AGVI– Automatic Generation, Verification, and Implementation of Security Protocols

Dawn Song dawnsong@cs.berkeley.edu,
Adrian Perrig perrig@cs.berkeley.edu, and
Doantam Phan dphan@hkn.eecs.berkeley.edu

University of California, Berkeley

Abstract. As new Internet applications emerge, new security protocols and systems need to be designed and implemented. Unfortunately the current protocol design and implementation process is often ad-hoc and error prone. To solve this problem, we have designed and implemented a toolkit *AGVI, Automatic Generation, Verification, and Implementation of Security Protocols*. With AGVI, the protocol designer inputs the system specification (such as cryptographic key setup) and security requirements. AGVI will then automatically find the near-optimal protocols for the specific application, proves the correctness of the protocols and implement the protocols in Java. Our experiments have successfully generated new and even simpler protocols than the ones documented in the literature.

1 Introduction

As the Internet and electronic commerce prospers, new applications emerge rapidly and require that new security protocols and systems are designed and deployed quickly. Unfortunately, numerous examples show that security protocols are difficult to design, to verify the correctness, and particularly hard to implement correctly:

– Different security protocols even with the same security properties vary in many system aspects such as computation overhead, communication overhead and battery power consumption. Therefore it is important to design *optimal* security protocols that suit specific applications. Unfortunately the current process of designing a security protocol is usually ad-hoc and involves little formalism and mechanical assistance. Such a design process is not only slow and error prone but also often miss the optimal protocols for specific applications.
– Experience shows that security protocols are often flawed even when they are designed with care. To guarantee the correctness of security protocols, we need formal and rigorous analysis of the protocols, especially automatic protocol verifiers.
– Software is notoriously flawed. Even if the design of the security protocol is correct, various implementation bugs introduced by programmers could still easily break the security of the system.

To solve these problems, we designed and implemented the *AGVI* toolkit which stands for Automatic Generation, Verification, and Implementation of Security Protocols. With AGVI, the protocol designer specifies the desired security requirements, such as authentication and secrecy, and system specification, e.g., symmetric or asymmetric encryption/decryption, low bandwidth. A *protocol generator* then generates *candidate* security protocols which satisfy the system requirements using an intelligent exhaustive search in a combinatorial protocol space. Then a *protocol screener* analyzes the candidate protocols, discards the flawed protocols, and outputs the correct protocols that satisfy the desired security properties. In the final step, a *code generator* automatically outputs a Java implementation from the formal specification of the generated security protocols.

Even a simple security protocol can have an enormous protocol space (for example, for a four-round authentication protocol, even after constraining message format and sending order, we estimate that there are at least 10^{12} possible variation protocols that one would need to consider to find an optimal one for the specific application!). Facing this challenge, we have developed powerful reduction techniques for the protocol generator to weed out obviously flawed protocols. Because the protocol generator uses simple criteria to rule out obviously flawed protocols, it is fast and can analyze $10,000$ protocols per second. Protocols that were not found flawed by the protocol generator are then send to the protocol screener which can prove whether the protocol is correct or not. Our protocol screener has the ability to analyze protocol executions with any arbitrary protocol configuration. When it terminates, it either provides a proof that a protocol satisfies its specified property under any arbitrary protocol configuration if it is the case, or it generates a counterexample if the property does not hold. It also exploits many state space reduction techniques to achieve high efficiency. On average, our protocol screener can check 5 to 10 synthesized protocols per second (measured on a 500 MHz Pentium III workstation running Linux).

We have successfully experimented with AGVI in several applications. We have found new protocols for authentication and key distribution protocols using AGVI and some of them are even simpler than the standard protocols documented in the literature such as ISO standards [Int93]. Details about the experiments and techniques in the tool can be found in [PS00a,PS00b].

2 Components in AGVI

2.1 The Protocol Generator

Our protocol generator generates candidate protocols that satisfy the specified system specification and discards obviously flawed protocols at an early stage. Intuitively, the protocol space is infinite. To solve this problem is to use *iterative deepening*, a standard search technique. In each iteration, we set a *cost threshold* of protocols. We then search through the protocol space to generate all the protocols below the given cost threshold. After sorting the protocols, the protocol screener tests them in the order of increasing cost. If one protocol satisfies the desired properties, it is minimal with respect to the cost metric function given by the user and the generation process stops. Otherwise, we increase the cost threshold and generate more protocols.

A simple three-party authentication and key distribution protocol has a protocol space of order 10^{12}. Our protocol generator generates and analyzes 10000 protocols per second, which would take over three years to explore the entire space. We have developed powerful protocol space reduction techniques to prune the search tree at an early stage. With these pruning techniques, it only takes the protocol generator a few hours to scan through the protocol space of order 10^{12}. More details are included in [PS00a,PS00b].

2.2 The Protocol Screener

We use Athena as the protocol screener [Son99,SBP00]. Athena uses an extension of the recently proposed Strand Space Model [THG98] to represent protocol execution. Athena incorporates a new logic that can express security properties including authentication, secrecy and properties related to electronic commerce. An automatic procedure enables Athena to evaluate well-formed formulae in this logic. For a well-formed formula, if the evaluation procedure terminates, it will generate a counterexample if the formula is false, or provide a proof if the formula is true. Even when the procedure does not terminate when we allow any arbitrary configurations of the protocol execution, (for example, any number of initiators and responders), termination could be forced by bounding the number of concurrent protocol runs and the length of messages, as is done in most existing automatic tools.

Athena also exploits several state space reduction techniques. Powered with techniques such as backward search and symbolic representation, Athena naturally avoids the state space explosion problem commonly caused by asynchronous composition and symmetry redundancy. Athena also has the advantage that it can easily incorporate results from theorem proving through unreachability theorems. By using the unreachability theorems, it can prune the state space at an early stage, hence, further reduce the state space explored and increase the likely-hood of termination. These techniques dramatically reduce the state space that needs to be explored.

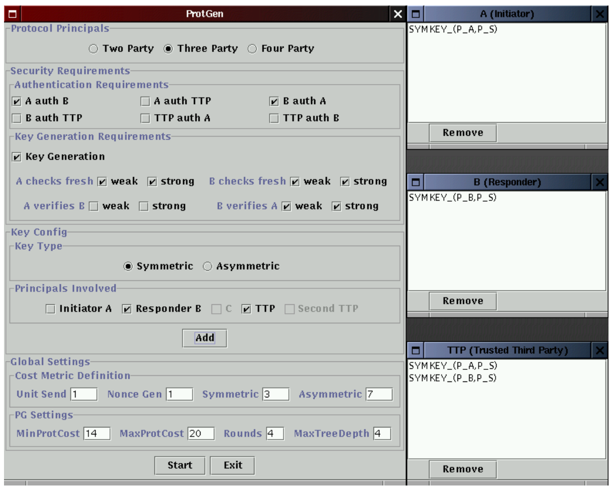
2.3 The Code Generator

Our goal for the automatic code generator is to prevent implementation weaknesses, and obtain a secure implementation if the initial protocol is secure. The code generator is essentially a translator which translates the formal specification into Java code. Given that the translation rules are correct, the final implemenation can be shown to be correct using proof by construction. In particular, we show that our implementation is secure against some of the most common vulnerabilities:

– **Buffer overruns** account for more than half of all recent security vulnerabilities. Since we use Java as our implementation language, our automatically generated code is immune against this class of attack.

– **False input attacks** result from unchecked input parameters or unchecked conditions or errors. Our automatic implementation ensures that all input parameters are carefully checked to have the right format before used.
– **Type flaws** occur when one message component can be interpreted as another message component of a different form. In the implementation, we use typed messages to prevent type flaws.
– **Replay attacks** and **freshness attacks** are attacks where the attacker can reuse old message components in the attack. Athena already ensures that the protocols are secure against these attacks. To ensure that the implementation is secure, we use cryptographically secure pseudo-random number generators to create secure nonces.

The code generator uses the same protocol description as Athena uses. The generated code provides a simple yet flexible API for the application programmer to interface with. More details about the code generator can be found in [PPS00].



```
Protocol Principals
    ○ Two Party   ● Three Party   ○ Four Party
Security Requirements
  Authentication Requirements
    ✓ A auth B        ☐ A auth TTP      ✓ B auth A
    ☐ B auth TTP      ☐ TTP auth A      ☐ TTP auth B
  Key Generation Requirements
    ✓ Key Generation
    A checks fresh ✓ weak ✓ strong   B checks fresh ✓ weak ✓ strong
    A verifies B ☐ weak ☐ strong     B verifies A ✓ weak ✓ strong
Key Config
  Key Type
    ● Symmetric   ○ Asymmetric
  Principals Involved
    ☐ Initiator A  ✓ Responder B  ☐ C  ✓ TTP  ☐ Second TTP
Global Settings
  Cost Metric Definition
    Unit Send 1   Nonce Gen 1   Symmetric 3   Asymmetric 7
  PG Settings
    MinProtCost 14   MaxProtCost 20   Rounds 4   MaxTreeDepth 4

A (Initiator)
SYMKEY_(P_A,P_S)

B (Responder)
SYMKEY_(P_B,P_S)

TTP (Trusted Third Party)
SYMKEY_(P_A,P_S)
SYMKEY_(P_B,P_S)
```

Fig. 1. AGVI GUI

3 Experiments

We have used AGVI to automatically generate and implement authentication and key distribution protocols involving two parties with or without a trusted third party. In one experiment, we vary the system aspects: one system specication has a low computation overhead but a high communication overhead and another system specication has a low communication overhead and a high computation overhead. The AGVI found different optimal protocols for metric functions used in the two different cases. In another experiment, we vary the security properties required by the system. Key distribution protocols normally have a long list of possile security properties and an application might only require a subset of the list. The AGVI also found different optimal protocols for different security requirements. In both experiments, AGVI found new protocols that are more efficient or as efficient as the protocols documented in the literature. More details can be found in [PS00a,PS00b].

Acknowledgments: We would like to thank Doug Tygar and Jon Millen for their encouragement and stimulating discussions.

References

[CJ97] J. Clark and J. Jacob. A survey of authentication protocol literature. http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz, 1997. Version 1.0.

[CJM98] E.M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *In Proceedings of the IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[Int93] International Standards Organization. *Information Technology - Security techniques — Entity Authentication Mechanisms Part 3: Entity authentication using symmetric techniques*, 1993. ISO/IEC 9798.

[Mea94] C. Meadows. The NRL protocol analyzer: An overview. In *Proceedings of the Second International Conference on the Practical Applications of Prolog*, 1994.

[Mil95] J. Millen. The Interrogator model. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 251–260, 1995.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murφ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1997.

[PPS00] Adrian Perrig, Doantam Phan, and Dawn Xiaodong Song. ACG–automatic code generation. automatic implementation of a security protocol. Technical Report 00-1120, UC Berkeley, December 2000.

[PS00a] Adrian Perrig and Dawn Song. A first step towards the automatic generation of security protocols. In *Network and Distributed System Security Symposium*, February 2000.

[PS00b] Adrian Perrig and Dawn Xiaodong Song. Looking for diamonds in the dessert: Automatic security protocol generation for three-party authentication and key distribution. In *Proc. of IEEE Computer Security Foundations Workshop CSFW 13*, July 2000.

[SBP00] Dawn Song, Sergey Berezin, and Adrian Perrig. Athena, a new efficient automatic checker for security protocols. *Submitted to Journal of Computer Security*, 2000.

[Son99] Dawn Song. Athena: An automatic checker for security protocol analysis. In *Proceedings of the 12th Computer Science Foundation Workshop*, 1999.

[THG98] F.Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of 1998 IEEE Symposium on Security and Privacy*, 1998.