neg p_2, v_2\}, (\nu, \Gamma_0)) \xrightarrow{(\alpha_2, Id)} (C, \{p_0, \neg p_1, \neg p_2, v_2\}, (\nu \bowtie \{n = n_0\}, \Gamma_0)) \end{aligned}$$

Fig. 7. Some sample transitions in the abstract composed model of our example. n_0 is a nonce; $\alpha_1 \equiv \text{new } n$; $\alpha_2 \equiv \text{assume}(v == 2)$.

Example 2. Recall, from Figure 4, our example client thread $\mathcal{T} = (Id, \nu, S, I, T)$ where: (i) $Id = (\eta, \mathbf{C})$, (ii) $\nu = \{i = \mathbf{C}, r = \mathbf{S}\}$, (iii) $S = \{1, 2, \dots\}$, (iv) $I = 1$, and (v) T is as shown in Figure 4. Let $\mathcal{P} = \{p_0, p_1, p_2, v_2\}$ be a set of predicates such that $p_0 \equiv (\mathbf{s} == 0)$, $p_1 \equiv (\mathbf{s} == 1)$, $p_2 \equiv (\mathbf{s} == 2)$ and $v_2 \equiv (\mathbf{v} == 2)$. Then the thread model $M(\mathcal{T}, \mathcal{P})$ is the LTS some of whose important transitions are shown in Figure 7.

Abstract Thread Composition. 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 $M(\mathcal{T}_1, \mathcal{P}) = (\widehat{\mathbb{S}}_1, \widehat{\mathbb{I}}_1, \Sigma_1, \widehat{\mathbb{T}}_1)$ and $M(\mathcal{T}_2, \mathcal{P}) = (\widehat{\mathbb{S}}_2, \widehat{\mathbb{I}}_2, \Sigma_2, \widehat{\mathbb{T}}_2)$ be their abstract models over some \mathcal{P} . Let $\widehat{E}_I = \mathcal{V}_{\mathcal{P}} \times \{\nu_1 \bowtie \nu_2\} \times \{I_0\}$. Then the composed model of the two threads $M(\mathcal{T}_1, \mathcal{T}_2, \mathcal{P})$ is the LTS $(\widehat{\mathbb{S}}, \widehat{\mathbb{I}}, \Sigma, \widehat{\mathbb{T}})$ such that: (i) $\widehat{\mathbb{S}} = S_1 \times S_2 \times \widehat{\mathcal{E}}_{\mathcal{P}}$, (ii) $\widehat{\mathbb{I}} = \{I_1\} \times \{I_2\} \times \widehat{E}_I$, (iii) $\Sigma = \Sigma_1 \cup \Sigma_2$, and (iv) $\widehat{\mathbb{T}}$ is defined as follows:

$$\widehat{\mathbb{T}} = \{((s_1, s_2, E), X, (s'_1, s'_2, E'))\}$$

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

4.2 Soundness of Abstract Model

Simulation relations between LTS's are useful for specifying and reasoning about properties. Later in this section, we prove that the LTS corresponding to the concrete (real) protocol is simulated by the LTS for the abstract protocol model (Lemma 2) and use this result to justify the soundness of the approach (Theorem 1).

Simulation. An LTS $M_1 = (\mathbb{S}_1, \mathbb{I}_1, \Sigma, \mathbb{T}_1)$ is said to be simulated [36] by an LTS $M_2 = (\mathbb{S}_2, \mathbb{I}_2, \Sigma, \mathbb{T}_2)$ (written as $M_1 \preceq M_2$) iff there exists a relation $R \subseteq \mathbb{S}_1 \times \mathbb{S}_2$ such that the following two conditions hold:

$$\forall s_1 \in \mathbb{I}_1. \exists s_2 \in \mathbb{I}_2. s_1 R s_2 \quad (\mathbf{INIT})$$

$$\forall s_1, s'_1 \in \mathbb{S}_1. \forall s_2 \in \mathbb{S}_2. \forall a \in \Sigma. s_1 R s_2 \wedge (s_1, a, s'_1) \in \mathbb{T}_1 \implies$$

$$\exists s'_2 \in \mathbb{S}_2. (s_2, a, s'_2) \in \mathbb{T}_2 \wedge s'_1 R s'_2 \quad (\mathbf{STEP})$$

Lemma 2 (Simulation). *Let \mathcal{T}_1 and \mathcal{T}_2 be two threads, and \mathcal{P} be a set of predicates. Then:*

$$M(\mathcal{T}_1, \mathcal{T}_2) \preceq M(\mathcal{T}_1, \mathcal{T}_2, \mathcal{P})$$

Proof. Let $M(\mathcal{T}_1, \mathcal{T}_2) = (\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$ and $M(\mathcal{T}_1, \mathcal{T}_2, \mathcal{P}) = (\widehat{\mathbb{S}}, \widehat{\mathbb{I}}, \Sigma, \widehat{\mathbb{T}})$. Consider the relation $R \subseteq \mathbb{S} \times \widehat{\mathbb{S}}$ defined as follows:

$$R = \{((s_1, s_2, E), (s_1, s_2, \widehat{E})) \mid (s_1, s_2, E) \in \mathbb{S}\}$$

We now prove that R is a simulation relation. For the **INIT** condition, note that:

$$\forall (s_1, s_2, E) \in \mathbb{I}. (s_1, s_2, \widehat{E}) \in \widehat{\mathbb{I}}$$

For the **STEP** condition, suppose that $((s_1, s_2, E), \alpha, (s'_1, s'_2, E')) \in \mathbb{T}$. W.l.o.g, suppose that $((s_1, E), \alpha, (s'_1, E')) \in \mathbb{T}_1$. Then from Lemma 1 and the definition of \mathbb{T}_1 and $\widehat{\mathbb{T}}_1$, we know that $((s_1, \widehat{E}), \alpha, (s'_1, \widehat{E}')) \in \widehat{\mathbb{T}}_1$. Hence, from the definition of $\widehat{\mathbb{T}}$, $((s_1, s_2, \widehat{E}), \alpha, (s'_1, s'_2, \widehat{E}')) \in \mathbb{T}$, and $((s'_1, s'_2, E'), (s'_1, s'_2, \widehat{E}')) \in R$ \square

Theorem 1 (Soundness). *Let \mathcal{T}_1 and \mathcal{T}_2 be two threads, and \mathcal{P} be a set of predicates. Then, for any security property φ :*

$$M(\mathcal{T}_1, \mathcal{T}_2, \mathcal{P}) \models \varphi \implies M(\mathcal{T}_1, \mathcal{T}_2) \models \varphi$$

Proof. We prove the contrapositive. Let φ be an authentication or secrecy property. W.l.o.g. let $CE = \langle q_0, \alpha_1, q_1, \dots, \alpha_k, q_k \rangle$ be a counterexample to $M(\mathcal{T}_1, \mathcal{T}_2) \models \varphi$. Let $q_i = (s_{i1}, s_{i2}, E_i)$ for $0 \leq i \leq k$. Then, by Lemma 2, $\langle (s_{01}, s_{02}, \widehat{E}_0), \alpha_1, \dots, \alpha_k, (s_{k1}, s_{k2}, \widehat{E}_k) \rangle$ is a counterexample to $M(\mathcal{T}_1, \mathcal{T}_2, \mathcal{P}) \models \varphi$. \square

Note that, in general, simulation preserves all ACTL* properties [14]. Thus, our abstraction preserves not only safety properties (which includes authentication and secrecy), but also non-safety properties (such as non-interference [23]) which are expressible in ACTL*.

In the first step of our methodology we extract an abstract model M of our concrete system using the technique presented in this section. In the second step, we verify the desired security property φ on M via reachability analysis. If $M \models \varphi$, then we know that the concrete system satisfies φ as well. In this case, we exit with “System Secure”. Otherwise we obtain a counterexample CE – finite trace of the abstract model M – and proceed with the remaining steps of counterexample validation and refinement, as described in the next section.

5 Counterexample Validation and Refinement

In this section, we define our procedure for counterexample validation and abstraction refinement in case the verification step on the abstract model yields a counterexample.

5.1 Counterexample Validation

Let the counterexample be a sequence of abstract states $CE = \langle \widehat{q}_0, \alpha_1, \widehat{q}_1, \dots, \alpha_k, \widehat{q}_k \rangle$. Let $\widehat{q}_i = (s_i^1, s_i^2, (V_i, \nu_i, \Gamma_i))$ for $0 \leq i \leq k$, and $\alpha_i = (St_i, Id_1)$ for $1 \leq i \leq k$.

Counterexample Validity. Recall that the concrete model is the LTS $M(\mathcal{T}_1, \mathcal{T}_2) = (\mathbb{S}, \mathbb{I}, \Sigma, \mathbb{T})$. For each $0 \leq i \leq k$, and each $\sigma \in \mathcal{S}$, we say that $\sigma \models CE(i)$ iff there exists a sequence of concrete states q_i, \dots, q_k of $M(\mathcal{T}_1, \mathcal{T}_2)$, and a sequence of stores $\sigma_i, \dots, \sigma_k$ such that: (i) for $i \leq j \leq k$, $q_j = (s_j^1, s_j^2, (\sigma_j, \nu_j, \Gamma_j))$, (ii) for $i \leq j < k$, $(q_i, \alpha_i, q_{i+1}) \in \mathbb{T}$, and (iii) $\sigma_i = \sigma$.

Intuitively, $\sigma \models CE(i)$ iff the concrete model $M(\mathcal{T}_1, \mathcal{T}_2)$ has a trace corresponding to the suffix $\hat{q}_i, \dots, \hat{q}_k$ of CE starting with the store σ . Finally, CE is valid iff $\exists \sigma \in \mathcal{S} \cdot \sigma \models CE(0)$.

Checking Validity of CE . The key idea behind checking the validity of CE is to compute, for each $0 \leq i \leq k$, an expression representing all stores σ such that $\sigma \models CE(i)$. Such an expression is called a verification condition. Formally, for $0 \leq i \leq k$, the verification condition VC_i is an expression that satisfies the following condition: $\forall \sigma \in \mathcal{S} \cdot \sigma \models CE(i) \iff \sigma[VC_i] \neq 0$. Fortunately, there is an effective procedure for computing verification conditions based on weakest preconditions, as defined below:

$$VC_k = \text{TRUE} \bigwedge VC_i = \text{WP}\{VC_{i+1}\}[St_{i+1}] \text{ for } 0 \leq i < k$$

Finally, from our definition of validity above, CE is valid iff VC_0 is satisfiable. This is decided via an automated theorem prover. If VC_0 is satisfiable, we report CE to be valid. If the theorem prover does not return a definite answer (since the problem is undecidable in general), we also report CE to be valid to preserve the soundness of our approach. If VC_0 is unsatisfiable, then CE is a spurious counterexample, and we proceed with abstraction refinement, as described in the next section.

5.2 Refinement

We refine our abstraction by changing our set of predicates to $\mathcal{P}' = \mathcal{P} \cup \{VC_0, \dots, VC_k\}$. In other words, we add all the verification conditions computed while validating CE to our set of predicates. We now argue that the abstract model computed with \mathcal{P}' can never yield a counterexample labeled with the same sequence of actions as CE . Indeed, suppose that we get a new counterexample $CE' = \langle \hat{q}'_0, \alpha_1, \hat{q}'_1, \dots, \alpha_k, \hat{q}'_k \rangle$ such that $\hat{q}'_i = (s_i^1, s_i^2, (V'_i, \nu'_i, \Gamma'_i))$ for $0 \leq i \leq k$.

Since VC_0 is unsatisfiable, $V'_0(VC_0) = \text{FALSE}$. Using the definition of the predicate valuation transformer $\mathcal{R}_{St, \mathcal{P}}$, and the definition of VC_i , it can be shown that for $0 \leq i < k$, $V'_i(VC_i) = V'_{i+1}(VC_{i+1})$. Therefore $V'_k(VC_k) = \text{FALSE}$. But note that $VC_k = \text{TRUE}$. Therefore, from the definition of $\mathcal{R}_{St, \mathcal{P}}$, the transition $(\hat{q}'_{k-1}, \alpha_k, \hat{q}'_k)$ is not possible, which is a contradiction.

5.3 Soundness and Completeness

The soundness and completeness of our approach (modulo the correctness of the library routine SSMs used) is summarized as follows:

1. If our approach reports that a protocol \mathcal{Q} satisfies a property φ under a configuration $Conf$, then \mathcal{Q} is indeed satisfies φ under $Conf$.
2. If our approach reports an attack on a protocol and the theorem prover gave a definite answer during the counterexample validation step, then it is a real attack on the protocol.
3. Our approach is incomplete in general, i.e. it may not terminate. This is acceptable since the overall problem is undecidable.

6 Tool Implementation

We implemented ASPIER on top of an existing iterative refinement based software model checker COPPER [13]. The input to ASPIER 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 $Conf$ of \mathcal{Q} , and (iv) a mapping κ from cryptographic libraries to finite state machine labeled with actions.

The output of ASPIER is either (i) “System Secure” – indicating that φ holds for \mathcal{Q} under $Conf$, (ii) “Real Attack” – indicating the existence of an attack, or (iii) “Unknown” – indicating that ASPIER could not arrive at either of the previous two conclusions. In the case of a “Real Attack”, ASPIER also emits a trace that exhibits a possible attack on \mathcal{Q} under $Conf$ that leads to a violation of φ . We now discuss the implementation of each step of our framework.

Thread Construction. The threads are obtained from the source code by constructing the control flow graph, replacing every cryptographic library call l with the state machine $\kappa(l)$, and instantiating the input parameters of the different roles.

Abstraction. The abstract model is then computed following the definitions in Section 4. The Simplify theorem prover is used to compute $\mathcal{R}_{St, \mathcal{P}}^V$. To compute \mathcal{R}_{St}^{Msg} , we have implemented and experimented with two decision procedures for message derivation. The first decision procedure (called **Simplify**) works by: (a) formalizing the attacker model as a logical theory, (b) reducing the message derivation problem to a validity problem in this theory, and (c) checking validity using Simplify.

The second decision procedure (called **Decons-Cons**) is based on the idea that message derivation can be decided via deconstruction followed by construction [43]. In essence, to decide if $\Gamma \vdash m$, we first compute the set of all terms that can be derived from Γ' using the second and sixth rules in Figure 5. Next, we compute the set of all terms Γ'' that can be derived from Γ' using the remaining rules in Figure 5. Finally, $\Gamma \vdash m \iff m \in \Gamma''$. Experimental results comparing between **Simplify** and **Decons-Cons** are presented in Section 7.

For simplicity, our technical presentation of predicate abstraction was based on a global set of predicates. In practice, COPPER uses different sets of predicates at different CFG nodes for efficiency. Note that ASPIER augments COPPER by combining $\mathcal{R}_{St, \mathcal{P}}^V$ with \mathcal{R}_{St}^{Msg} .

Verification. Verification is performed via explicit state reachability analysis. Recall that a connection topology is a function mapping each client session C to the server with which C initiates a connection. A key aspect of ASPIER, compared to COPPER, is that each possible connection topology (modulo a simple symmetry reduction) is verified separately, thereby trading off time for space. This technique is motivated by the observation that different connection topologies yield fairly disjoint statespaces. For example, the statespace of a system where the first client session connects with the first server session is quite different from the one where the first client session connects with the attacker. Thus, independent exploration of different parameter instantiations is not penalized

Server	Client	Server-Rollback					Client-Rollback				
		Present	Inst	T1	T2	Mem	Present	Inst	T1	T2	Mem
1-SSL3	1-SSL3	No	4	4.8	4.5	18.8	No	4	5.8	5.1	18.8
1-SSL3	1-SSL23	No	4	5.2	4.7	18.9	No	4	6.2	5.3	19.2
1-SSL23	1-SSL3	No	4	6.0	5.2	19.7	No	4	7.3	6.0	19.7
1-SSL23	1-SSL23	Yes	4	79.9	77.8	37.7	Yes	4	79.3	78.7	34.0
2-SSL3	2-SSL3	No	25	72749	35893	415	No	25	99224	50165	437
2-SSL3	2-SSL23	No	25	96740	49868	520	No	25	134869	111530	632
2-SSL23	2-SSL3	No	25	171308	86373	966	No	25	258459	162185	1294
2-SSL23	2-SSL23	Yes	1	5292	2719	136	Yes	1	4249	2711	139

Fig. 8. Experimental results for “version-rollback” attack. n -SSL3 = n OpenSSL3 clients/servers; n -SSL23 = n OpenSSL23 clients/servers; **Present** = whether rollback was detected; **Inst** = no. of parameter instantiations; **T1** = time (in seconds) with **Simplify** for message derivation; **T2** = time (in seconds) with **Decons-Cons** for message derivation; **Mem** = memory requirement (in MB) with **Decons-Cons** for message derivation; memory requirement with **Simplify** was slightly higher.

C#	S#	AuthSrvr			AuthClnt			Secrecy		
		Inst	Time	Mem	Inst	Time	Mem	Inst	Time	Mem
2	2	25	34987	410	25	48539	434	25	43936	424
3	3	225	226953	625	225	499535	1583	225	399279	1132

Fig. 9. Experimental results for different configurations of OpenSSL3 servers and clients. **C#** = no. of clients; **S#** = no. of servers; **Inst** = no. of parameter instantiations; **Time** = time (in seconds) with **Decons-Cons** for message derivation; **Mem** = memory requirement (in MB).

heavily by redundant work. As our experimental results indicate, this strategy appears to be a good foil to the statespace explosion problem in practice.

Counterexample Validation and Refinement. These two steps in ASPIER are essentially the same as COPPER. A few points are worth noting here. First, our counterexample validation procedure is based on weakest preconditions. However, counterexample validation based on strongest post-conditions [6] are also relevant. Second, instead of adding the verification conditions themselves as predicates, we add predicates derived from the “UNSAT core”, i.e., the smallest reason for the unsatisfiability of VC_0 . Finally, our abstraction refinement approach is also based on weakest preconditions. There are other abstraction refinement techniques, such as those based on interpolants [27], which are also applicable.

7 Experimental Results

We experimented with the C source code of OpenSSL version 0.9.6c that implements the initial handshake protocol between a server and a client. We performed

two verification tasks using various versions of OpenSSL 0.9.6c, as described below. All our experiments were done using a 2.4 GHz machine with 4 GB of RAM.

7.1 Detecting Version-Rollback Attack

In the first task, we confirmed the presence of the “version-rollback” attack in OpenSSL when it allows negotiation of the SSL version during the handshake phase. Since the version is not adequately integrity protected, a malicious attacker can force a client and a server to use SSL 2.0 even when both intend to use SSL 3.0. Let us denote the SSL implementation that allows version negotiation as *OpenSSL23* and the implementation of SSL hardwired to version 3.0 as *OpenSSL3*. We validated – for configurations of up to 2 servers and 2 clients – that when either the servers or the clients implement *OpenSSL3*, there is no version-rollback. This is essentially because integrity protection of the version by even one party is sufficient to prevent version-rollback. We also confirmed – for configurations of up to 2 servers and 2 clients – that when both the servers and the clients implement *OpenSSL23*, version-rollback exists.

Due to symmetry, it sufficed to verify check for version-rollback only for the first server and the first client. Also, each property was verified independently for every possible instantiation of the input parameters (modulo symmetry) of the client and server 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.

Our results are tabulated in Figure 8. We started each experiment with an initial set of predicates since our primary goal was to evaluate the effectiveness of ASPIER even without automated abstraction refinement. For the cases without rollback attacks, these initial predicates sufficed to prove the absence of rollbacks. For the cases with rollbacks, additional abstraction refinement was necessary before ASPIER found a real attack. In each such case, three iterations of abstraction refinement occurred and four new predicates were inferred from the spurious counterexamples.

For configurations with one server and one client, the cases without rollback actually completed very quickly, while the cases with rollback required more time and memory. This is essentially because, for these configurations, the abstract statespaces were quite small (in the order of a few thousand) and hence verification required less resources than abstraction refinement.

For configurations with two servers and two clients, the situation was just the reverse. In these cases, the abstract statespaces were much larger (in the order of millions) and hence verification completely swamped out abstraction refinement. Moreover, for the cases without rollback, all 25 parameter instantiations had to be verified. In contrast, for the cases with rollback, the very first parameter instantiation lead to the discovery of an attack, and hence to the termination of the verification procedure. Finally, it is noteworthy that as the number of *OpenSSL23* roles increase, so does the resource consumption. This is because *OpenSSL23* roles enable more behavior (since they allow version negotiation)

and hence lead to larger statespace. The obvious exception to this trend is when all roles are *OpenSSL3*, as explained above. Finally, verifying absence of client rollback requires more time and memory since the corresponding authentication property involves a longer sequence of events, and hence entails a deeper search.

7.2 Verifying Authentication and Secrecy of OpenSSL3

In the second task we verified, for configurations of up to three servers and three clients, 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.

As in the previous task, 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 (modulo symmetry) of the client roles. We started each experiment with an initial set of predicates to speed up the verification. These predicates sufficed to prove the security properties of interest. Figure 9 summarizes our results. Once again, verifying **AuthClnt** requires more time and memory than **AuthSrvr** since it involves a longer sequence of events, and hence entails a deeper search. Verifying **Secrecy** takes more time than **AuthSrvr** but less time than **AuthClnt**.

8 Related Work

There has not been much work on analysis of security protocol implementations. We summarize below some promising complementary approaches reported in recent work.

Bhargavan et al. [9] develop a method and tool for establishing security properties of protocols written in F# by translating it into the *applied π -calculus* and using PROVERIF [10], an automated tool, for verifying the resulting translation. They incorporate a symbolic attacker into the model just like us. While their security results hold for an unbounded number of concurrent sessions, their approach does not apply to legacy C code. They rely on a translation from a subset of λ -calculus to the *applied pi-calculus*. In contrast, our approach extracts a model from protocol implementations written in C using automated abstraction-refinement techniques. In a related effort, Bengtson et al. have developed a type system for verifying authentication properties of protocols written in F# [8].

Goubault-Larrecq and Parrennes [24] develop a tool which performs inter-procedural pointer analysis to derive Horn clauses from C code for protocol implementations. Horn clauses are also used to represent the intruder deduction relation and the external trust model. Secrecy properties are then cast in terms of a satisfiability problem for a set of Horn clauses. The authors apply their method to the Needham-Schroeder protocol, constructing Horn clauses from C

code for one role; the clauses for the other role are supplied via the external trust model. They focus on secrecy and do not handle authentication.

Udrea et al. [46] develop PISTACHIO, a tool for checking that implementations of protocols such as SSH conform to rule-based specifications capturing the protocol description in the RFC. The method and tool is successful in identifying a number of low level bugs in protocol implementations. There are several differences between this work and the approach reported in this paper. First, our analysis seeks to provide assurance that protocol implementations in C provide security properties even in the presence of an attacker, whereas [46] focuses on ensuring that the implementation conforms to the RFC specification and does not explicitly model an attacker. Second, they use static analysis methods while our approach is based on software model-checking.

Automated abstraction refinement for software was pioneered by the SLAM project [5]. It has since become an area of intense research, leading to the development of a number of model checking tools such as BLAST [28]. Simulation conformance of OpenSSL code to the SSL RFC specification has been verified previously by the MAGIC tool [12]. However, that work did not incorporate either an attacker model or the interactions between cryptographic and numeric operations. Other software model checking tools such as CMC [39] and VERISOFT [22] have also been used to verify network protocol implementations. The work reported in this paper differs from this line of prior work because of the inclusion of a formal and explicit attacker in the abstract model extracted from the code.

9 Conclusion and Future Work

In this paper, we present the SPIER framework for automated analysis of security protocol implementations. We have implemented SPIER and used it to verify 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 a general purpose theorem prover and deconstruction followed by construction) and present their comparison in the context of OpenSSL verification.

In future work, we plan to develop support for additional C features, such as pointers and bit-wise operations. We also plan to investigate how orthogonal approaches for analyzing control flow properties [3] can be used and extended to discharge the assumption of control flow integrity in this work. Another direction for future work is to build on results from analysis of cryptographic libraries [17] to justify the soundness of the library routine specifications.

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 and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.
3. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, pages 340–353, 2005.
4. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.
5. Thomas Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In Matthew 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.
6. Thomas Ball and Sriram K. Rajamani. Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical report MSR-TR-2002-09, Microsoft Research, Redmond, Washington, USA, January 2002.
7. David A. Basin, Sebastian Mödersheim, and Luca Viganò. Ofmc: A symbolic model checker for security protocols. *Int. J. Inf. Sec.*, 4(3):181–208, 2005.
8. Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffei. Refinement types for secure implementations. In *CSF*, pages 17–32, 2008.
9. Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *CSFW*, pages 139–152, 2006.
10. Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society, 2001.
11. I. Cervesato, A.D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key kerberos. In *Proceedings of the 11th Annual Asian Computing Science Conference*, 2006.
12. Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.
13. Sagar Chaki, James Ivers, Natasha Sharygina, and Kurt Wallnau. The ComFoRT Reasoning Framework. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 164–169, Edinburgh, Scotland, July 6–10, 2005. New York, NY, July 2005. Springer-Verlag.
14. Edmund Clarke, Orna Grumberg, and David Long. Model Checking and Abstraction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*, pages 342–354, Albuquerque, NM, January 19–22, 1992. New York, NY, January 1992. Association for Computing Machinery.
15. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, Chicago, IL, July 15–19, 2000. Berlin, Germany, July 2000. Springer-Verlag.
16. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.

17. Galois Connections. Cryptol reference manual, 2005.
18. Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol Composition Logic (PCL). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.
19. T. Dierks and C. Allen. The TLS protocol version 1.0, 1999. RFC 2246.
20. D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.
21. Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0, 1996. Internet Draft.
22. Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.
23. Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
24. Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI*, pages 363–379, 2005.
25. Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 22–25, 1997. New York, NY, June 1997. Springer-Verlag.
26. Changhua He and John C. Mitchell. Security analysis and improvements for IEEE 802.11i. In *Proc. NDSS*, 2005.
27. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
28. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, volume 37(1) of *SIGPLAN Notices*, pages 58–70, Portland, OR, January 16–18, 2002. New York, NY, January 2002. Association for Computing Machinery.
29. S. Kent and R. Atkinson. Security architecture for the internet protocol, 1998. RFC 2401.
30. J.T. Kohl and B.C. Neuman. The Kerberos network authentication service (version 5). IETF RFC 1510, September 1993.
31. Patrick D. Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. Probabilistic Polynomial-Time Equivalence and Security Protocols. In *Formal Methods World Congress, vol. I*, number 1708 in *LNCS*, pages 776–793, 1999.
32. G. Lowe. Some new attacks upon security protocols. In *Proc. CSFW*, pages 162–169. IEEE, 1996.
33. C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
34. 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.
35. Catherine Meadows and Dusko Pavlovic. Deriving, attacking and defending the GDOI protocol. In *Proc. ESORICS 2004*, volume 3193 of *LNCS*, pages 53–72, 2004.
36. Robin Milner. *Communication and Concurrency*. Prentice-Hall International, London, 1989.
37. J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In *Proceedings of the Seventh USENIX Security Symposium*, pages 201–216, 1998.

38. John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using mur-phi. In *IEEE Symposium on Security and Privacy*, pages 141–151, 1997.
39. Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *OSDI*, 2002.
40. R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
41. OpenSSL website. <http://www.openssl.org>.
42. L.C. Paulson. Proving Properties of Security Protocols by Induction. In *Proc. CSFW*, pages 70–83, 1997.
43. Michaël Rusinowitch and Mathieu Turuani. Protocol Insecurity with Finite Number of Sessions is NP-Complete. In *CSFW*, pages 174–, 2001.
44. Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols*. Addison-Wesley, 2001.
45. D. Song. Athena: a New Efficient Automatic Checker for Security Protocol Analysis. In *Proc. CSFW*, pages 192–202. IEEE, 1999.
46. Octavian Udea, Cristian Lumezanu, and Jeffrey S. Foster. Rule-based static analysis of network protocol implementations. In *Usenix Security*, pages 193–208, 2006.
47. David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the 7th Annual Network and Distributed System Security Symposium (NDSS '00)*, San Diego, CA, October 31–November 5, 2004. Reston, VA, 2000. Internet Society.