Security Modeling and Analysis

A uniform conceptual framework that precisely defines system security will help analyze security, support comparative evaluation, and develop useful insight into design, implementation, and deployment decisions.

Jason Bau and John C. Mitchell
Stanford University

Computer security, as a field, is the study of how to make computer systems resistant to misuse. Some areas of computer security have established technical frameworks, such as access control, network security, and malware detection. However, those new to the area might see computer security research as a disorganized collection of disconnected efforts. Many conference papers point out a flaw in some system or design, suggest what might seem like an ad hoc repair, and wrap up without showing conclusively that the repaired system is free of further flaws. In addition, it's often unclear how to use one such point solution to solve similar problems in other systems. To make ongoing research more effective, results should be stated in a uniform conceptual framework with precise definitions. This allows progressive case studies to improve scientific and systematic engineering methods while solving specific practical problems.

Evaluating system security requires a precise definition of security. We can't answer the question, "Is this system secure?" without asking more specific questions, such as whether a network protocol is secure against man-in-the-middle attacks, or whether an access control mechanism is secure against insider attacks. Even these questions aren't fully specified because they don't tell us what it means for an attack to succeed. To assess a system's security, we must be clear about three things: system behavior, attackers' resources, and the system's security properties.

We describe a conceptual framework for defining system security and explain how modeling can help analyze security, support comparative evaluation, and develop useful insight into design, implementation, and deployment decisions.

Security Modeling and Analysis

Our security modeling and analysis framework reflects decades of research in specific areas, such as network protocol security. However, this framework has also been broadly adapted to study the security of a wide variety of systems, including custom processor architectures,[1] OS microkernels,[2] permissions models for mobile OSs,[3] and the World Wide Web platform.[4]

Security Modeling

A security model has three components:

- *System model.* We need a clear definition of the system of interest to understand how the system behaves when subjected to its intended operating conditions, as well as unintended input or operating conditions. A system model might be based on a standards document specifying behavioral requirements, a design specification, or a specific version or set of versions of source code.
- *Threat model.* A clear definition of attackers' computational resources and system access is necessary. For example, network attackers might have access to network messages but not to the internal state of hosts communicating on the network. Or, they

 1540-7993/11/$26.00 © 2011 IEEE

might have unbounded storage but insufficient computational power to break cryptography. OS attackers might be able to place malicious code in a user process but unable to modify the OS kernel.
- *Security properties.* We must clearly define the properties that we hope to prevent attackers from violating. For each behavior, such as a sequence of inputs, outputs, and state changes, we must clearly determine whether the desired security properties hold or fail.

A security model is secure if the system design achieves the desired properties against the chosen threat model. A system model might consist of a set of traces (action sequences) or some other set of possible behaviors. Some traces might occur only through actions intended by the system designers, and others might occur when attackers perform actions that aren't expected. In some cases, the desired properties might be trace properties—for every trace, the security properties either hold or fail. Such a system is secure if no trace that could arise as the result of intended or attacker actions causes any of the desired security properties to fail. Thus, no definition of security exists apart from the security model. Unless we know how a system behaves, what attackers might do, and which security properties are intended, we can't determine whether the system is secure.

Common security properties include *confidentiality* (no sensitive information is revealed), *integrity* (attackers can't destroy the system's meaningfully operable condition), and *availability* (the attacker can't render the system unavailable to intended users). However, there's no foundational understanding of why these properties are considered security properties and others aren't, and there's no standard way to decompose a given property into confidentiality, integrity, and availability components. Therefore, in future research, we should clearly define different classes of security properties and their relationships.

Security Analysis

Security models provide a basis for security analysis—the process of evaluating whether the system design achieves the desired properties against the chosen threat model. It also lets us compare the relative strengths of different system designs.

Analysts use traditional methods such as manual inspection, team discussion, and mathematical proof to examine whether a design achieves its desired goals. Formal and automated methods can also aid human reasoning and are often effective because of the complexity of many systems and the difficulty of ensuring that all details have been properly considered. Formal methods require an analysis conforming to specific rules and procedures that often originate in mathematical logic, and automated methods provide computer support for formal methods.

Two automated methods are model checkers and automated theorem provers. Model checking is a broad topic that includes tools that enumerate all possible executions of a finite-state system and symbolic model checkers. When a security model is formulated or approximated as a finite-state system, these tools are effective for finding security flaws. When abstraction methods "collapse" an infinite-state system to a finite state,[5] a finite-state tool can also demonstrate security by the absence of any sequence of attacker actions that causes the desired security properties to fail. However, model checkers are often insufficient in showing the absence of successful attacks. In contrast, automated theorem provers can establish a model's security by mathematically demonstrating that no combination of attacker actions that the threat model allows can cause the desired properties to fail.[6]

Evaluation. Security analysis evaluates models according to threats and intended security properties. Another important issue is whether these threats and properties adequately reflect practical use. Email systems are an interesting example. Originally, the system's purpose was to convey every email message to its specified address. Later, users discovered that this wasn't the complete specification for the desired system; now its recognized purpose is to carry *wanted* email from a sender to a receiver and discard or set aside spam.

Metrics. Although security models don't intrinsically provide a numeric security metric, we can compare them by comparing the relative strengths of system defenses, threat models, and security properties. For example, we can develop qualitative comparisons by ordering properties and threat models—systems satisfying a stronger security property will satisfy a weaker property in the same threat model, and systems satisfying a stronger threat model's security property will satisfy that property in a weaker threat model. We hope that future research will develop simulation relations between systems, so we can compare the strength of two different systems for the same property against the same threat model. Once we establish comparative techniques for varying the system, the threat model, and the properties individually, we can combine them to produce a multidimensional comparative security theory.

We illustrate the security modeling and analysis process using model-checking examples from network protocol security, hardware security, and Web security; however, we only scratch the surface of the topic with these examples.

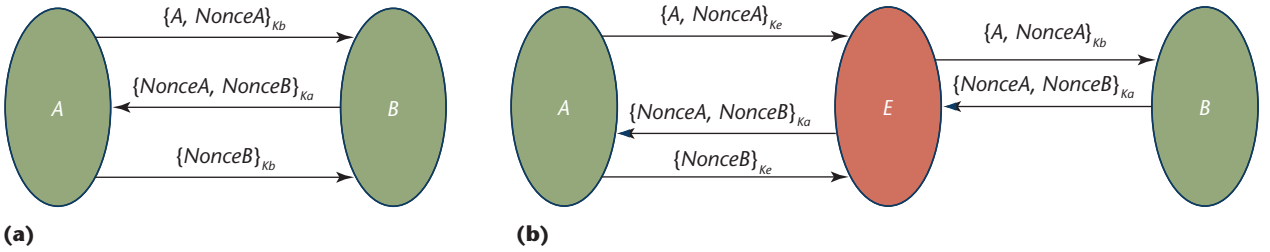

Figure 1. Needham-Schroeder protocol. (a) Legitimate parties *A* and *B* participate in the protocol by transmitting the messages indicated by the arrows. (b) Attacker *E* can compromise the protocol by intercepting and retransmitting messages. It poses as party *A* from party *B*'s perspective and gains the secrets shared between *A* and *B*.

Network Protocol Modeling and Analysis

Network protocols with security requirements are critically important to Internet security. Some well-known examples are the Secure Sockets Layer (SSL) protocol and its successor for Transport Layer Security (TLS); protocols for using wireless access points, such as Wired Equivalency Privacy (WEP) and Wi-Fi Protected Access (WPA); and secure versions of network infrastructure protocols, such as Domain Name System (DNSSEC).

Security modeling and analysis is a natural fit for studying network protocol security because

- the protocols' distributed nature makes manual reasoning about the full implications of multiparty participation difficult, and
- we can derive a protocol operation model directly from the protocol standard, making analysis results directly relevant to the standardization process.

Therefore, network protocol modeling might be the most significant and successful example of the security modeling and analysis process. It has become a robust field, with publications every year at academic conferences. For instance, the 2010 IEEE Computer Security Foundations Symposium program included publications using security modeling and analysis to verify the ad hoc mobile routing[7] and RFID[8] protocols' security properties.

Because it's easily accessible, we describe some aspects of network protocol security modeling using the Needham-Schroeder (NS) public-key protocol (see Figure 1).[9] Surprisingly, after its publication, it took nearly 10 years of academic research on protocol security before Gavin Lowe found a subtle problem with the protocol while conducting security analysis.[10] The problem isn't an actual attack on a property that the designers claimed for their protocol, but the failure of a property that many protocol users might expect. In addition, Lowe proposed a very simple modification to the protocol that clearly improves its security.

Modeling System Behavior

The first step of security modeling is to describe the protocol operations down to an appropriate detail level. The NS protocol uses public-key cryptography to exchange private random numbers, *NonceA* and *NonceB*, between two parties, *A* and *B*, without revealing them to observers. Public-key cryptography provides party *A* with a public key, K_a, and a private key, K_a^{-1}, and lets any other party use K_a to encrypt a message *M* to *A*, denoted as $\{M\}_{K_a}$, with only *A* possessing the ability to decrypt it.

This protocol has been modeled in many ways, including with formal languages[10] and finite-state enumerator languages such as Murphi.[11,12] Here, we focus on finite-state modeling. In the model, parties *A* and *B* are represented as a set of states with a set of state action rules. The states denote both the protocol's progress and the actual knowledge gained from the protocol, such as the nonce of the other party. For the NS protocol, each party stores its own nonce as well as a possible nonce from the network. Each party has three states: *initial sleep*, *waiting for response*, and *committed*.

The set of action rules "perform" the next protocol step on the basis of the current state and the validity of information received from the network. Party *A* has two state-transition rules:

- sending message 1 and proceeding to the waiting-for-response state, and
- verifying message 2 as containing *NonceA* and, if properly verified, sending message 3 and changing to the committed state.

Party *B* also has two rules:

- accepting message 1, sending message 2, and moving to the waiting-for-response state, and

- verifying message 3 as containing *NonceB* and, if properly verified, moving to the committed state.

The network is modeled as shared states between the parties, and thus each specific network message is represented as a particular network state setting.

Modeling Threat Behavior

The next aspect of security modeling is to explicitly define attackers' capabilities and operations. In network protocol security, attackers are typically given the following abilities, commonly referred to as the Dolev-Yao model and used in many studies (including John Mitchell and his colleagues' "Automated Analysis of Cryptographic Protocols Using Murphi"[12]):

- eavesdropping on any network message and breaking its content (as captured) into parts,
- recording parts of any eavesdropped packet into storage,
- removing messages from the network, and
- sending network messages containing new or eavesdropped content to any legitimate party.

The attackers' knowledge, with which they might forge network messages, is formalized as the union of a set of initial knowledge, such as public keys and participants' names, and the data obtained from eavesdropping. Also, each attacker capability is distinctly represented as a rule reading or manipulating network state, similar to legitimate participants' protocol steps. The ability to represent each attacker action in a fine-grained manner enables direct comparison of threat models.

Modeling Cryptography

Because network protocols use cryptography, we must include it in the model. One simple but surprisingly effective approach involves idealized cryptography. In a model with idealized cryptography, attackers can't compromise cryptographic protections, such as encryption and signatures, without the appropriate key. In the NS model, attackers can only record and replay the ciphertext form of encrypted data captured from the network and can't compromise the plaintext content. Attackers can also create ciphertext using their own private key, assuming proper decryption can only be performed using the attackers' identifying public key.

Modeling Security Properties

The third aspect of security modeling is to represent the security properties in the modeling framework. Security properties conveying integrity or confidentiality are typically expressed as invariants—logical expressions on the state of the model that must be guaranteed. For example, in the NS model, the integrity invariants specify that, for party *A*, reaching the committed state means that the accepted secret is *NonceB*—and vice versa for party *B*. The NS model's secrecy invariant will specify that no attackers can decrypt and learn secrets from any intended parties, despite their expressed capabilities.

Integrity and secrecy invariants are common in many protocols' security models. Availability or liveness, on the other hand, are often more difficult to express, especially in a finite-state model.

Protocol Vulnerabilities and Fixes

Figure 1b illustrates the weakness uncovered using the automated model-checking tools for the NS protocol. The tool finds a protocol-execution trace where the attacker *E* learns secrets meant to be kept between parties *A* and *B* by acting as a man in the middle. The discovery of this trace triggers violations of the model's secrecy and integrity invariants.

Beyond simply finding vulnerabilities, security modeling and analysis can also verify fixes for these vulnerabilities—we simply add the fixing protocol feature to the model, then recheck against attacker capabilities to ensure that the previously violated security properties are now inviolate. For NS, the fix—which requires party *B* to send its identity encrypted in message 2, and party *A* to validate that identity against message 1's intended recipient—was verified in the model as upholding the security invariants.

Hardware Security Modeling and Analysis

Researchers have used security modeling and analysis to study hardware and software system security such as the Execute Only Memory (XOM) processor architecture[3] and Google Android's permissions-based security.[5] In addition, related verification techniques, such as formal verification of software against specification and model checking for bug finding (which differ from our security modeling and analysis definition by the omission of a threat model), have been fruitful academic fields, producing interesting results such as the full verification of the seL4 OS microkernel[4] and the discovery of serious bugs in widely used file systems.[13]

In this example, we focus on the XOM processor's architecture to further illustrate the security modeling and analysis framework's adaptability. XOM is a generic microprocessor architecture that maintains secure memory compartments for programs while assuming attacker control over privileged code, such as OSs.[3] XOM tries to guarantee that a user program's memory integrity would be equivalent to making the program the only code executing on the machine. To

Table 1. Modeled XOM instructions.

User-level instructions		Kernel-level instructions	
Instruction	**Description**	**Instruction**	**Description**
Register use	Read a register	Register save	Encrypt a user register into another register
Register define	Write a register	Register restore	Decrypt an encrypted user register
Store	Store register to memory	Prefetch cache	Move data from memory into cache
Load	Load register from memory	Write cache	Overwrite data in the cache
		Flush cache	Flush cache line into memory
		Trap	Interrupt user
		Return from trap	Return execution to user

this end, XOM provides a tamper-resistance property that guarantees other privileged programs, including operating systems, won't read or manipulate user program data without detection.

XOM Operational Overview

XOM creates a tamper-resistant memory hierarchy by tagging data at the processor, register, and cache levels and encrypting data in the main memory.

Each user program in an XOM machine has a unique key, called a *compartment key*, associated (one to one) with an XOM ID tag. The program is initially encrypted with the compartment key, and when the code executes, it's read from memory, decrypted, and tagged with the program's XOM ID. Any on-chip data or code that belongs to a program is also tagged with that program's XOM ID. The tag identifies the data writer and thus determines who can read the data. The XOM machine tags data with the XOM ID as a proxy for encrypting it, deferring the encryption to when the data leaves the chip boundary to be stored in memory.

When data is stored to memory, it's encrypted with the compartment key, and a hash of the data and its address is added to protect against tampering with memory values. Only a program that knows the compartment key can correctly modify or view that compartment's content. The architecture records the data writer and ensures it matches the reader. Thus, if attackers try to tamper with data by overwriting it with a faulty value, the architecture will detect a user/writer mismatch when the user program tries to read that data.

By using cryptography, XOM defends against attacks in which adversaries have subverted the OS to their needs. Although attackers' OSs can execute both privileged and unprivileged instructions, they can't forge the user's XOM ID. Thus, the XOM machine should prevent attackers from tampering with user data by checking the data's XOM ID tag against the active program's tag.

Modeling System Behavior

Similar to the network protocol model, the XOM model is divided into a set of states and state-transition functions. The modeled XOM state consists of registers, cache lines, and memory words. Modeled registers contain fields for data and the XOM ID tag—as well as two fields used when one register stores an encrypted copy of another (for example, when the OS performs a context switch)—the key, and the hash of the original register location. Modeled cache lines have fields for the data value, the tag, and the memory address. Modeled memory words have fields for the data value, the hash of the address for preventing attacks that copy ciphertext from another address, and the key—associated with the XOM ID tag—used for encryption and hashing.

The XOM model's state-transition functions essentially model the XOM architecture, which (as in the actual hardware design) manipulates the register, cache, and memory states, depending on a set of security checks, which are a function of the current state. Table 1 shows the modeled user- and kernel-level instructions.

Modeling Threat Behavior

In the XOM model, users have access to only the user-level instructions, whereas attackers have access to all user- and kernel-level instructions. Using Murphi's exhaustive state exploration capabilities, the model interleaves all possible user-instruction streams with all possible combinations of instructions by an attacker's OS, subject to ideal cryptography in which the attacker can't forge hashes or decrypt without the proper key.

Modeling Security Properties

The model expresses the two goals of tamper resistance in the XOM design: attackers can't read user data or modify it without detection. The first invariant, "no observation," states that user-created data should never be tagged with attackers' XOM ID.

The model's data fields contain only values from one of two complementary finite sets—one originating only from users and another originating only from attackers—thus enabling this type of check.

The second invariant, "no modification," checks that the user-observable state of the model with an attacker is identical to the user-observable state of a "golden" model without an attacker. This golden model is a simpler version of the full XOM model, eliminating all kernel-level instructions (and thereby the attacker) as well as cache states, which are opaque to user programs. The two models' synchronicity is guaranteed by manipulating the states of both the golden model and the full model together in the state-transition rules covering user-level instructions. Thus, the model checks the "no modification" invariant by ensuring that the user-observable state—which amounts to only the registers in XOM's load/store reduced-instruction-set computing (RISC) architecture—is identical across both models after every state transition owing to user-level instructions.

Analysis Results

Analysis using Murphi's finite-state enumerator produced two types of feedback on the XOM design. First, the model verified a finite-state form of correctness. This analysis replicated two previously known errors, uncovered two new errors, and validated design fixes for these errors. One attack trace that Murphi found let attackers replay a memory location because the write to memory and the hash calculation weren't atomic. Table 2 shows the sequence of events that leads to the attack. Analysis also validated a fix for this attack. The model containing the bug fix—making the write and the hash atomic—eliminated the previous safety property violations.

Second, we used Murphi analysis to evaluate whether any checks performed by the hardware and present in the model were extraneous. We compared system models with incrementally removed security checks from the state-transition functions. If a removed action doesn't cause a safety property violation in the new system, then the checking action is extraneous. This process found one extraneous check. When a user loads data from memory, checking that the data is encrypted with the user's key is unnecessary. It's sufficient to simply tag the register in which the data is stored with the key that encrypted the data.

Web Security Modeling and Analysis

In our third example, we attempt to abstract the World Wide Web platform into a model that serves as a basis for security analysis of several current Web mechanisms and expanded models for analyzing Web mechanisms.[6] Because of the Web's complexity, its security model is too involved for us to describe in its entirety here. Instead, we highlight one case study in which various HTTP header fields are used to defend against breaches in integrity assumptions for client-server Web sessions, which can result in attacks such as cross-site request forgery (CSRF).

Table 2. XOM error found by security model analysis.

Action	Cache	Hash	Memory
User program writes *A* to cache	*A*	*Ø*	*Ø*
Machine flushes cache	*Ø*	*A*	*H(A)*
User program writes *B* to cache	*B*	*A*	*H(A)*
Adversary invalidates cache	*Ø*	*A*	*H(A)*
User program reads memory (should be *B*!)	*Ø*	*A*	*H(A)*

CSRF and HTTP Header Defense

Briefly, CSRF is an attack in which remote adversaries commandeer users' credentials on a third-party site to perform malicious actions. Attackers control a website (attack site) with content, such as a script, that can cause victims' browsers to issue HTTP requests to a third-party target site, such as a bank. If victims have valid credentials from the target site, such as a logged-in cookie, then these attackers' requests to the target site will carry these credentials and might confuse the target into granting an action, such as a funds transfer.

CSRF is an example of a breach in a user session's integrity assumption: only users' willful interaction with a site will cause the site to manipulate their account. Several proposed defenses for this type of attack require sites to check the HTTP request header for evidence that the request legitimately resulted from user action.[14] One form of this defense uses the *referer* field, which carries the full URL of the webpage that caused the request. In this scenario, this field would contain a reference to a page located at the attack site, allowing the target site to reject the request. Because the referer field is sometimes suppressed—for example, because of proxying or for privacy—researchers proposed another HTTP header, *origin*, which indicates the domain, instead of the full URL, that caused the HTTP request.[14]

Modeling System Behavior

The Web security model's implementation is expressed in Alloy, a logical language that allows a higher-level expression of the model than the more literal finite-state description languages in the previous examples.

The formal Web model in this example describes what could occur if a user navigates the Web and visits sites according to the Web's design intention. Many details regarding the Web must be modeled to

analyze a simple mechanism, such as the header validation defenses for CSRF. To effectively model the browser and its interaction with the attack page, the model uses the ScriptContext concept, which embodies the execution environment for a remote script in a client browser. ScriptContext is parameterized by the set of HTTP requests and responses it has generated and the executing script's origin. Origins are parameterized by DNS name, port, and so on. DNS names exist as a many-to-many relationship to servers at network locations to capture the mechanisms for DNS resolution.

The model also includes networks as the medium of communication between browsers and servers. It models these communications, basically HTTP requests and responses, with significant internal structure. Most relevant to our header validation defenses against CSRF is the retention of many HTTP semantics, such as response codes (`ok` and `redirect`) and headers (`referer` and `origin`).

A final relevant detail of the Web model is the designation of Web roles. The model contains principals—which own a set of DNS names and servers and are either malicious or legitimate—and browsers, which stand for individual users. All HTTP requests and responses record all the principals and browsers that helped generate them in a causal chain.

Modeling Threat Behavior

This model includes three distinct attacker types: a Web attacker, an active network attacker, and a gadget attacker. The most relevant threat model for CSRF is the Web attacker. A Web attacker operates a malicious website and might use a browser, but can see only requests or responses directed to the hosts it operates. Active network attackers have all the abilities of Web attackers plus the ability to eavesdrop, block, and forge network messages, and gadget attackers can inject certain content into otherwise honest Web sites.

Modeling Security Properties

This model formulates two widely applicable security goals that we can evaluate for various mechanisms:

- new mechanisms shouldn't violate any of the common practices that websites have come to rely on as invariants, and
- a mechanism should exhibit session integrity—attackers must be completely unable to cause honest servers to undertake harmful actions.

The session integrity condition prohibits instantiations in which an HTTP request or response that its recipient considers legitimate was in fact generated with an attacker principal in the causal chain.

Analysis Results

Using Alloy to conduct security analysis, we discovered that HTTP redirects violated the security model's session integrity condition, especially as it pertains to CSRF defense. The referer field's semantics only captures the site that originated the request and omits intermediate redirects. Thus, it's possible for an attacking site to include itself undetected in an HTTP request's causal chain by redirecting a request originally targeted at it to a victim site.

Interestingly, the analysis also found that the proposed origin header had the same drawbacks as the referer header in neglecting redirects. After this analysis, the researchers eliminated the possibility of such attacks by updating the origin header to record all domains involved in redirects.[15] In test runs, Alloy verified that sites using the updated origin header properly disregarded any requests with attackers in their causal chain, thus maintaining session integrity.

Through these brief examples of security modeling and analysis, we've described a framework that provides a scientific basis for defining, evaluating, and comparing computing systems' security. We hope that by grounding the diverse range of ongoing security work in a uniform conceptual framework, future research will prove more effective, and results will prove more widely applicable as solutions for the security community.

Our continuing research aims to use this framework to analyze and improve the security of complex and important platforms, such as cloud computing and the Web. We've also made security modeling and analysis a part of our security curriculum at Stanford University, with a graduate-level course in which students conduct quarter-long projects performing security modeling and analysis on a wide range of real-life protocols and systems. □

Acknowledgments

We acknowledge the support of the US National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research.

References

1. D. Lie et al., "Specifying and Verifying Hardware for Tamper-Resistant Software," *Proc. 2003 IEEE Symp. Security and Privacy*, IEEE CS Press, 2003, pp. 166–177.
2. G. Klein et al., "seL4: Formal Verification of an OS Kernel," *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles*, ACM Press, 2009, pp. 207–220.
3. W. Shin et al., "Towards Formal Analysis of the Permission-Based Security Model for Android," *Proc. 5th Int'l Conf. Wireless and Mobile Comm.*, IEEE CS Press, 2009, pp. 87–92.

4. D. Akhawe et al., "Towards a Formal Foundation of Web Security," *Proc. 23rd IEEE Computer Security Foundations Symp.*, IEEE CS Press, 2010, pp. 290–304.
5. C. Flanagan and S. Qadeer, "Predicate Abstraction for Software Verification," *Proc. 29th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (POPL 02), ACM Press, 2002, pp. 191–202.
6. L.C. Paulson, "The Inductive Approach to Verifying Cryptographic Protocols," *J. Computer Security*, vol. 6, vol. 1–2, 1998, p. 85128.
7. M. Arnaud, V. Cortier, and S. Delaune, "Modeling and Verifying Ad Hoc Routing Protocols," *Proc. 23rd IEEE Computer Security Foundations Symp.*, IEEE CS Press, 2010, pp. 59–74.
8. M. Bruso, K. Chatzikokolakis, and J. den Hartog, "Formal Verification of Privacy for RFID Systems," *Proc. 23rd IEEE Computer Security Foundations Symp.*, IEEE CS Press, 2010, pp. 75–88.
9. R.M. Needham and M.D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Comm. ACM*, vol. 21, no. 12, 1978, pp. 993–999.
10. G. Lowe, "Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR," *Proc. 2nd Int'l Workshop on Tools and Algorithms for Construction and Analysis of Systems* (TACA 96), Springer-Verlag, 1996, pp. 147–166.
11. D.L. Dill, "The Murphi Verification System," *Proc. 8th Int'l Conf. Computer Aided Verification* (CAV 96), Springer-Verlag, 1996, pp. 390–393.
12. J. Mitchell, M. Mitchell, and U. Stern, "Automated Analysis of Cryptographic Protocols Using Murphi," *Proc. 1997 IEEE Symp. Security and Privacy*, IEEE CS Press, 1997, pp. 141–151.
13. J. Yang et al., "Using Model Checking to Find Serious File System Errors," *ACM Trans. Computer Systems*, vol. 24, no. 4, 2006, pp. 393–423.
14. A. Barth, C. Jackson, and J.C. Mitchell, "Robust Defenses for Cross-Site Request Forgery," *Proc. 15th ACM Conf. Computer and Comm. Security* (CCS 08), ACM Press, 2008, pp. 75–88.
15. A. Barth, "The Web Origin Concept," 26 Nov. 2010; http://tools.ietf.org/html/draft-abarth-origin.

Jason Bau *is a PhD student at Stanford University's Computer Security Lab. His research interests include analyzing and enhancing network protocol and Web application security using formal techniques and automated tools. Bau has an MEng in electrical engineering from the Massachusetts Institute of Technology. Contact him at jbau@stanford.edu.*

John C. Mitchell *is the Mary and Gordon Crary Family Professor at Stanford University's Department of Computer Science. His research focuses on Web security, network security, privacy, programming language analysis and design, formal methods, and applications of mathematical logic to computer science. Mitchell has a PhD in computer science from the Massachusetts Institute of Technology. He's editor in chief of the* Journal of Computer Security *and has been actively involved in IEEE and ACM conference organization and program committees. Contact him at mitchell@cs.stanford.edu.*