ASPIER: An Automated Framework for Verifying Security Protocol Implementations

Sagar Chaki
Software Engineering Institute
chaki@sei.cmu.edu

Anupam Datta
Carnegie Mellon University
danupam@cmu.edu

***Abstract*—We present ASPIER – the first framework that combines software model checking with a standard protocol security model to automatically analyze authentication and secrecy properties of protocol implementations in C. The technical approach extends the iterative abstraction-refinement methodology for software model checking with a domain-specific protocol and symbolic attacker model. We have implemented the ASPIER tool and used it to verify authentication and secrecy properties of a part of an industrial strength protocol implementation – the handshake in 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 evaluated them in the context of OpenSSL verification. ASPIER detected the "version-rollback" vulnerability in OpenSSL 0.9.6c source code and successfully verified the implementation when clients and servers are only willing to run SSL 3.0.**

***Keywords*-security protocol; verification; model checking; abstraction refinement.**

I. Introduction

Network protocols such as SSL [1], TLS [2], Kerberos [3], IPSec [4], and IEEE 802.11i [5] 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 [6], [7], [8], [9], [10]. 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* [11], [12], [13], [14], [15], [16], [6], [17], [18], [19]. Independently, in recent years, there has been significant progress in automatically verifying non-trivial properties of software *implementations*. In this context, one successful technique is *software model checking* – a combination of predicate abstraction [20] and model checking [21] with automated abstraction refinement [22], [23].

In this paper, we present ASPIER[1]—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 [24] 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. We also discuss our underlying assumptions.

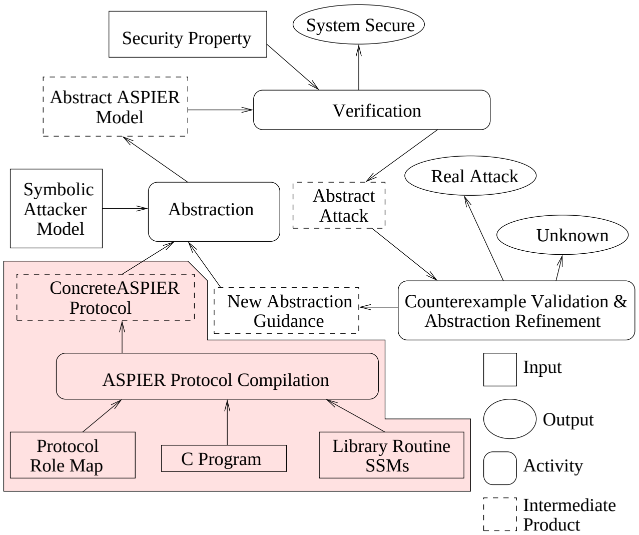


Figure 1. Overall security verification framework. SSMs = specification state machines. Shaded area represents the protocol compilation stage.

Method: The ASPIER framework is depicted in Figure 1, and consists of two main stages – protocol compilation and verification.

ASPIER PROTOCOL COMPILATION: We begin with the C programs for the protocol and translate them into programs in the ASPIER protocol language (presented in Section II). Thus, a *concrete* ASPIER protocol is defined by a set of programs, one for each protocol role. Protocol compilation 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

[1] ASPIER is an acronym for "Automated Security Protocol Implementation verifiER"

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 protocol implementations involve two kinds of operations – numeric and cryptographic. The ASPIER 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 involving numeric operations are executed by an agent based on the value (e.g., version number received) of a message variable. However, direct assignments of numeric expressions to message variables, and message terms to numeric variables, are disallowed. We believe that this is a reasonable restriction since messages are constructed by calls to library routines for encryption, signature generation etc. Technically, this choice ensures that the term algebra abstraction for messages is preserved; the capabilities of the symbolic attacker can then be defined in terms of derivability of terms in the algebra. Without this restriction, a realistic attacker is able to operate arbitrarily on numbers (bits), and formal analysis becomes intractable. This design choice thus reflects tradeoffs among expressivity of the language, and tractability of ASPIER. The separation is also explicit in the operational semantics of the ASPIER programming language, where cryptographic state is maintained using a *context* and the *attacker knowledge*, while numeric state is captured by a *store* (see Section II-C and Figure 6 for details).

VERIFICATION: Once the C programs are translated into a concrete ASPIER protocol, we verify that they satisfy the desired security properties. However, this cannot be achieved via direct reachability analysis since the concrete protocol is an infinite state system. We therefore use a combination of predicate abstraction [20] with counterexample guided abstraction refinement (CEGAR) [22]. Predicate abstraction represents an infinite set of memory configurations with a valuation to a set of predicates involving program variables (e.g., $x > 0$), and yields a finite abstract model of the concrete protocol. CEGAR refines the abstract model iteratively to a level of detail needed for proving the security property, following the steps below:

1. Abstraction: This step automatically extracts an *abstract* ASPIER model M of the concrete protocol via ASPIER's predicate abstraction. The main challenge that we address involved developing a uniform *finite* model that combines predicate abstraction for C-style code, protocol actions, and a *symbolic attacker* model [25], [26]. Technically, this involves abstracting the numeric state via predicates over the store variables, retaining the concrete cryptographic state, and handling unbounded depth attacker computations via automated deduction techniques (cf. [27], [28]). The abstraction yields a finite model because: (i) numeric unboundedness is handled via predicate abstraction, and (ii) cryptographic unboundedness is handled by the term abstraction, and the requirement that only bounded messages are exchanged over the network. Our main technical result (Theorem 1) is that this abstraction is sound. Specifically, for a security property φ, if M satisfies φ, then the concrete system also satisfies φ. Section III explains this step in detail.

2. Verification: Our verification approach combines model checking with theorem proving. 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 V 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. This step yields one of three possible results. First, if we determine that CE is a real counterexample, we exit with "Real Attack" and a concretization of CE. Second, if we find that CE does not concretize to any real attack, i.e., CE is spurious, we proceed to Step 4. Finally, if we are unable to determine whether CE concretizes to a real attack or not (this is possible since the counterexample validation problem is undecidable), 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 IV 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 IV explains this step in detail.

Tool: We implemented the ASPIER framework by extending the COPPER tool [29]. Specifically, we augmented each stage of COPPER's CEGAR engine to support ASPIER's concrete and abstract model. We also implemented two mechanisms for handling unbounded attacker computations via automated deduction techniques. The ASPIER implementation is presented in Section V.

Application: We used ASPIER to establish authentication and secrecy properties of the *OpenSSL* [24] implementation of the SSL handshake protocol. When clients and servers implement only SSL 3.0, ASPIER verifies authentication and secrecy properties of OpenSSL for configurations comprising of up to 3 servers and 3 clients. When clients and servers implement both SSL 2.0 and SSL 3.0 and negotiate the version during handshake, ASPIER 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. We observe that CEGAR works well for this application because predicate abstraction over OpenSSL's branch conditions exposes a sufficiently precise control flow skeleton. Further details about our OpenSSL case study are presented in Section VI.

Assumptions and Limitations: The current ASPIER implementation treats floating point data as integers, and bitwise 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 [23] or BLAST). Others, such as the proper treatment of floating points during software analysis, are open problems. We note that in the context of our OpenSSL case study, these limitations were relevant only within the bodies of library routines replaced by SSMs. Thus, they were handled via our assumption that the SSMs used as input to ASPIER are sound abstractions of their corresponding library routines.

We also assume that the control flow graph accurately represents the possible executions of the program. We believe that this assumption is dischargable via orthogonal techniques (for example, Control Flow Integrity [30]). Detecting low-level security issues, such as buffer overflows, also require other program analysis methods [31]. Inspite of these restrictions, we found ASPIER to be useful in reasoning about an industrial strength protocol implementation.

Running Example: We use ASPIER to verify authentication and secrecy properties of the OpenSSL 0.9.6c implementation of SSL 3.0 handshake. Figures 2(a) and 2(b) show *simplified* versions of the OpenSSL client and server code respectively. Figure 2(c) shows the CFG extracted from the client code. 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. We use the program in Figure 2 as a running example to illustrate our approach. 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.

II. Concrete Protocol Model and Properties

In this section, we describe the syntax and semantics (i.e., concrete model) of the ASPIER protocol language. The language supports cryptographic operations, communication actions, assignments, conditionals, and loops. We also define authentication and secrecy in the concrete model.

A. 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:

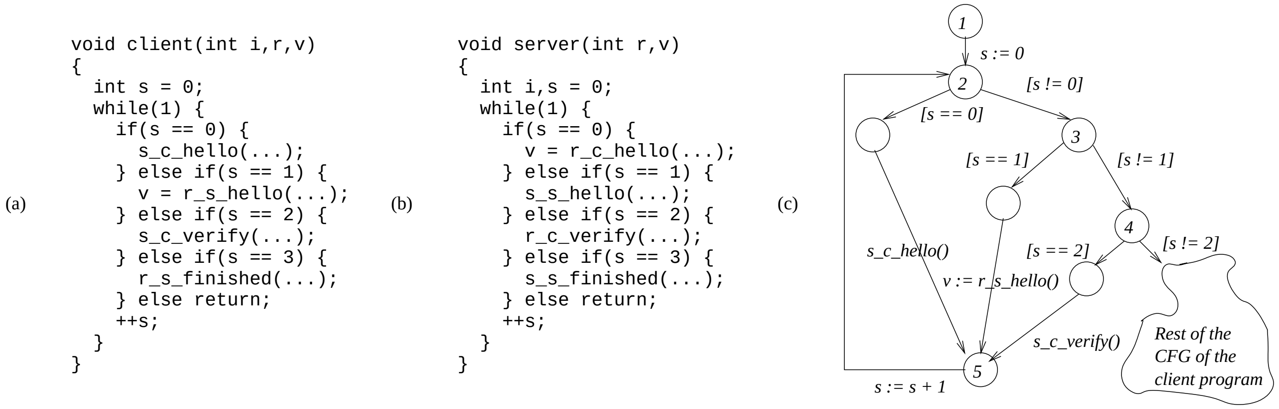
$$Key = K \mid privkey(Name) \mid pubkey(Name)$$

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.



(a)

```
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;
  }
}
```

(b)

```
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;
  }
}
```

Figure 2. An implementation of a two-party signature-based challenge-response protocol: (a) client code; (b) server code; (c) client CFG.

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:

$$Act = \texttt{new}\ MVar \mid \texttt{send}\ Term \mid \texttt{recv}\ Term$$
$$\mid MVar := Msg$$

where: (a) $\texttt{new}\ v$ denotes creating a fresh nonce and storing it in v, (b) $\texttt{send}\ t$ denotes sending a message, (c) $\texttt{recv}\ t$ denotes receiving a message and matching it to t, and (d) $v := m$ denotes assigning m to v. 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:

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

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:

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

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 $\nu[t] = \bot$).

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 instantiated by the thread.

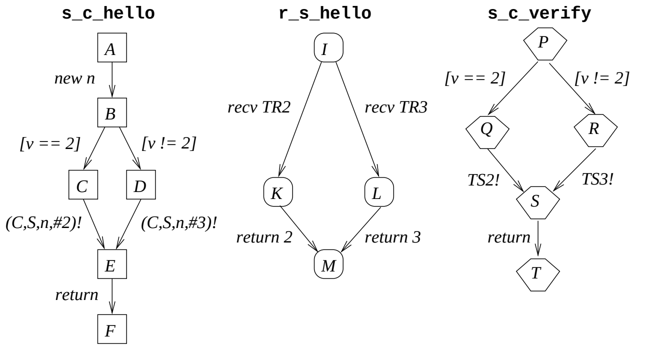


Figure 3. SSMs for library routines called by `client` procedure from Fig. 2(a).

Example 1: Recall that our running example protocol consists of two roles: client and server. Figure 3 shows the SSMs for some of the library routines called by `client`. The control flow graph (CFG) of the client role is derived via *specification inlining* from the `client` CFG shown in Figure 2(c). Figure 4 shows a fragment of the resulting client role CFG. Each client role CFG node shares its shape and name with its corresponding SSM state.

Figure 4 uses the following convention: (i) $\texttt{[}e\texttt{]} \equiv \texttt{assume}\ e$; (ii) $t\texttt{!} \equiv \texttt{send}\ t$; (iii) $\#n \equiv$ message constant n; (iv) $\{t\}t' \equiv \{t\}_{t'}$; (v) $(t_1, \dots, t_n) \equiv (t_1, (\dots(t_{n-1}, t_n)\dots)$; (vi) S is the server's name; (vii) C is the client's name; (viii) CA

is the certifying authority's name; (ix) n, n' are nonces; (x) Sec is the client's secret; (xi) $TR2 \equiv (S, C, n', \#2, Sig(privkey(CA), (S, pubkey(S))))$; (xii) $TR3 \equiv (S, C, n', \#3, Sig(privkey(CA), (S, pubkey(S))))$; (xiii) $TS2 \equiv (C, S, Sig(privkey(CA), (C, pubkey(C))),$
$Sig(privkey(C), HS_2), \{Sec\}_{pubkey(S)}, Hash_{Sec}(HS_2))$
where $HS_2 \equiv$ all previous messages sent and received by the client **excluding** the version number; (xiv) $TS3 \equiv (C, S, Sig(privkey(CA), (C, pubkey(C))),$
$Sig(privkey(C), HS_3), \{Sec\}_{pubkey(S)}, Hash_{Sec}(HS_3))$
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 VI.

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) $Id = (\eta, \mathtt{C})$, (ii) $\nu = \{i = \mathtt{C}, r = \mathtt{S}\}$, (iii) $S = \{1, 2, \dots\}$, (iv) $I = 1$, and (v) T is as shown in Figure 4.

B. 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 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 Figure 2(c)).

Attacker Model. We use the standard symbolic (Dolev-Yao) attacker model [26], [25]. 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 use Γ (with decorations) to denote the attacker's knowledge.

$$\Gamma \vdash m \wedge \Gamma \vdash k \implies \Gamma \vdash \{m\}_k$$

$$\Gamma \vdash \{m\}_k \wedge \Gamma \vdash k^{-1} \implies \Gamma \vdash m$$

$$\Gamma \vdash m \wedge \Gamma \vdash k' \implies \Gamma \vdash Sig(k', m)$$

$$\Gamma \vdash m \wedge \Gamma \vdash k \implies \Gamma \vdash Hash_k(m)$$

$$\Gamma \vdash m_1 \wedge \Gamma \vdash m_2 \implies \Gamma \vdash (m_1, m_2)$$

$$\Gamma \vdash (m_1, m_2) \implies \Gamma \vdash m_1 \wedge \Gamma \vdash m_2$$

Figure 5. Attacker message derivation rules; $k' = privkey(N)$ for name N.

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 during a message receive (which is used to model operations such as projection, decryption and signature verification) uses a function $match$. Specifically, $match : Msg \times Term \hookrightarrow \mathcal{C}$ 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 $\perp$.

In subsequent sections, we present the concrete and abstract semantics of ASPIER protocols. However, we note here that, since we combine message receives with pattern matches, the non-determinism due to a message receive in both the concrete and abstract semantics of ASPIER protocols is always finite. This holds even when the attacker is able to construct an infinite number of possible messages. The finite non-determinism property is crucial to ensure that the abstract model of any ASPIER protocol is finite-state, which in turn is necessary for model checking.

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:

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

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

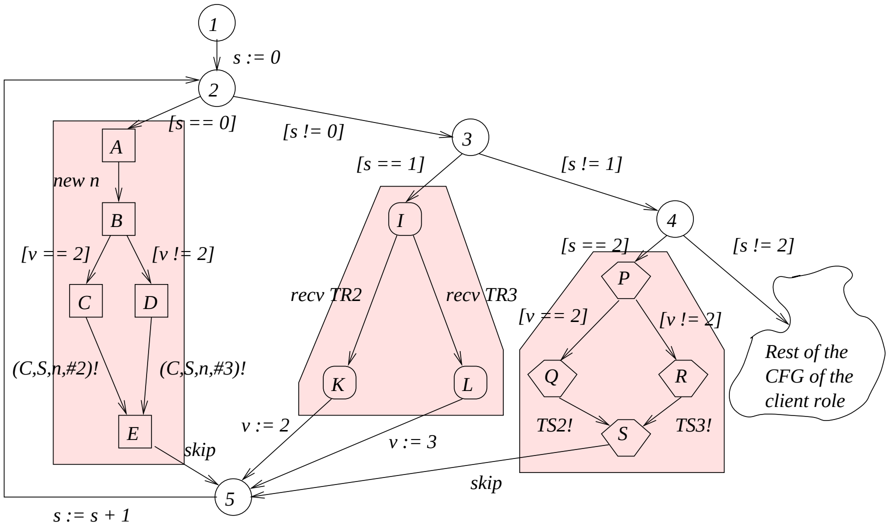


Figure 4. Client role derived via specification inlining using CFG in Fig. 2(c), and library routine SSMs from Figure 3. Shaded parts are derived from SSMs.

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

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

$$\mathcal{R}^{Msg}_{\texttt{recv}\ t} = \{(\nu,\Gamma,\nu',\Gamma') \mid \nu' = \nu \bowtie match(m,t) \wedge \Gamma \vdash m \wedge \Gamma' = \Gamma\}$$

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

$$\mathcal{R}^{\mathcal{S}}_{v:=e} = \{(\sigma,\sigma') \mid \sigma' = \sigma \bowtie \{v = \sigma[e]\}\} \qquad \mathcal{R}^{\mathcal{S}}_{\texttt{assume}e} = \{(\sigma,\sigma) \mid \sigma[e] \neq 0\}$$

Figure 6. Transformer rules for statements.

$\mathcal{S} \times \mathcal{C} \times 2^{Msg}$ 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. 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 $\mathcal{R}^{Msg}_{St} \subseteq \mathcal{C} \times 2^{Msg} \times \mathcal{C} \times 2^{Msg}$ is shown in Figure 6. For any non-action statement St, $\mathcal{R}^{Msg}_{St}$ is the identity relation.
2) For any non-action statement St, the transformer relation $\mathcal{R}^{\mathcal{S}}_{St} \subseteq \mathcal{S} \times \mathcal{S}$ is shown in Figure 6. For any action statement St, $\mathcal{R}^{\mathcal{S}}_{St}$ 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:

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

C. Protocol Language Concrete Semantics

We defined protocols and threads in Section II-A. 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 $\mathcal{S} \times \{\nu\} \times \{\Gamma_0\}$ where $\Gamma_0 \subseteq Msg$ 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:

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

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, i.e. the secrecy of m has been violated. This is a special transition used to specify secrecy.

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 $E_I = \mathcal{S} \times \{\nu_1 \bowtie \nu_2\} \times \{\Gamma_0\}$. 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 $(s_i = s_i')$.

D. Security Properties and Satisfaction

For any sequence w and any set u of events, we write $w \downharpoonright u$ to mean the subsequence of w obtained by eliding events not in u. For instance, $\langle \alpha,\gamma,\beta,\beta,\gamma,\beta,\alpha \rangle \downharpoonright \{\alpha,\gamma\} = \langle \alpha,\gamma,\gamma,\alpha \rangle$. 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) $\forall 1 \le i \le k \centerdot (q_{i-1}, b_i, q_i) \in \mathbb{T}$, (iii) $b_k = \alpha_n$, and (iv) $\langle b_1, \dots, b_k \rangle \downharpoonright \{\alpha_1, \dots, \alpha_n\} \neq \langle \alpha_1, \dots, \alpha_n \rangle$.

For 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 thread identifier of the client role, Id_2 is the thread identifier of 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) $\forall 1 \le i \le k \centerdot (q_{i-1}, b_i, q_i) \in \mathbb{T}$, and (iii) $b_k = m\#$. For OpenSSL, m is the secret generated by the client role.

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.

III. Abstract 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, ASPIER 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 model[2].

A. Abstract Protocol Model

We begin with 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:

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

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

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

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

Abstract Transformer. In predicate abstraction [20], every statement is viewed as a predicate valuation transformer. Specifically, for any statement St and predicate set $\mathcal{P}$, the transformer relation $\mathcal{R}^{\mathcal{V}}_{St,\mathcal{P}} \subseteq \mathcal{V}_{\mathcal{P}} \times \mathcal{V}_{\mathcal{P}}$ is defined as follows:

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

Intuitively, V is related to V' by the abstract transformer $\mathcal{R}^{\mathcal{V}}_{St,\mathcal{P}}$ if there are stores σ and σ' such that $V = V(\sigma)$, $V' = V(\sigma')$, 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:

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

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 $\mathcal{R}^{\mathcal{V}}_{St,\mathcal{P}}$ 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:

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

Note that $\mathcal{R}_{St,\mathcal{P}}$ is computable since both $\mathcal{R}^{\mathcal{V}}_{St,\mathcal{P}}$ and $\mathcal{R}^{Msg}_{St}$ 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:

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

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 $(\sigma, \sigma') \in \mathcal{R}^{\mathcal{S}}_{St}$ and $(\nu, \Gamma, \nu', \Gamma') \in \mathcal{R}^{Msg}_{St}$. Then, by Fact 1, we know that $(V_{\mathcal{P}}(\sigma), V_{\mathcal{P}}(\sigma')) \in \mathcal{R}^{\mathcal{V}}_{St,\mathcal{P}}$. 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. ■

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 $\mathcal{V}_{\mathcal{P}} \times \{\nu\} \times \{\Gamma_0\}$ where $\Gamma_0 \subseteq Msg$ 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:

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

Example 2: Recall, from Figure 4, our example client thread $\mathcal{T} = (Id, \nu, S, I, T)$ where: (i) $Id = (\eta, \mathsf{C})$, (ii) $\nu = \{i = \mathsf{C}, r = \mathsf{S}\}$, (iii) $S = \{1, 2, \ldots\}$, (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 $p_0 \equiv \texttt{(s == 0)}, p_1 \equiv \texttt{(s == 1)}, p_2 \equiv \texttt{(s == 2)}$ and $v_2 \equiv \texttt{(v == 2)}$. 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 $\widehat{E_I} = \mathcal{V}_{\mathcal{P}} \times \{\nu_1 \bowtie \nu_2\} \times \{\Gamma_0\}$. 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 $(s_i = s'_i)$.

B. 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 [32] by an LTS $M_2 = (\mathbb{S}_2, \mathbb{I}_2, \Sigma, \mathbb{T}_2)$ (written as

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

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

$M_1 \preceq M_2$) iff there exists a relation $R \subseteq \mathbb{S}_1 \times \mathbb{S}_2$ such that the following two conditions hold:

$$\forall s_1 \in \mathbb{I}_1 \centerdot \exists s_2 \in \mathbb{I}_2 \centerdot s_1 \ R \ s_2 \qquad \textbf{(INIT)}$$

$$\forall s_1, s_1' \in \mathbb{S}_1 \centerdot \forall s_2 \in \mathbb{S}_2 \centerdot \forall a \in \Sigma \centerdot s_1 \ R \ s_2 \wedge (s_1, a, s_1') \in \mathbb{T}_1 \implies$$
$$\exists s_2' \in \mathbb{S}_2 \centerdot (s_2, a, s_2') \in \mathbb{T}_2 \wedge s_1' \ R \ s_2' \qquad \textbf{(STEP)}$$

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:

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

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 $((s_1', s_2', E'), (s_1', s_2', \widehat{E'})) \in R$ ■

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 $CE = \langle q_0, \alpha_1, q_1, \ldots, \alpha_k, q_k \rangle$ 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, $\langle (s_{01}, s_{02}, \widehat{E_0}), \alpha_1, \ldots, \alpha_k, (s_{k1}, s_{k2}, \widehat{E_k}) \rangle$ is a counterexample to $M(\mathcal{T}_1, \mathcal{T}_2, \mathcal{P}) \models \varphi$. ■

Note that, in general, simulation preserves all ACTL* properties [33]. Thus, our abstraction preserves not only safety properties (which includes authentication and secrecy), but also non-safety properties (such as non-interference [34]) 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.

IV. Counterexample Validation and Refinement

Given a counterexample CE on the abstract model, counterexample validation determines whether CE is a real counterexample, i.e., it concretizes to a real attack. This results in one of three possible outcomes:

1) If we determine that CE is a real counterexample, ASPIER exits with "Real Attack" and a concretization of CE exhibiting a real attack.
2) If we find that CE does not concretize to any real attack, i.e., CE is spurious, we proceed to the refinement phase.
3) Finally, if we are unable to determine whether CE concretizes to a real attack or not (this is possible since the counterexample validation problem is undecidable), we exit with "Unknown".

In the refinement phase, we use the spurious CE to update the abstraction information and repeat the CEGAR loop. The main property ensured by the refinement process is that CE does not arise as a counterexample in the refined model. For completeness, we now provide technical details of the counterexample validation and refinement procedures. Readers who are not interested in these details may skip to the next section without loss of continuity.

Counterexample Validation. Let the counterexample be a sequence of abstract states $CE = \langle \widehat{q}_0, \alpha_1, \widehat{q}_1, \ldots, \alpha_k, \widehat{q}_k \rangle$. 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 $q_i, \ldots, q_k$ of $M(\mathcal{T}_1, \mathcal{T}_2)$, and a sequence of stores $\sigma_i, \ldots, \sigma_k$ such that: (i) for $i \leq j \leq k$,

$q_j = (s^1_j, s^2_j, (\sigma_j, \nu_j, \Gamma_j))$, (ii) for $i \le j < k$, $(q_i, \alpha_i, q_{i+1}) \in \mathbb{T}$, 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 $\widehat{q}_i, \ldots, \widehat{q}_k$ of CE starting with the store σ. Finally, CE is valid iff $\exists \sigma \in \mathcal{S} \centerdot \sigma \models CE(0)$.

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

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

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.

Refinement. We refine our abstraction by changing our set of predicates to $\mathcal{P}' = \mathcal{P} \cup \{VC_0, \ldots, VC_k\}$. 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 $CE' = \langle \widehat{q'}_0, \alpha_1, \widehat{q'}_1, \ldots, \alpha_k, \widehat{q'}_k \rangle$ such that $\widehat{q'}_i = (s^1_i, s^2_i, (V'_i, \nu'_i, \Gamma'_i))$ for $0 \le i \le k$.

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 $0 \le i < k \centerdot V'_i(VC_i) = V'_{i+1}(VC_{i+1})$. 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 $(\widehat{q'}_{k-1}, \alpha_k, \widehat{q'}_k)$ is not possible, which is a contradiction.

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.

V. Tool Implementation

We implemented ASPIER by extending the COPPER [29] tool. 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, highlighting specific augmentations to COPPER involved.

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. To implement this step, we augmented COPPER's machinery for representing and inlining library routine SSMs so that the transitions in these SSMs are labeled with cryptographic terms instead of uninterpreted symbols.

Abstraction. The abstract model is then computed by using the definitions and concepts for constructing abstract thread composition presented in Section III. The key change to COPPER required in this step was the combination of the two abstract transformers – $\mathcal{R}^{\mathcal{V}}_{St,\mathcal{P}}$ and $\mathcal{R}^{Msg}_{St}$. As originally implemented in COPPER, the Simplify theorem prover is used to compute $\mathcal{R}^{\mathcal{V}}_{St,\mathcal{P}}$. To compute $\mathcal{R}^{Msg}_{St}$, we 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 [27]. 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, $\Gamma \vdash m \iff m \in \Gamma''$. Note that **Decons-Cons** is a valid decision procedure for ASPIER since we only use atomic keys. Experimental results comparing between **Simplify** and **Decons-Cons** are presented in Section VI. For simplicity, our technical presentation of predicate abstraction was based on a global set of predicates. In practice, ASPIER inherits COPPER's use of *predicate localization* [35] for efficiency, i.e., it employs different sets of predicates at different CFG nodes.

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 heavily by redundant work. As our experimental results indicate, this strategy appears to be a good foil to the statespace explosion problem in practice. We believe that symbolic reachability algorithms – used by tools like OFMC [28] – have the potential of making this verification step even more efficient.

Counterexample Validation and Refinement. These two steps in ASPIER augment the corresponding COPPER stages with protocol specific concepts. In particular, these changes are driven by the fact that transitions in the library routine SSMs, and abstract models, are labeled by cryptographic terms instead of uninterpreted symbols. 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 [36] 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 [37], which are also applicable.

VI. OpenSSL Case Study

We experimented with the OpenSSL 0.9.6c implementation of the SSL handshake using a 2.4 GHz machine with 4 GB of RAM.

Benchmarks. We obtained our benchmarks by:

1) Extracting parts of OpenSSL that deal with the core handshake implementation. These were files named `s3_clnt.c` and `s3_srvr.c` for the client and server respectively.
2) Manually eliding code fragments that deal with logging, reconnection, etc., while retaining code that implements a first time handshake. The final preprocessed C files that we analyzed consisted of about 1200 LOC for each server and each client.
3) Manually creating SSMs for each library routine called by the top-level functions – `ssl3_accept` for the server and `ssl3_connect` for the client. We derived the behavior of each such routine via manual code and document inspection, and created SSMs that soundly abstract the behavior of their corresponding routines.

Task 1. Detecting Version-Rollback Attack. We confirmed the "version-rollback" attack in OpenSSL when it allows version negotiation during handshake. Since the version is not adequately integrity protected, the attacker forces a client and a server to use SSL 2.0 even when both intend to use SSL 3.0. Let *OpenSSL23* be the OpenSSL allowing version negotiation, and *OpenSSL3* be the OpenSSL hardwired to version 3.0. We validated – for configurations of up to 2 servers and 2 clients – that version-rollback is absent with either *OpenSSL3* clients or servers. This is because integrity protection of the version by even one party prevents version-rollback. We also confirmed – for configurations of up to 2 servers and 2 clients – that when both roles implement *OpenSSL23*, version-rollback exists.

Our results are tabulated in Figure 8. For configurations with one server and one client, the cases without rollback completed very quickly, while the cases with rollback required more time and memory. This is 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.

Task 2. Verifying 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

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

Figure 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

Figure 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).

input parameters (modulo symmetry) of the client roles. We started each experiment with an initial set of predicates, derived manually from the branch conditions in the C code and the property being verified. We believe that, eventually, these predicates would be inferred by the abstraction refinement process. Therefore, starting with them simply speeds up the verification. The selected initial 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**.

VII. Related Work

We summarize below some complementary approaches reported in recent work. Goubault-Larrecq and Parrennes [38] use Horn clauses to represent the intruder deduction relation, and a model derived from C protocol implementations via inter-procedural analysis. Secrecy properties are expressed as satisfiability problems for a set of Horn clauses. The method is applied to the Needham-Schroeder protocol, where Horn clauses are constructed for one role; clauses for other roles are supplied via an external trust model. They handle secrecy, but not authentication.

Bhargavan et al. [39] 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 [40], an automated tool, for verifying the resulting translation. They incorporate a symbolic attacker, and in recent work [41], extend this approach to the computational model. Their security results hold for an unbounded number of concurrent sessions. Their approach does not apply to C code. In contrast, we verify C implementations with bounded sessions using CEGAR. In a related effort, Bengtson et al. use a type system to verify authentication properties of protocols written in F# [42].

Udrea et al. [43] develop PISTACHIO, a static analysis tool for checking that implementations of protocols such as SSH conform to rule-based specifications capturing the protocol description in the RFC. They identify several bugs in protocol implementations. They focus on ensuring that the implementation conforms to the RFC specification and do not explicitly model an attacker.

A number of projects [23], [35] have explored CEGAR for verifying software correctness. Software model checking tools [44], [45], [46] have also been used to verify network protocol implementations. However, our work is distinguished by the inclusion of an explicit attacker model.

VIII. Conclusion and Future Work

The ASPIER framework described in this paper makes progress towards automated analysis of security protocol implementations. We identify three areas for future work: (a) improved support for a richer class of C features, such as pointers and bit-wise operations; (b) discharging the control flow integrity assumption; and (c) justifying the soundness of library routine specifications. To this end, we plan to leverage progress in integrating pointer analysis [23] and bitwise semantics [47] with software model checkers, runtime enforcement of control flow integrity [30], and analysis of cryptographic libraries [48].

References

[1] A. O. Freier, P. Karlton, and P. C. Kocher, "The SSL protocol version 3.0," 1996, Internet Draft.

[2] T. Dierks and C. Allen, "The TLS protocol version 1.0," 1999, RFC 2246.

[3] J. Kohl and B. Neuman, "The Kerberos network authentication service (version 5)," IETF RFC 1510, 1993.

[4] S. Kent and R. Atkinson, “Security architecture for the internet protocol,” 1998, RFC 2401.

[5] “IEEE Standard 802.11-1999. Local and metropolitan area networks - specific requirements - part 11: Wireless LAN Medium Access Control and Physical Layer specifications.” 2004.

[6] G. Lowe, “Some new attacks upon security protocols,” in *Proceedings of the 9th IEEE Computer Security Foundations Workshop (CSFW ’96)*, 1996, pp. 162–169.

[7] C. Meadows, “Analysis of the Internet Key Exchange protocol using the NRL Protocol Analyzer,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 1998, pp. 216–231.

[8] C. He and J. C. Mitchell, “Security analysis and improvements for IEEE 802.11i.” in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS ’05)*, 2005.

[9] C. Meadows and D. Pavlovic, “Deriving, attacking and defending the GDOI protocol.” in *Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS ’04)*, 2004, pp. 53–72.

[10] 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.

[11] M. Abadi and A. Gordon, “A calculus for cryptographic protocols: the spi calculus,” *Information and Computation*, 1999.

[12] M. Abadi and C. Fournet, “Mobile values, new names, and secure communication,” in *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL ’01)*, 2001, pp. 104–115.

[13] 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*, no. 1708, 1999.

[14] C. Meadows, “The NRL protocol analyzer: An overview,” *Journal of Logic Programming*, vol. 26, no. 2, 1996.

[15] D. Song, “Athena: a New Efficient Automatic Checker for Security Protocol Analysis,” in *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW ’99)*, 1999, pp. 192–202.

[16] L. Paulson, “Proving Properties of Security Protocols by Induction,” in *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW ’97)*, 1997, pp. 70–83.

[17] J. C. Mitchell, M. Mitchell, and U. Stern, “Automated analysis of cryptographic protocols using Mur-phi.” in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 141–151.

[18] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe, *Modelling and Analysis of Security Protocols*. Addison-Wesley, 2001.

[19] A. Datta, A. Derek, J. C. Mitchell, and A. Roy, “Protocol Composition Logic (PCL),” *Electronic Notes in Theoretical Computer Science*, pp. 311–358, 2007.

[20] S. Graf and H. Saïdi, “Construction of Abstract State Graphs with PVS,” in *Proceedings of the 9th International Conference on Computer Aided Verification (CAV ’97)*, vol. 1254. Springer-Verlag, 1997, pp. 72–83.

[21] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2000.

[22] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-Guided Abstraction Refinement,” in *Proceedings of the 12th International Conference on Computer Aided Verification (CAV ’00)*, vol. 1855. Springer-Verlag, 2000, pp. 154–169.

[23] T. Ball and S. K. Rajamani, “Automatically Validating Temporal Safety Properties of Interfaces,” in *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN ’01)*, vol. 2057. Springer-Verlag, 2001, pp. 103–122.

[24] “OpenSSL website,” `http://www.openssl.org`.

[25] R. Needham and M. Schroeder, “Using encryption for authentication in large networks of computers,” *Communications of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[26] D. Dolev and A. Yao, “On the security of public-key protocols,” *IEEE Transactions on Information Theory*, vol. 2, no. 29, 1983.

[27] M. Rusinowitch and M. Turuani, “Protocol Insecurity with Finite Number of Sessions is NP-Complete,” in *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW ’01)*, 2001, pp. 174–.

[28] D. A. Basin, S. Mödersheim, and L. Viganò, “OFMC: A symbolic model checker for security protocols,” *Int. J. Inf. Sec.*, vol. 4, no. 3, pp. 181–208, 2005.

[29] S. Chaki, J. Ivers, N. Sharygina, and K. Wallnau, “The ComFoRT Reasoning Framework,” in *Proceedings of the 17th International Conference on Computer Aided Verification (CAV ’05)*, vol. 3576. Springer-Verlag, 2005, pp. 164–169.

[30] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow Integrity,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2005, pp. 340–353.

[31] D. Wagner, J. S. Foster, E. A. Brewer, and A. 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)*. Internet Society, 2000.

[32] R. Milner, *Communication and Concurrency*. Prentice-Hall International, 1989.

[33] E. Clarke, O. Grumberg, and D. Long, “Model Checking and Abstraction,” in *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’92)*. Association for Computing Machinery, 1992, pp. 342–354.

[34] J. A. Goguen and J. Meseguer, “Security policies and security models,” in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[35] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy Abstraction,” in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL ’02)*, vol. 37(1). Association for Computing Machinery, 2002, pp. 58–70.

[36] T. Ball and S. K. Rajamani, “Generating Abstract Explanations of Spurious Counterexamples in C Programs,” Microsoft Research, Technical report MSR-TR-2002-09, 2002.

[37] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, “Abstractions from Proofs,” in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL ’04)*, 2004, pp. 232–244.

[38] J. Goubault-Larrecq and F. Parrennes, “Cryptographic Protocol Analysis on Real C Code,” in *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI ’05)*, 2005, pp. 363–379.

[39] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, “Verified Interoperable Implementations of Security Protocols,” in *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW ’06)*, 2006, pp. 139–152.

[40] B. Blanchet, “An Efficient Cryptographic Protocol Verifier Based on Prolog Rules,” in *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW ’01)*, 2001, pp. 82–96.

[41] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, “Cryptographically verified implementations for TLS,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2008, pp. 459–468.

[42] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, “Refinement Types for Secure Implementations,” in *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF ’08)*, 2008, pp. 17–32.

[43] O. Udrea, C. Lumezanu, and J. S. Foster, “Rule-Based Static Analysis of Network Protocol Implementations,” in *Usenix Security*, 2006, pp. 193–208.

[44] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, “CMC: A Pragmatic Approach to Model Checking Real Code,” in *Proceedings of OSDI*, 2002.

[45] P. Godefroid, “Model Checking for Programming Languages using Verisoft,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL ’97)*, 1997, pp. 174–186.

[46] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, “Modular Verification of Software Components in C,” *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 6, pp. 388–402, 2004.

[47] “CBMC website,” `http://www.cprover.org/cbmc`.

[48] Galois Connections, “Cryptol Reference Manual,” 2005.