In International Journal of Network Security, 4:1, January 2007, pp. 45-58

Injecting Heterogeneity Through Protocol Randomization

Li Zhuang[1], J. Doug Tygar[2], and Rachna Dhamija[3]
(Corresponding author: Li Zhuang)

Computer Science Division, U. C. Berkeley[1]
Soda Hall #1776, Berkeley, CA 94720, USA (Email: zl@cs.berkeley.edu)
Computer Science Division & SIMS, U. C. Berkeley[2]
Center for Research on Computation and Society, Harvard & SIMS, U. C. Berkeley[3]

(Received Aug. 15, 2005; revised and accepted Sept. 15, 2005)

Abstract

In this paper, we argue that heterogeneity should be an important principle in design and use of cryptographic protocols. We use automated formal analysis tools to randomly generate security protocols as a method of introducing heterogeneity. We present the results of simulations for the case of two party authentication protocols and argue that choosing protocols randomly out of sets numbering in the hundreds of millions is practical and achievable with an acceptable overhead. To realize the simulation, we implemented a highly efficient protocol verifier, achieving approximately two orders of magnitude improvement in performance compared to previous work.

Keywords: Computer security, memory symptom, virus detection

1 Introduction

Can we improve the operational security of protocols by randomizing them? A few years ago, this would have simply been unthinkable, but in recent years, with the development of powerful automated protocol generation systems [5, 15, 21, 23, 24, 26, 37], it has become feasible. Suppose that two parties need to perform an operation, such as authentication. They use the following approach:

Step 1: Two parties agree to use randomized protocol generation

Step 2: They agree on a random seed

Step 3: They then use that seed to generate the same random protocol

Step 4: They execute the generated randomized protocol

This paper explains how each of these steps can be accomplished. We suggest the use of protocols that are chosen at random from a very large set of protocols satisfying specific security and functional properties. By using security protocol analysis tools, we can generate millions of protocols which satisfy the same security specification. Thus, different pairs (or groups) of communicating parties can use different protocols with the same functionality. Even with this broad sketch, concerns immediately rise. Here are two common concerns:

1) In Step 2, one needs to have a way to generate a random seed, and generating a random seed itself requires a seed agreement protocol. Have we realized any advantages? Answer: in most cases choosing a random seed is simpler than other secure protocols, so we still enjoy advantages. (We discuss this at greater length in Section 4.2).

2) If one has a common random seed in Step 2, why not just skip step 3 and execute a standard library protocol in Step 4 – using the common key for encryption? Answer: using a fixed protocol with encryption may still be vulnerable to known problems in the protocol, and moreover the entire exchange will be vulnerable to traffic analysis. (We discuss this at greater length in Section 4.4).

What advantages can randomized protocol generation bring to the table? To understand this, we must step back a little and look at the broader context of security protocols. Security protocols often have flaws in them. These flaws are sometimes inherent in the protocol itself and sometimes arise from operational use of a protocol. For example, the Needham-Schroeder public key authentication protocol [28] has inherent flaws — even in the presence of a correct implementation and correct operator use, an adversary can still disrupt the protocol. Once an attack is discovered for one protocol, all implementations of that protocol become similarly vulnerable. Even worse, "attack scripts", viruses and worms can be written to automatically exploit these vulnerabilities.

Diversity, which plays an important role in ensuring robustness in biological systems, may also have an important role to play in securing a distributed computing environment. To introduce diversity into computer systems, we can create software components that are similar in functional behavior but that contain variations in design or data [9, 20]. This forces an attacker to create a new attack for each software instance.

In this paper, we argue that heterogeneity should be an important principle in design and use of cryptographic protocols. Our first goal is to make it difficult for an attacker to discover or to exploit flaws in security protocols because of the large space of protocols that must be analyzed. Our second goal is to prevent widespread automated attacks on protocols by making any successful attack hard to replicate. Finally, we suggest the use of rapidly changing protocols so that any vulnerability that may exist with a specific protocol will be short-lived.

We explain how to generate and verify heterogeneous protocols that satisfy particular security properties in Section 2. In Section 3, we present the results of a simulation of random protocol generation for two party authentication. In Section 4, we discuss the security benefits and tradeoffs involved in applying heterogeneity to protocol design. Finally, we provide an overview of related work in Section 5 and our conclusions in Section 6.

2 Protocol Generation

In practice, how can we hope to feasibly find a large number of correct and secure protocols that are equivalent in functionality? Clearly, enumerating millions of protocols with a given property is undesirable; not only would the enumeration require excess amounts of computation, but storing all possible protocols would be highly wasteful of space. In this paper, we propose generating random protocols on the fly. We use two-party authentication protocol as an example to illustrate the process. To analyze two-party authentication protocols, we selected *Athena* [30, 31, 36, 37, 38] as our formal verification tool. However, the framework we introduce applies to different types of security protocols and protocol analysis tools. Instead of using a formal analysis tool to find the single most efficient protocol, we are interested in finding a *random* (and fairly efficient) protocol that satisfies a particular property.

In practice, a protocol generator has several logical components. First, we generate pseudo-random protocols (which may or may not satisfy any particular security requirements). We then use a formal analysis tool to automatically test the pseudo-random protocols to see if they are secure with respect to the required security requirements. Finally, if they do satisfy the security requirements, we automatically generate code to realize the protocols.

To appreciate the ideas behind pseudo-random protocol generators, first recall the structure of the familiar pseudo-random number generator. This generator consists of two parts: a sequence generator F and a number generator B. Seeds are chosen from of a set S of potential seeds, called a *seed space*. Our functions satisfy additional properties, namely $F : S \rightarrow S$ and $B : S \rightarrow \{0, \ldots, n-1\}$. We start with an initial seed s_0 and generate a subsequent sequence $s_0, s_1, s_2, \ldots$ using the rule $s_{i+1} = F(s_i)$. We then generate random numbers by applying B to the S values in turn: $B(s_0), B(s_1), B(s_2), \ldots$ to yield a sequence of numbers[1].

Now consider an application with specific requirements on the output. For example, suppose that instead of generating numbers uniformly over $\{0, \ldots, n-1\}$, we wish to uniformly generate *prime numbers* less than n. There are some efficient ways of doing this [22], but for the sake of discussion, consider a brute force approach. We run the pseudo-random number generator and test each number as it is generated. If it is not a prime, we discard it and generate another. The resulting sequence will consist of a uniform pseudo-random sequence of prime numbers. We call the space of acceptable output values, in this case, primes less than n, a *valid output space.*

We can view pseudo-random protocol generators similarly. However, instead of B taking a seed and transforming it into a number, we now map S into random sets of message exchanges (i.e., a protocol). Instead of using primality as a test of whether to keep an output, we use a formal protocol verifier to see if it satisfies particular security properties such as *two party authentication* or *key exchange.* As above, our set of seeds forms a *seed space* and the set of protocols surviving the security test forms a *valid output space*, or in this case, a *valid protocol space.*

We can now posit a *Protocol Generating Function* (PGF) described by the following algorithm in Figure 1 and illustrated in Figure 2. The s is a random seed in Figure 1, and given ways to define B and F (see the following subsections), the question of choosing a protocol at random is solved.

Two challenges of injecting heterogeneity in design and use cryptographic protocols are:

- **Protocol validness:** Our automatically generated protocols should be valid and at least as secure as manually designed protocols. To achieve this, we propose the use of a high quality protocol verifier. We chose to use Athena as our protocol verifier for two-party authentication protocols because of its quality and efficiency in verifying authentication protocols. However, our general scheme also allows other verifiers and other types of protocols to be used.

- **Protocol space size:** Ideally, the protocol space should be sufficiently large that an adversary with

[1]We assume our pseudo-random number generation is strong in the sense of Yao [44], namely the sequence of values is both uniformly distributed and cryptographically secure. That is, given $s_0, \ldots, s_m$ no polynomial time algorithm can predict s_{m+1} with probability greater than $1/n + \epsilon$ for any $\epsilon > 0$.

```
Input: seed $s$
Output: protocol $p$

repeat {
    $p = B(s)$;
    $s = F(s)$;
} until ($p$ satisfies security conditions)
```

Figure 1: Protocol generation function

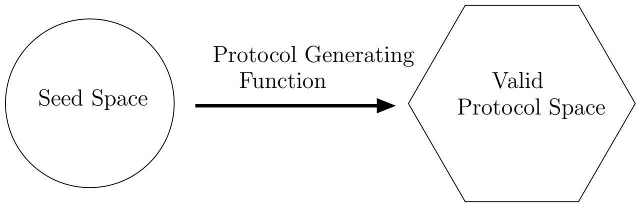


Figure 2: Overview of random protocol generation

high probability is unable to guess the protocol during the period of time it is used. However, increasing the protocol space also has a drawback: a large protocol space will, by necessity, include protocols that are longer than the most efficient protocol, increasing the overhead involved. As we show below, this increase in overhead is small.

The procedure of picking a protocol randomly and uniformly from the space of valid protocols is illustrated in Figure 3. In our simulation, both the early pruning and the full protocol verification steps use the corresponding steps built into Athena.

2.1 Efficiency of Protocol Generation

The protocol generation scheme proposed above raises some questions about efficiency:

- How many protocols must be generated on average before a protocol that passes our security test is found? We call this number N.
- What is the average time T to find a valid protocol?

The answer to the second question determines whether the protocol generation scheme is practical. If the verification time of a protocol is a constant, T is proportional to N. In general, the formal analysis of a security protocol is not a cheap operation. However, we do not need to perform a complete analysis for each protocol enumerated. By using some simple criteria, most invalid protocols can be pruned at a very early stage. We suggest using two stages of pruning protocols — an initial fast early pruning step and a later more exhaustive verification step in protocol generation which is similar to Song et al. in [37]. The key insight here is that by using a set of efficient early pruning policies, most invalid protocols are pruned in the first stage. We call a protocol that passes early pruning a *candidate protocol*. To measure the efficiency of two-stage protocol generation, we define the following values:

- p_1: the probability that a protocol passes early pruning, i.e., the ratio of candidate protocols to all protocols.
- p_2: the probability that a candidate protocol is valid, i.e., the ratio of valid protocols to candidate protocols.
- t_1: the average time required for the early pruning step. (In practice, this value is the same whether a protocol turns out to be a candidate protocol or not.)
- t_{2a}: the average verification time if a candidate protocol is invalid.
- t_{2b}: the average verification time if a candidate protocol is valid.

Now we can compute N, which is the expected number of tests before a valid protocol, and T, which is the expected time needed to find a valid protocol:

$$\begin{aligned} N &= \frac{1}{p_1 p_2}, \\ T &= (\frac{1}{p_1} - 1) \cdot \frac{1}{p_2} \cdot t_1 + (\frac{1}{p_2} - 1) \cdot (t_1 + t_{2a}) \\ &\qquad + (t_1 + t_{2b}). \end{aligned}$$

The following example illustrates the effect of early pruning in reducing T. Suppose that the verification time for both valid and invalid protocols are the same: t_{2a}=t_{2b}=t_2. If early pruning is efficient and takes 0.01% of the verification time, t_1=$10^{-4}t_2$. Let p_1=0.1 and p_2=0.5. It follows that N=20 and T=$2t_2$. Without early pruning, T=Nt_2=$20t_2$ is much longer than the result achieved with early pruning, $2t_2$. In fact, the values in this example agree with what we achieved in our simulation of two-party authentication protocols using Athena (Section 3).

2.2 Message Format

In this section, we introduce our formal representation of protocols and messages, which follows the representation used in the Athena and is generally applicable to different types of security protocols with minor changes.

A message consists of a sender, a receiver and a message body, as shown in the following:

$$\mathsf{Sender} \rightarrow \mathsf{Receiver} : \mathsf{MsgBody}$$

We follow Athena [38] in defining MsgBody (see Figure 4). Data represents a piece of public data such as the names of the participating principals in the protocol. IndData contains generated independent data, for example, nonce values. Data and IndData are two types of Variables. Key refers to the general cryptographic keys including asymmetric public keys, asymmetric private keys and symmetric keys. Variable and Key are two types of

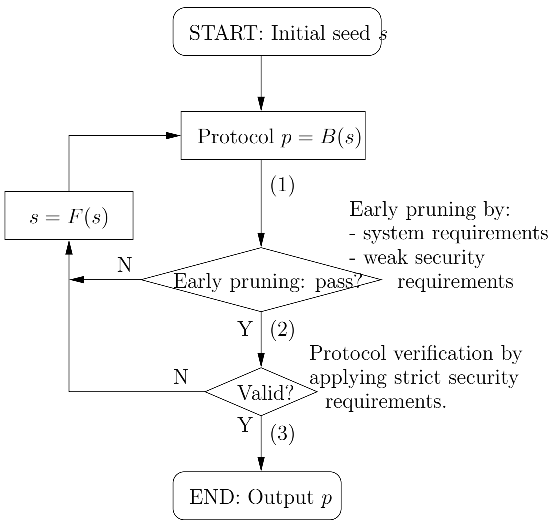


Figure 3: Protocol generation

$$\begin{aligned} \mathsf{MsgBody} &::\equiv \mathsf{Term}[,\mathsf{Term}...] \\ \mathsf{Term} &::\equiv \mathsf{Variable} \mid \{\mathsf{MsgBody}\}_{\mathsf{Key}} \\ \mathsf{Variable} &::\equiv \mathsf{Data} \mid \mathsf{IndData} \\ \mathsf{Key} &::\equiv \mathsf{PublicKey} \mid \mathsf{PrivateKey} \mid \mathsf{SymmetricKey} \end{aligned}$$

Figure 4: The format of MsgBody

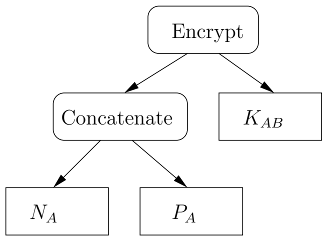


Figure 5: Example: the message tree of $\{N_A, P_A\}_{K_{AB}}$

atomic terms. Term[, Term...] is a message body with one or more terms. Finally, $\{\mathsf{MsgBody}\}_{\mathsf{Key}}$ indicates that MsgBody is encrypted using Key.

For example, $\{N_A, P_A\}_{K_{AB}}$ is a valid message body, where P_A is Data, the name of principal A in the protocol, N_A is IndData, the nonce generated by A, and K_{AB} is the symmetric key shared between the two principals A and B. A message can also be represented using a message tree as shown in Figure 5. The depth of a message tree is called *message depth*. The message depth of the message shown in Figure 5 is 3. The number of *rounds* in a protocol is the number of messages transmitted between the principals.

Given a variable set, $\{P_A, P_B, N_A, N_B\}$, several three-round protocols are randomly selected from all protocols. These are shown as examples in Figure 6. All five protocols have three messages (round=3). Each message in the protocols is formed by concatenation or/and encryption of variables P_A, P_B, N_A, and N_B. We will refer to these examples again to illustrate the protocol generation, pruning and verification steps in the following sections.

2.3 Enumerating All Protocols

In Figure 6, we illustrate some protocols from the space of all protocols created from the set of variables $\{P_A, P_B, N_A, N_B\}$. In general, we can enumerate all protocols from a set of variables as follows. First, we enumerate all possible messages that can be formed from these variables. Given a maximum message depth, the number of all possible messages will not go to infinity. Then, a protocol is composed of randomly selected messages.

The size of a protocol space is determined by the number of possible messages at each round and the total number of rounds allowed. Suppose that in t-round protocols, there are m_k possible messages in the k-th round ($1 \le k \le t$): $\mathsf{msg}_1^k, \cdots, \mathsf{msg}_{m_k}^k$. Then the size of the protocol space is: $m_1 \times m_2 \times \cdots \times m_t$.

A protocol ID is defined as

$$s_i = (\mathsf{msg}_{i_1}^1, \mathsf{msg}_{i_2}^2, \cdots, \mathsf{msg}_{i_t}^t),$$

where $\mathsf{msg}_{i_k}^k$ is the i_k-th message in the k-th round. That is, protocol ID s_i is a vector of all its message ID's. s_i is used as an input to the function B, which generates a protocol from a random value.

A *variable set* for a message round is the set of Variables that can appear in a message of that round. For example, in a two-party authentication protocol, the variable set for the first round is $\{P_A, P_B, N_A\}$ and for the second round is $\{P_A, P_B, N_A, N_B\}$ (see Figure 6 for an example).

For a message round, given a set of Variables (VarSet), a set of Keys (KeySet), and maximum message depth, all possible messages can be enumerated in the following way. Suppose $\mathsf{VarSet} = \{v_1, v_2, \cdots, v_n\}$ and $\mathsf{KeySet} = \{k_1, k_2, \cdots, k_m\}$. The number of all possible messages of depth d is represented as $M(d)$.

$$\begin{aligned} \text{depth} = 1 : M(1) &= |\mathsf{VarSet}| = n \\ \text{depth} = 2 : M(2) &= |\text{concatenation of msgs of depth 1}| \\ &\quad +|\text{encryption of msgs of depth 1}| \\ &= (2^n - 1) + n * m \\ &\quad \ldots\ldots \\ \text{depth} = d : M(d) &= 2^{M(d-1)} - 1 + M(d-1) * m \end{aligned}$$

For example, the first message in a two-party authentication protocol is composed of a three variables set $\mathsf{Var} = \{P_A, P_B, N_A\}$ and a key set $\mathsf{Key} = \{K_{AB}\}$. Then, $M(1) = 3$, $M(2) = 2^3 - 1 + 3 * 1 = 10$, $M(3) = 2^{10} - 1 + 10 * 1 = 1033$, $M(4) = 2^{1033} - 1 + 1033$.

The calculations above show that the number of possible messages increases (more than) exponentially as the maximum message depth increases, which in turn increases the complexity of the message. However, we can

$A \to B : P_A, P_B$	$A \to B : P_A, P_B$	$A \to B : N_A$	$A \to B : N_A, P_A$	$A \to B : N_A, P_A$
$B \to A : N_A$	$B \to A : \{N_B, P_A\}_{K_{AB}}$	$B \to A : N_B, \{N_A\}_{K_{AB}}$	$B \to A : \{N_A, N_B, P_B\}_{K_{AB}}$	$B \to A : \{N_A, N_B, P_B\}_{K_{AB}}$
$A \to B : N_B$	$A \to B : N_A$	$A \to B : N_B, \{P_B\}_{K_{AB}}$	$A \to B : N_B$	$A \to B : \{N_A, N_B\}_{K_{AB}}$
(a)	(b)	(c)	(d)	(e)

Figure 6: Example: Three-round protocols formed by $\{P_A, P_B, N_A, N_B\}$. (a) is eliminated because it is unreasonable; (b) fails in early pruning; (c) is eliminated by the protocol verifier; (d) and (e) are valid outputs of our random protocol generator given "two-party authentication" as security specification

greatly reduce the number of all possible messages if we apply restrictions. For example, the messages enumerated above may have redundancy, e.g. terms encrypted by the same key more than once. Restrictions can be added to remove some redundancies. It is easy to prove that the maximum message depth is $2 \times |\mathsf{KeySet}| + 2 = 2m + 2$, if we disallow multiple-encryption by the same key (see Appendix A).

2.4 Review of Athena

Athena is a formal verifier for authentication protocols. In this section, we very briefly review the two steps that we use from it: early pruning and protocol verification.

2.4.1 Preliminary Pruning

A protocol is *unreasonable* if some of its messages cannot be generated by the principal who executes it, and reasonable otherwise. Protocol (a) in Figure 6 is an unreasonable protocol. In the second message of protocol (a), B cannot send nonce N_A because B does not yet know it, since N_A is generated by A. Unreasonable protocols are eliminated early, and only reasonable protocols are subjected to protocol level pruning as discussed below.

A candidate protocol is a protocol which passes some simple pruning policies. The pruning rules depend on the security requirements of the final valid protocol. We use two-party authentication protocols to illustrate these pruning rules. The pruning techniques we use here are taken from [38]. Pruning is done at two levels: the *message level* and the *protocol level*.

Message-level pruning eliminates messages which are syntactically undesirable. For example, if we wish to exclude multiple encryptions, we would screen for messages encrypted more than one time by the same key, for example, $\{\{N_A\}_{K_1}\}_{K_1}$, or more strictly, $\{\{N_A\}_{K_1}\}_{K_2}$. A message with the features above are referred to as *ill-formed messages* as compared to *well-formed* messages.

Protocol level pruning examines each message in the protocol, viewed as a single whole. In an authentication protocol, the protocol level pruning screens for two obvious flaws:

- Impersonation attacks. Every message sent between principals potentially increases the knowledge of a third party eavesdropper. If after any number of rounds, we have a message that could be generated by an eavesdropper, the protocol is considered flawed and is then eliminated.
- Replay attacks. We check whether the principals use nonces correctly. For example, they should send a nonce in the first several messages and reply them correspondingly.

All candidate protocols that survive early pruning are sent to a protocol verifier, where a stronger (and slower) verification step is executed. Since pruning is fast and cheap, separating the screening into two levels offers significant advantages. For example, given "two-party authentication" as our security specification in Figure 6, protocol (b) is eliminated in early pruning because it does not pass the impersonation attack test, and protocols (c) (d) (e) pass the early pruning and are sent to the protocol verifier.

When inserting heterogeneity in other types of security protocols, we can use a different protocol verifier. For any verifier, it is possible to design a set of early pruning criteria according to the security protocol specifications, as we did with Athena.

2.4.2 Protocol Verification

We use the Strand Space method [41] for verifying candidate protocols. We briefly review the key algorithms: Given a security protocol and its security requirements, we verify whether a list of security requirements are satisfied by the protocol. For example, the security specification that "the initiator and responder authenticate each other" is expressed by the following formulas in Athena [36].

$$\begin{aligned} \forall C_P.C_P &\models (\text{initiator}[P_A, P_B, N_A, N_B] \\ &\quad \Longrightarrow \text{responder}[P_A, P_B, N_A, N_B]) \\ \forall C_P.C_P &\models (\text{responder}[P_A, P_B, N_A, N_B] \\ &\quad \Longrightarrow \text{initiator}[P_A, P_B, N_A, N_B]) \end{aligned}$$

The first formula expresses that "the initiator authenticates responder", i.e., if the protocol claims to an initiator A, that A has completed the protocol with a responder B and nonce values N_A and N_B, then a corresponding responder B must exist who has executed a protocol with A, also using nonce values N_A and N_B. The second formula similarly expresses that "the responder authenticates the initiator." The implication of the security specification is that A and B will successfully authenticate each other

Table 1: Average time to generate three round authentication protocols

Cost	≤12	≤15
The probability that a protocol passes early pruning (p_1)	0.49%	6.4%
The probability that a candidate protocol is valid (p_2)	37.72%	53.86%
Average time early pruning (t_1) (ms)	4e-3	4e-3
Average verification time for invalid protocols (t_{2a}) (ms)	14	19
Average verification for valid protocols (t_{2b}) (ms)	22	22
Expected number of tests before a valid protocol (N)	541.04	29.01
Expected time taken before a valid protocol is found (T) (ms)	47.28	38.39
T/t_{2b}	2.15	1.75

even if zero to an unlimited number of attackers exist. That is, protocols that pass the Athena verifier are secure even if zero to an unlimited attackers exist.

Protocol execution in Athena is represented as *states*. Athena transforms verification into a proof search of a state tree. Starting from an initial state, the verification process is a process of *goal-binding*. A set of *inference rules* is applied to each state and yields a set of subsequent states, growing a proof tree rooted from the initial state. If the proof tree grows to include a bad state that violates the security specification, we report that the protocol is incorrect. If the proof tree is completed with no bad states, we report that the protocol is correct.

Variables, concatenation, and encryption are each assigned a cost metric, so that every message has a cost metric. For example, each variable is assigned a single unit cost, a concatenation is assigned a zero unit cost, and each encryption is assigned a single unit cost. The cost of message $\{P_A, N_A\}_{K_A}$ is therefore $1+1+0+1 = 3$. The cost of a protocol is the sum of cost of every message in the protocol. To control protocol cost, a cost threshold is set and protocols exceeding that threshold are pruned.

Athena [36] attempts to find the protocol with lowest possible cost satisfying a security specification. Our goal is somewhat different. Instead, we wish to generate a large number of protocols that satisfy an input security specification. For example, in Figure 6, the protocols (c), (d), and (e) are sent to the verifier. Protocol (c) is eliminated because it suffers from a man-in-the-middle attack. Both (d) and (e) pass the verifier and will be used as valid protocols, even though (e) is not as efficient as (d)[2].

3 Implementation and Simulation of Two-Party Authentication

We built a random protocol generator that supports two party authentication to test our ideas. Although we used symmetric key algorithms for our test, our experience with automatic protocol generation suggests that asymmetric encryption will yield similar results.

[2]Note that Protocol (e) is actually the ISO/IEC 9787 symmetric-key three pass mutual authentication protocol [16]

Song and her collaborators implemented verification in Athena using SML [30, 31, 37, 38]. In this paper, we reimplemented it in Java to make the operation much faster and to integrate well with other parts of our system. Casual inspection suggests that our Java version is approximately several hundreds times faster than the original SML version, although the original also supported advanced user interface functions not supported in our version. The performance improvement was achieved by: 1) adding a stronger early pruning heuristic; 2) changing the order of state tree expansion in the strand space model such that invalid subtrees are cut early; 3) fixing bugs in the previous SML implementation.

Our experiments were performed on a Pentium-IV 3.0GHz running Linux with 1GB of memory. Our experiments allowed us to compute values for the parameters p_1, p_2, t_1, t_{2a} and t_{2b} (see Section 2). These are presented in Table 1 and discussed below.

3.1 Preliminary Pruning

Preliminary pruning winnows the space of all possible protocols into a smaller set of candidate protocols. As an initial step, we generated the set of all well-formed messages once and stored them onto a hard drive for later use. The number of well-formed messages are shown in the following table for the case of one to two keys and three to four variables:

	|KeySet| = 1	|KeySet| = 2
|VarSet| = 3	63	511
|VarSet| = 4	255	4095

The space of all protocols is more compact if we use only well-formed messages (see Section 2.4.1) to enumerate protocols, which eliminates the need for message level pruning. In a two-party authentication protocol using only two nonces, the variable set for the first message is (P_A, P_B, N_A), from which 63 well-formed messages are composed. The variable set for the second and later messages are a subset of (P_A, P_B, N_A, N_B), from which 255 well-formed message are composed. (If more than two nonces are used, the variable set will include more nonces and thus form more messages.) For a three-round protocol enumerated from these messages, the number of all protocols is $63 * 255 * 255 = 4096575 \approx 4.10 \times 10^6$.

Table 2: Protocol-level pruning results for 3 to 6 round two-party authentication protocols. (A) Number of reasonable protocols generated under the cost threshold; (B) Number of protocols that fail impersonation attack test; (C) Number of protocols that fail replay attack test; (D) Number of candidate protocols left (and its ratio p_1 in all protocols). Note: the sum of the (B), (C) and (D) is larger than (A), because some protocols are eliminated for both impersonation attacks and replay attacks

rounds		cost≤9	cost≤12	cost≤15	cost≤18	cost≤21	cost≤24	cost≤27
3	A	21122	210532	977607	2110418	2549249	2581571	2582823
	B	9903	132363	574493	1022907	1116873	1119390	1119392
	C	13087	64400	148237	204807	220983	222433	222447
	D	255 (0.006%)	20108 (0.49%)	262373 (6.4%)	890208 (21.73%)	1218897 (29.75%)	1247252 (30.44%)	1247488 (30.45%)
4	A	68154	1970803	2.964×10^7	2.599×10^8	1.343×10^9	3.957×10^9	6.736×10^9
	B	26435	1285501	2.227×10^7	1.898×10^8	8.745×10^8	2.207×10^9	3.267×10^9
	C	51770	776119	5.369×10^6	1.970×10^7	4.215×10^7	6.071×10^7	6.968×10^7
	D	178	54889	2.926×10^6	5.293×10^7	4.300×10^8	1.693×10^9	3.404×10^9
5	A			2.504×10^8	5.254×10^9			
	B			1.822×10^8	4.155×10^9			
	C			7.215×10^7	7.441×10^8			
	D			1.073×10^7	5.106×10^8			
6	A			1.013×10^9				
	B			6.941×10^8				
	C			3.802×10^8				
	D			2.074×10^7				

We assigned a cost metric of a single unit cost for the use of a name or nonce, and we assigned encryption a single unit cost. With these cost assignments, the total cost of a three-round authentication protocol is at most 27. All reasonable protocols under the cost threshold are sent on to protocol-level pruning.

In our experiment, we considered a variety of different values for the number of message rounds, ranging from 3 to 6. The experimental results for the number of protocols at each pruning stage are shown in Table 2.[3]

The average time for the early pruning test on a protocol was $4 \cdot 10^{-3}$ ms in our experiment. This value is proved to be almost constant, even as the number of message rounds increased or the cost threshold increased. For the case of three round protocols, the values of parameter p_1 were computed for later use (Section 3.3) according to the following formula:

$$\frac{\text{\# of candidate protocols}}{\text{\# of all protocols}} = \frac{\text{\# of candidate protocols}}{4096575},$$

where 4096575 is the number of all three-round authentication protocols shown above.

3.2 Protocol Verification

In the protocol verification step, we winnowed the space of candidate protocols to yield the space of valid protocols. We focused on finding 1) the number of valid protocols; and 2) the time taken for verification. We examined the verification of three round protocols with a cost threshold of 12 and 15.

3.2.1 The Number of Valid Protocols

Table 3 shows that the ratio of valid protocols to candidate protocols (p_2) is significant, especially when the cost threshold increases. The increase in cost reflects the increase complexity of protocols. Our experiments surprisingly suggest that complicated candidate protocols are more likely to be valid than simpler protocols.

3.2.2 Verification Time

Table 5 shows the time used per protocol in early pruning and verification. Early pruning time (t_1) is dwarfed by the time taken to verify a candidate protocol (t_{2a} or t_{2b}). That is, $t_1 \ll t_{2a}$ and $t_1 \ll t_{2b}$.

Median verification time does not significantly increase as the cost threshold increases. However, this is not true for the mean verification time — a small fraction of the candidate protocols require an extremely long time for verification. The distribution of verification time with cost threshold 12 is shown in Figure 7 and 8. For the cost threshold of 15, the distribution is similar. (Note that both diagrams abbreviate the x axis since only a small number of protocols require more than 2500 ms.)

Table 5 shows that valid protocols take longer to verify than invalid ones take to fail verification. This is generally true because the verifier tests more strand states for valid protocols than for invalid protocols. Statistics for

[3]To indicate the trend, we present the results for 5-round and 6-round protocols using cost thresholds of 15 and 18. Time constraints prevented us from using higher cost thresholds that produced significantly more protocols.

Table 3: Ratio of valid protocols to candidate protocols

Cost	≤12	≤15
Candidates	20108	262373
Valid Candidates (p_2)	7585 (37.72%)	141311 (53.86%)
Invalid Candidates ($1 - p_2$)	12623 (62.28%)	121062 (46.14%)

Table 4: Number of verification states

		12	15
States # of verifying an invalid protocol	MEAN	9.3	14.68
	MEDIAN	9	9
	MAX	382	734
States # of verifying a valid protocol	MEAN	25.76	35.77
	MEDIAN	12	13
	MAX	755	755

Table 5: Time used in protocol generation

		12	15
Average early pruning time per protocol - t_1		4e-3ms	4e-3ms
Verification time per protocol for invalid protocols - t_{2a}	MEAN	35.10ms	100.48ms
	MEDIAN	14ms	19ms
	MAX	9485ms	19190ms
Verification time per protocol for valid protocols - t_{2b}	MEAN	133.85ms	244.84ms
	MEDIAN	22ms	22ms
	MAX	11335ms	16715ms

the number of states and their distribution (cost $\leq$ 12) are shown in Figure 9 and 10. When the cost threshold is set at 15, the distribution is similar.

3.3 Protocol Generation Time

Given the simulation results from Tables 2, 3 and 5, we can compute two performance related parameters defined in Section 2: N, the average number of protocols that must be tested before a valid protocol is found, and T, the average time required to find a valid protocol. We are also interested in T/t_{2b}, which is the ratio of total time to the verification time of valid protocols. Table 1 shows the values of N and T for a three-round authentication protocol as the cost threshold is set to 12 and 15.

Table 5 shows that t_{2a} and t_{2b} all increase as cost threshold increases, but t_{2a}/t_{2b} remains roughly constant. This is because verification time increases proportionally for both valid and invalid protocols. Given $t_1 \ll t_{2a}$ and $t_1 \ll t_{2b}$, we have (C is a constant):

$$\begin{aligned}\frac{T}{t_{2b}} &= \frac{(\frac{1}{p_1}-1)\frac{1}{p_2}t_1 + (\frac{1}{p_2}-1)(t_1+t_{2a}) + (t_1+t_{2b})}{t_{2b}} \\ &\approx (\frac{1}{p_2}-1)\frac{t_{2a}}{t_{2b}} + 1 \approx (\frac{1}{p_2}-1)C + 1\end{aligned}$$

Therefore, T/t_{2b} decreases as p_2 increases. Table 3 show that p_2 increases as cost threshold increases, thus, we expect T/t_{2b} is even smaller for higher threshold than 15.

4 Discussion and Future Work

In this section, we explore the security benefits and tradeoffs in applying heterogeneity to the design and use of cryptographic protocols. We also discuss open problems and future work.

4.1 Application Scenarios

Heterogeneous protocols are beneficial to applications in which the failure of some machines is tolerated but the failure of all machines is not acceptable. For example, in a system designed for data redundancy and distributed storage, it may be acceptable if a number of machines (e.g., 30%) are faulty [17]. However, in order to prevent catastrophic data loss, at least one machine or a given subset of the machines must function properly. If such a system is designed using homogeneously configured machines, it is possible for all machines to fail in the presence of an automated replicated attack. To guarantee that failures among individual machines are independent, each software component in individual machines should be diversified while providing the same high-level interface.

Protocol heterogeneity is not a panacea for all attacks on security protocols. In particular, heterogeneity is designed to prevent protocol-based attacks where the attacker gain is proportional to the number of machines compromised. By significantly increasing the cost and difficulty for each attack, protocol heterogeneity can slow down, if not completely stop, automatically replicated attacks. Heterogeneity may not prevent those attacks for which it is sufficient to compromise one machine. Slowing down the attack replication process is beneficial to the distributed computing environment (e.g. the network infrastructure) and thus protects the population of computers as a whole.

Protocol heterogeneity itself is not enough to diversify machines. Rather, protocol heterogeneity can be used as a basic building block, along with other approaches to diversify software, to increase diversity in computer populations. When homogeneous protocols are used, the protocol layer is a single point of failure in systems that require diversity.

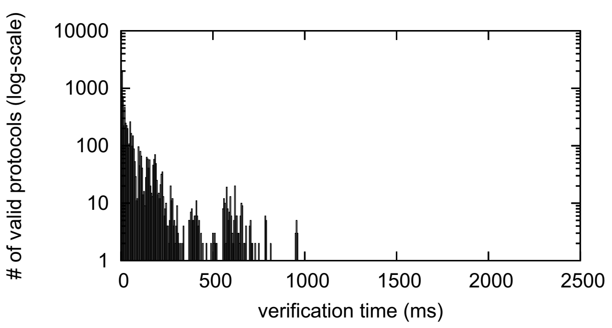


Figure 7: The distribution of verification time (t_{2b}) for valid protocols

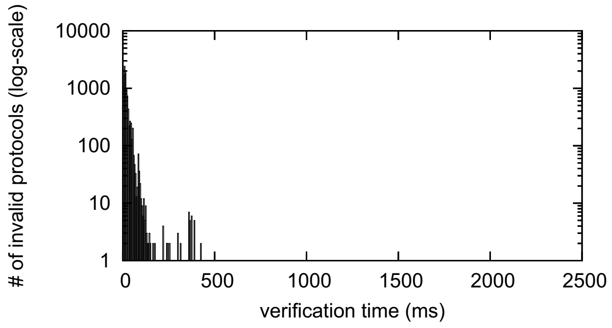


Figure 8: The distribution of verification time for (t_{2a}) invalid protocols

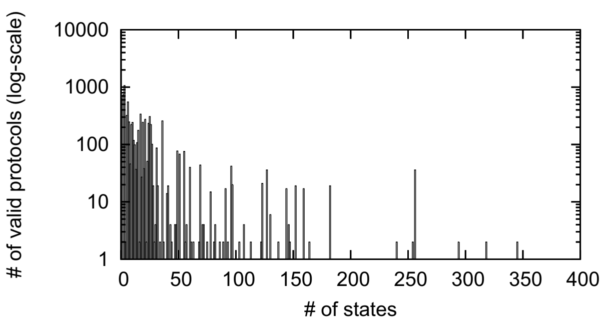


Figure 9: The distribution of number of states in verification for valid protocols

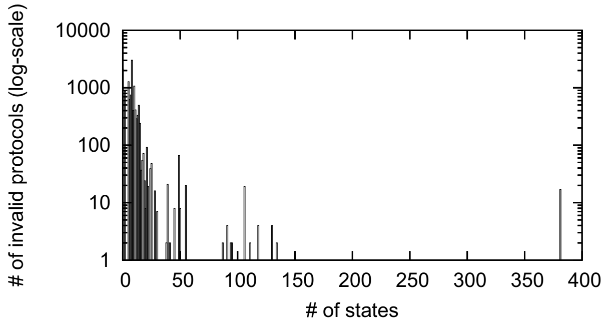


Figure 10: The distribution of number of states in verification for invalid protocols

4.2 Seed Agreement

We have shown in previous sections how a valid protocol can be generated from a seed s. In order to reach the same protocol, the communicating parties first agree on a common seed s. In most cases, choosing a random seed is simpler than other secure protocols. A seed agreement method ideally reveals the seed to both parties and yields a seed value that can not be controlled by either party. This property is important in preventing an adversary from forcing a particular protocol to be chosen out of the space of valid protocols. Below we provide alternative methods for seed creation.

In cases where the seed is public, we assume that the protocol is also public (because it is possible to derive the seed and thus the protocol). For example, if weak time synchronization is possible, the seed could simply be a function of the IP addresses of the communicating machines and the time or date. Suppose A wants to initiate a session with B, the protocol to be used is generated by a seed s as discussed in Section 2, $s = H'(\mathrm{IP}_A, \mathrm{IP}_B, \mathrm{time})$, where IP_A and IP_B are IP addresses of A and B, $H()$ is an one-way function and "time" can be "date", "hour" or "minute" depending on how fast we want the protocol to be changed and how accurate our time synchronization is. Section 3 shows that protocol generation is cheap, so the overhead of generating a new protocol is not a concern.

In this example, an attacker must generate all possible valid protocols for various times in the immediate future and discover flaws in these protocols. An attacker must have knowledge of a specific IP, and have the resources to spoof that IP address, in order to mount attack at some specific time. Then the malicious user must wait for the corresponding time to mount the attack. This attack can only succeed against a small number of machines in the whole network and only for the short time that the protocol is used.

There are at least three security benefits that can be achieved with heterogeneous protocols using public seeds. First, an attacker must spend some effort in deriving the seed and generating many protocols before attacking a protocol as usual. Second, it is very difficult to construct an automatically replicated attack across the whole network. Even if an attacker is able to mount a successful attack for one pair of communicating parties, he is not able to easily replicate that attack across the network with other parties using other short-lived protocols. Finally, since the duration of use of each protocol is short, any vulnerabilities introduced by a flawed protocol are limited, in contrast to homogeneous protocols where flaws may exist until discovered or exploited.

Thus, heterogeneous protocols do not rely on "security by obscurity". The robustness of heterogeneous protocols does not rely on keeping the seed secret or on hiding flaws in protocols. Assuming a non-perfect protocol verifier, we do not claim that protocol heterogeneity prevents all possible attacks on all peers. Instead, heterogeneity increases robustness and fault-tolerance of the entire network by making attacks more difficult to mount and replicate.

In some cases, we may want to increase the difficulty of an attack by preventing third parties from easily obtaining the seed, which requires an extra seed exchange step. Seed exchange can occur offline (in the most secure case) or through a online seed handshake protocol. Communicating parties can decide to handshake a new seed at any time, with any frequency. As an example, we present a seed exchange protocol in Appendix B. In the case of private seeds, an adversary must first discover the seed and then find the corresponding protocol before mounting an attack. For short-lived uses of protocols drawn from large protocol spaces, this is likely to be difficult or intractable.

4.3 Dependence on the Security of Protocol Generators and Verifiers

What is the impact on heterogeneous protocols if the verifier is flawed? To analyze this, suppose that a fraction p of valid protocols generated from a specific protocol generator are "truly" correct and secure. The higher the quality of a generator is, the closer p is to 100%. When we say a verifier is flawed, it means that $p < 100\%$.

In theory, automatic protocol verification tools such as Athena can be proved to be correct. However we can never claim that a specific implementation (e.g., our Athena implementation) of a formal verification method (e.g., the strand space model) is without flaws. In the case of heterogeneous protocols, a flaw in the verifier implementation may introduce some flawed protocols into the valid protocol pool, depending on p, but will not result in flaw in every protocol generated. A flawed heterogeneous protocol, like a flawed homogeneous protocol, can introduce serious vulnerabilities to communicating parties using that protocol. As discussed above, some security benefits to a population of computers may still apply because of the difficulty in replicating attacks. Reducing the duration of use of protocols will also mitigate vulnerabilities.

Another difficulty in using automated protocol generation tools is the problem of "incomplete specifications." Often, it is difficult to specify the complete list of requirements that a security protocol must meet if it is to satisfy application needs. If the specification is incomplete, flawed protocols can be introduced into the valid protocol pool. In contrast, humans design protocols based on the stated specifications, but also using their knowledge and intuition to avoid security flaws. In this sense, human-generated protocols are much more robust to "incomplete specification" failure than computer-generated protocols are.

To limit the number of flawed protocols introduced in the valid pool, several verifiers can be used together. To further introduce diversity, different sets of verification tools can be used at different points in a network. This solution may present challenges for interoperability and deserves further study.

4.4 Comparing Protocol Heterogeneity to Other Approaches

There are other approaches to achieve heterogeneity at protocol layer. For example, we can encrypt any homogeneous protocol with an exchanged (random) symmetric key. Suppose all messages in a t-round protocol $P = \{m_1, \cdots, m_t\}$ are encrypted by a symmetric key K. However, the result is another t-round homogeneous protocol $P' = \{(m_1)_K, \cdots, (m_t)_K\}$. The entire exchange of messages will be vulnerable to traffic analysis. To reduce the vulnerabilities of a single key, K must be changed often, which presents key management challenges.

Another option to introduce heterogeneity is to vary the implementation rather than the protocol. However, in practice, we can generate many more protocols than implementations. Therefore, the heterogeneity achieved by protocol randomization is much higher than that achieved with multiple implementations of a single protocol.

4.5 Heterogeneity and Standardization

When comparing the benefits of diversified software or communication protocols to standardized homogeneous software or protocols, we can not ignore the cost savings and security benefits of standardization. Homogeneous software is a larger target of attack, because a larger number of machines can be compromised with the same cost to an attacker. Homogeneous software is also subject to greater scrutiny and security analysis, because of the larger benefit that can be achieved by discovering one vulnerability (in preventing a larger number of attacks). Once a vulnerability is discovered in the homogeneous case, it is more likely to be addressed or patched, if it affects a larger number of machines.

Furthermore, heterogeneous software can impose higher interoperability and maintenance costs. In general, it is easier to maintain multiple machines if they have the same configuration. Currently, building and maintaining diversified systems is more difficult than the homogeneous case and may incur extra cost.

5 Related Work

Formal verification tools for security protocol analysis have been studied for many years (an incomplete sampling includes [5, 15, 21, 23, 24, 26, 37]). One influential direction of research investigates the use of model checking systems for protocol verification, using Murϕ [7, 8, 26], a finite-state machine verification tool. After modeling the protocol and desired properties into Murϕ language, the Murϕ system checks if all reachable states of the model satisfy a given specification through explicit state enumeration. Murϕ is a more general tool than Athena and has been used for analyzing many different types of security protocols [25, 33, 34, 35]. But Murϕ suffers from a state space explosion problem, as do other model checker methods.

Our work is related to other approaches to automatically diversify software to improve security. Forrest, Somayaji and Ackley first proposed several methods for achieving software diversity based on techniques such as adding or deleting non-functional code, randomization in code order and randomization in memory allocation [11]. Several researchers have used randomization to mitigate attacks that exploit memory programming errors, such as buffer overflows. Chew and Song [4] proposed randomization of system call mappings, global library entry points, and stack placement. Xu, Kalbarczyk and Iyer developed Transparent Runtime Randomization, modifying the Linux kernel to randomly relocate a program's stack, heap, shared libraries and parts of its runtime control data structures inside the application memory address space [42]. PaX uses address space layout randomization (ASLR) to randomize the base address of the heap, code, stack, and data regions [29]. Bhatkar, DuVarney and Sekar implement similar address obfuscations, however, they randomly transform object files and executables at link-time and load-time, instead of modifying the operating system or compilers [2]. Cowan, Beattie, Johansen and Wagle describe a compiler technique to defend against pointer related buffer overflows by randomizing (encrypting) pointers when stored in memory [6]. Barrantes et al. prevent binary code injection attacks by using randomized instruction set emulation (RISE), an instruction set obfuscation technique implemented at the machine emulator level [1]. The idea of automating diversity to secure software and systems continues to be an active area of research [3, 20]. Researchers have also argued that diversity in operating systems can improve network's resistance to distributed attacks [13].

6 Conclusion

In this paper, we argue that heterogeneity should be an important principle in design and use of cryptographic protocols. We designed a scheme for randomly generating security protocols as a method of introducing heterogeneity. In order to simulate a random protocol generator, we implemented a highly efficient protocol verifier, achieving approximately two orders of magnitude improvement in performance. In our simulation for the case of two party authentication protocols with three rounds, we found that the number of candidate protocols that passed the verifier was high (up to 54%) and that the average time to generate a random valid protocol was on the order of 38-47 ms. This leads us to believe that choosing protocols randomly out of sets numbering in the hundreds of millions is practical and achievable with an acceptable overhead.

7 Acknowledgments

This work was in part funded by the United States Postal Service and US National Science Foundation contracts EIA-01225989 and CCS-0424422. The views in this paper are those of the authors and do not necessarily reflect the views of the US government or any funding sponsor.

References

[1] Barrantes, Ackley, Forrest, Palmer, Stefanovic, and Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM CCS conference*, pp. 281–289, 2003.

[2] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th Usenix Security Symposium*, pp. 105-120, 2003.

[3] Carnegie Mellon University Cylab, *Cyberdiversity. Carnegie mellon researchers tap biology to fend off computer worms, virus attacks*, 2004.

[4] M. Chew and D. Song, *Mitigating buffer overflows by operating system randomization*, Technical Report CMU-CS-02-197, 2002.

[5] E. M. Clarke, S. Jha, and W. R. Marrero, "Partial order reductions for security protocol verification," *Tools and Algorithms for Construction and Analysis of Systems*, pp. 503-518, 2000.

[6] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium*, IEEE Computer Society Press, pp. 91-104, 2003.

[7] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *International Conference on Computer Design*, pp. 522-525, 1992.

[8] D. L. Dill, "The Murϕ verification system," in *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, Rajeev Alur and Thomas A. Henzinger, vol. 1102, pp. 390-393, Springer Verlag, New Brunswick, NJ, USA, 1996.

[9] D. Evans, *What biology can (and can't) teach us about security*, Invited talk at USENIX Security Symposium, 2004, http://www.cs.virginia.edu/ evans/talks/usenix04/.

[10] Federal Information Processing Standards Publication, *Data encryption standard (DES)*, no. 46-2, 1993.

[11] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, IEEE Computer Society Press, pp. 67-72, 1997.

[12] A. O. Freier, P. Karlton, and P. C. Kocher, *The SSL Protocol (Version 3.0)*, Mar. 1996, http://wp.netscape.com/eng/ssl3/ssl-toc.html.

[13] D. Geer, R. Bace, P. Gutmanm, P.Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier,

CyberInsecurity: The cost of monopoly. How the dominance of Microsoft's products poses a risk to security, Computer and Communications Industry Association Report, 2003, http://www.ccianet.org/filings/cybersecurity /cyberinsecurity.pdf.

[14] O. Goldreich, *Secure multi-party computation*, Final (incomplete) draft, version 1.4, 2002.

[15] N. Heintze, J. D. Tygar, J. Wing, and H. C. Wong, "Model checking electronic commerce protocols," in *Proceedings of the USENIX 1996 Workshop on Electronic Commerce*, pp. 147-164, Nov. 1996.

[16] International Standards Organization, *Information technology - Security techniques entity – Authentication mechanisms part 3: Entity authentication using symmetric techniques*, ISO/IEC 9798, 1993,

[17] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker, "The phoenix recovery system: Rebuilding from the ashes of an Internet catastrophe," in *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating System*, pp. 73-78, May 2003.

[18] J. Kelsey, B. Schneier, and D. Wagner, "Protocol Interactions and the Chosen Protocol Attack," in *Security Protocols, 5th International Workshop April 1997 Proceedings*, pp. 91-104, Springer-Verlag, 1998.

[19] V. Klima, O. Pokorny, and T. Rosa, "Attacking RSA-based sessions in SSL/TLS," in *CHES*, LNCS 2779, 3-540-40833-9, pp. 426-440, Springer-Verlag, 2003.

[20] J. Knight, *GENESIS: A farmework for achieving component diversity*, DARPA Self-regenerative Systems PI Kickoff Meeting, 2004, http://www.cs.virginia.edu/genesis /darpa.srs.kickoff.ppt.

[21] G. Lowe, " Breaking and fixing the Needham-Schroeder public-key protocol using FDR," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 1055, pp. 147-166, Springer-Verlag, Berlin Germany, 1996.

[22] U. M. Maurer, "Fast generation of prime numbers and secure public-key cryptographic parameters," *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, vol. 8, no. 3, pp. 123-155, Summer, 1995.

[23] C. A. Meadows, "A model of computation for the NRL protocol analyzer," *CSFW*, pp. 84-89, 1994.

[24] C. A. Meadows, "Formal verification of cryptographic protocols: A survey," in *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*, LNCS 917, pp. 135-150, Springer-Verlag, 1994.

[25] J. C. Mitchell, V. Shmatikov and U. Stern, "Finite-state analysis of security protocols," *Computer Aided Verification*, pp. 71-76, 1998.

[26] J. C. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using Murϕ," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 141-153, May 1997.

[27] K. D. Mitnick, W. L. Simon, and S. Wozniak, *The art of deception: Controlling the human element of security*, John Wiley & Sons 1 edition, Oct. 2002.

[28] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Communications of the ACM*, ACM Press, 0001-0782, vol. 21, no. 12, pp. 993-999, 1978.

[29] PaX Team, *Documentation for the PaX project*, 2003, http://pax.grsecurity.net/docs/index.html.

[30] A. Perrig and D. Song, "A first step towards the automatic generation of security protocols," in *Symposium on Network and Distributed Systems Security (NDSS)*, pp. 73-83, 2000.

[31] A. Perrig and D. Song, "Looking for diamonds in the desert — extending automatic protocol generation to three-party authentication and key agreement protocols," in *Proceeding of 13th IEEE Computer Security Foundations Workshop*, pp. 64-76, 2000.

[32] R. L. Rivest, A. Shamir, L. M. Adelman, *A method for obtaining digital signatures and public-key cryptosystems*, no. MIT/LCS/TM-82, pp. 15, 1977.

[33] V. Shmatikov and U. Stern, " Efficient finite-state analysis for large security protocols," in *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*, IEEE Computer Society Press, pp. 106-116, 1998.

[34] V. Shmatikov and J. C. Mitchell, "Analysis of a fair exchange protocol," in *Proceeding of the 7th Annual Symposium on Network and Distributed System Security*, pp. 119-128, 2000.

[35] V. Shmatikov and J. C. Mitchell, "Finite-state analysis of two contract signing protocols," *Theoretical Computer Science*, pp. 419-450, June 2002.

[36] D. Song, *An automatic approach for building secure systems*, Ph.D. Thesis, Berkeley, 2000.

[37] D. Song, S. Berezin, and A. Perrig, "Athena: A novel approach to efficient automatic security protocol analysis," *Journal of Computer Security*, vol. 9, no. 1/2, pp. 47-74, 2001.

[38] D. Song, A. Perrig, and D. Phan, "AGVI — Automatic generation, verification, and implementation of security protocols," in *Proceedings of 13th Conference on Computer Aided Verification (CAV)*, pp. 241-245, 2001.

[39] SSH Communications Security Inc., *Cryptography A-Z: Random number generator*, http://www.ssh.com/support/cryptography /algorithms/random.html.

[40] J. Steiner, C. Neuman, and J. Schiller, "An authentication service for open network systems," in *Proceedings of the USENIX Winter Conference*, pp. 191-202, Feb. 1988.

[41] J. Thayer, J. Herzog, and J. Guttman, "Strand spaces: Proving security protocols correct," *Journal of Computer Security*, pp. 191-230, 1999.

[42] J. Xu, Z. Kalbarczyk and R. Iyer, *Transparent runtime randomization for security*, Technical Report UILU-ENG-03-2207, 2003.

[43] A. Yao, "How to generate and exchange secrets," in *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pp. 162-167, 1986.

[44] A. C. Yao, "Theory and applications of trapdoor functions," in *23rd Annual Symposium on Foundations of Computer Science*, pp. 80-91, Nov. 1982.

Appendix A: Proof of Maximum Message Depth

In Section 2.3, we mentioned that the maximum message depth is $2 \times |\mathsf{KeySet}| + 2 = 2m + 2$, if we disallow multiple-encryption by the same key. Here we present an informal proof. Because no key can be used to do encryption twice, the most complicated message is formed by "concatenation - encryption - $\cdots$ - concatenation - encryption - concatenation - variable", where "encryption" shows up at most $|\mathsf{KeySet}|$ times. That is, in the message, there are at most $|\mathsf{KeySet}|$ of encryptions, $|\mathsf{KeySet}| + 1$ of concatenations and one level of variable. All these add up to $|\mathsf{KeySet}| + (|\mathsf{KeySet}| + 1) + 1 = 2 \times |\mathsf{KeySet}| + 2$. Thus, the maximum message depth is $2 \times |\mathsf{KeySet}| + 2 = 2m + 2$.

Appendix B: A Variant of SSL Supporting Online Seed Exchange

In Section 4.2, we discussed the use of secret seeds in automatically generating protocols. In this section, we discuss secret seeds in more detail, and we present an online common (secret) seed exchange protocol that is based on SSL and multi-party secure computation [14, 43].

In the most secure case, the seed exchange can occur offline, which is made more practical by the small size of the seed. In this scenario, two parties first agree on an initial seed $s^{(0)}$ and may also share a pseudo random function $s^{(i+1)} = f_K(s^{(i)})$ which is based on their shared secret key K. This ensures that the protocols used in future communications will only be known by the parties themselves.

We can also use an online seed handshake protocol. The frequency of seed exchange can occur once per session, once per several sessions, or can be low as only once for a pair of hosts. As one example, we present a seed exchange protocol based on SSL and multi-party secure computation below. In this protocol, the seed is derived from the contributions of both parties, which is not the case in normal SSL. After the initial seed $s^{(0)}$ is agreed upon, seeds used in next sessions are generated the same as the offline case above without executing the handshake protocol during each session. To avoid the danger of long time secrets (i.e. the sequence of $\{s^{(i)}\}$ originated from $s^{(0)}$), the two parties can restart the handshake process at some random time.

The SSL protocol [12] is a good example of protocol that supports handshaking shared secrets between communicating parties. While in many configurations it also supports secure authentication of one or more parties, one of its valid configurations supports shared secret exchange without authentication. The customized SSL handshake protocol in Figure 11 provides a general framework for sharing a secret. In this protocol, the client and server each decide the content of their own secret and share it with the other party in a secure way. However, in some cases, secrecy from outsiders is not enough. That is, the common seed also requires a contribution from both parties so that neither party has full control over the seed that is finally selected.

Suppose that A and B wish to establish a common seed s. Initially, A has secret s_A and B has s_B, where $s_A \in \{0,1\}^*$ and $s_B \in \{0,1\}^*$ are selected uniformly at random. The established common seed s should be

$$s = H(s_A, s_B),$$

where H is a one way function known by both A and B. To prevent B from controlling the selection of s, A uses secret (symmetric) key SK_A to protect its secret: $\{s_A\}_{\mathsf{SK}_A}$. Similarly, B uses $\{s_B\}_{\mathsf{SK}_B}$. A will not send B the key SK_A until it receives $\{s_B\}_{\mathsf{SK}_B}$, and vice versa. After both parties receive the encrypted secret from each other, they send out their protection key (SK_A or SK_B). This process is similar to the coin flipping problem in multi-party secure computation, where no party opens its own commitment until it receives the other commitments [14]. The seed handshake protocol is shown in Figure 12. Note that s will be unknown to an outside observer if the protocol is executed successfully. In Figure 12, $\{s_A, \{\text{checksum}(s_A)\}_{K'_A}\}_{\mathsf{SK}_A}$ and $\{s_B, \{\text{checksum}(s_B)\}_{K'_B}\}_{\mathsf{SK}_B}$ are commitments of s_A and s_B. The checksum prevents the following attack: Suppose that one of the two parties, A, is malicious, and wishes to force an initial seed value of s_0, thereby forcing the selection of protocol $p = \text{PGF}(s_0)$. A prepares a set of keys SK_i satisfying

$$Q = \{0\}_{\mathsf{SK}_0} = \{1\}_{\mathsf{SK}_1} = \cdots\cdots = \{n\}_{\mathsf{SK}_n}.$$

A sends Q as its $\{s_A\}_{\mathsf{SK}_A}$. He then defers sending his key SK_A until he receives B's key SK_B. At this point, A knows s_B. A computes $H(i, s_B)$ for every $i = 1, 2, \cdots, n$ to see if there exists a i_0 such that $H(i_0, s_B) = s_0$. A can force the common seed to be s_0 by sending SK_{i_0} as its SK_A. In our protocol, to correctly open the commitments, A and B have to send each other the original SK_A and SK_B, which were previously used for encryption. The checksum is actually in the format of $\{\text{checksum}(s_A)\}_{K'_A}$ and $\{\text{checksum}(s_B)\}_{K'_B}$, which can also be viewed as the commitments of K_A and K_B. Therefore, a malicious man-in-the-middle can not change the public key K_A and K_B to his own public keys.

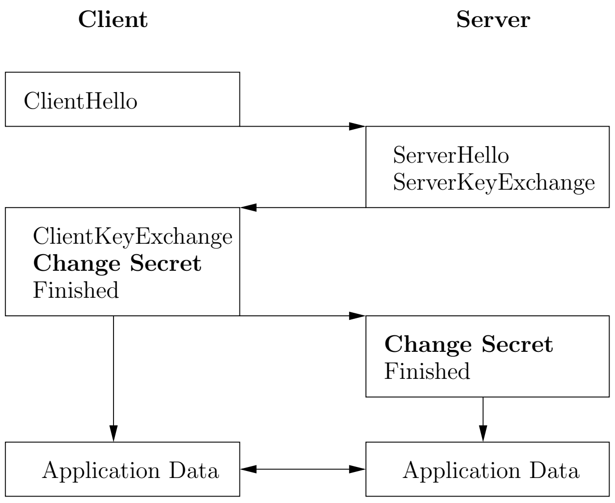


Figure 11: The customized SSL handshake protocol for shared secret exchange only

$A \to B : \left\langle \{s_A, \{\text{checksum}(s_A)\}_{K'_A}\}_{\mathsf{SK}_A}, K_A \right\rangle$

$B \to A : \left\langle \{s_B, \{\text{checksum}(s_B)\}_{K'_B}\}_{\mathsf{SK}_B}, K_B \right\rangle$

$A \to B : \{\mathsf{SK}_A\}_{K'_A}$

On receiving the message, B

- Uses K_A to decrypt SK_A;

- Use SK_A to decrypt s_A and $\{checksum(s_A)\}_{K'_A}$

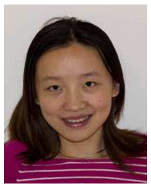
- Use K_A to decrypt checksum(s_A) and verify checksum. If not correct, protocol fails.

$B \to A : \{\mathsf{SK}_B\}_{K'_B}$

On receiving the message, A repeats what B did in the previous step.

Figure 12: The seed handshake protocol. K_A and K_B are the public keys of A and B. K'_A and K'_B are the corresponding private keys

Note that this is a seed exchange protocol, without authentication, that can be used with heterogeneous authentication protocols. If A and B successfully agree on the same s, the protocol guarantees that no outsider knows s. A man-in-middle, C, may attempt to agree on one seed, s', with A and on another seed, s'', with B. However, C will not be able to authenticate with A or B, because the heterogeneous authentication protocol will fail. For heterogeneous protocols that are not authentication protocols, an authentication component be added to the seed exchange.



Li Zhuang is a Ph.D. student at Electrical Engineering and Computer Sciences Department at U. C. Berkeley. Her research interests include privacy, networking and operating systems. Before coming to Berkeley, she got her master and bachelor degrees in Computer Science from Tsinghua University, Beijing, China.

J. Doug Tygar is Professor of Computer Science at UC Berkeley and also a Professor of Information Management at UC Berkeley. He works in the areas of computer security, privacy, and electronic commerce. His current research includes privacy, security issues in sensor webs, digital rights management, and usable computer security. His awards include a National Science Foundation Presidential Young Investigator Award, an Okawa Foundation Fellowship, a teaching award from Carnegie Mellon, and invited keynote addresses at PODC, PODS, VLDB, and many other conferences. Doug Tygar has written three books; his book *Secure Broadcast Communication in Wired and Wireless Networks* (with Adrian Perrig) is a standard reference and has been translated to Japanese. He designed cryptographic postage standards for the US Postal Service and has helped build a number of security and electronic commerce systems including: Strongbox, Dyad, Netbill, and Micro-Tesla. He served as chair of the Defense Department's ISAT Study Group on Security with Privacy, and was a founding board member of ACM's Special Interest Group on Electronic Commerce. He helped create and remains an active member of TRUST (Team for Research in Ubiquitous Security Technologies). TRUST is a new National Science Foundation Science and Technology Center with headquarters at UC Berkeley and involving faculty from Berkeley, Carnegie Mellon, Cornell, Stanford, and Vanderbilt. Before coming to UC Berkeley, Dr. Tygar was tenured faculty at Carnegie Mellon's Computer Science Department, where he continues to hold an Adjunct Professor position. He received his doctorate from Harvard and his undergraduate degree from Berkeley.

Rachna Dhamija is a Postdoctoral Fellow at the Center for Research on Computation and Society at Harvard University. Her research interests span the fields of computer security and human computer interaction. She received a Ph.D. from the School of Information Management and Systems at U.C. Berkeley in 2005. Her thesis focused on the design and evaluation of usable systems for user and server authentication. Prior to Berkeley, she worked on electronic payment system privacy and security at CyberCash.